栈和队列

栈

栈的类型定义

栈（stack）是一种遵循先入后出逻辑的线性数据结构

常用操作

方法	描述	时间复杂度
`push()`	压入栈	O（1）
`pop()`	弹出栈	O（1）
`peek()`	访问栈顶	O（1）

栈的实现

1.基于线性表的顺序实现

```
//----- 栈的顺序存储表示 -----
#define  STACK_INIT_SIZE  100
#define  STACKINCREMENT   10
typedef int Status
typedef int SElemType
typedef struct {
    SElemType  *base;
    SElemType  *top;
    int  stacksize;
} SqStack;

Status InitStack (SqStack *S)
{// 构造一个空栈S
    S->base = (ElemType*)malloc(STACK_INIT_SIZE *  sizeof(ElemType));
    if (!S->base)   exit(1); //存储分配失败
    S->top = S->base;
    S->stacksize = STACK_INIT_SIZE;
    return OK;
}

Status Push (SqStack *S, SElemType e) {
    if (S->top - S->base > = S->stacksize) {//栈满，追加存储空间
        S->base = (ElemType *)realloc(S.base, (S.stacksize + STACKINCREMENT)
*  sizeof (ElemType));
        if (!S->base)   return ERROR; //存储分配失败
        S->top = S->base + S->stacksize;
        S->stacksize += STACKINCREMENT;
    }
    *S->top++ = e;//*S->top = e,(S->top)++
    return OK;
```

```
}
```

队列

第三章 栈和队列

3.1 栈和队列的定义

- **共同点**：栈和队列都是**操作受限**的线性表，其逻辑结构和基本操作集是线性表的子集。
- **不同点**：
  - **栈 (Stack)**：限定只能在表的一端（**栈顶**）进行插入和删除。
    - **特点**：**后进先出 (Last-In, First-Out, LIFO)**。
  - **队列 (Queue)**：限定在表的一端（**队尾**）进行插入，在另一端（**队头**）进行删除。
    - **特点**：**先进先出 (First-In, First-Out, FIFO)**。

3.2 栈 (Stack)

3.2.1 抽象数据类型 (ADT) 定义

```c
ADT Stack {
    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, ..., n }
    // 约定an 端为栈顶，a₁ 端为栈底。

    基本操作：
        InitStack(&S);       // 构造一个空栈 S
        DestroyStack(&S);    // 销毁栈 S
        ClearStack(&S);      // 将 S 清为空栈
        StackEmpty(S);       // 判断 S 是否为空
        StackLength(S);      // 返回 S 的元素个数
        GetTop(S, &e);       // 用 e 返回 S 的栈顶元素 (不出栈)
        Push(&S, e);         // 插入元素 e 为新的栈顶元素 (入栈)
        Pop(&S, &e);         // 删除 S 的栈顶元素，并用 e 返回其值 (出栈)
        StackTraverse(S, visit()); // 遍历栈
} ADT Stack
```

3.2.2 栈的应用举例

1. **数制转换**：
   - **原理**：$N = (N \div d) \times d + (N \pmod d)$。一个十进制数 `N` 转换为 `d` 进制数时，其各位数字是 `N` 不断对 `d` 取余数得到的。
   - **过程**：计算时，先得到的是低位的数字，后得到的是高位的数字。而输出时需要从高位输出到低位。这个“先算出的后输出”的顺序正好符合栈的 LIFO 特性。
   - **算法**：
     1. 当 `N` 不为0时，循环执行：
     2. `Push(S, N % d)` (余数入栈)
     3. `N = N / d` (更新N)
     4. 循环结束后，依次将栈中元素弹出并输出。
   - **C语言示例**：

```c
// 数制转换函数示例（十进制转八进制）
void conversion() {
    SqStack S;      // 假设使用顺序栈
    InitStack(&S);  // 初始化栈
    int N, e;

    printf("请输入一个非负十进制整数: ");
    scanf("%d", &N);

    if (N == 0) {
        printf("0");
        return;
    }

    // 当N不为0时，重复取余并入栈
    while (N) {
        Push(&S, N % 8);
        N = N / 8;
    }

    // 依次弹栈并输出，得到转换后的八进制数
    while (!StackEmpty(S)) {
        Pop(&S, &e);
        printf("%d", e);
    }
    printf("\n");
}
```

2. **括号匹配的检验**：
   - **问题**：检验表达式中的括号 `()` 和 `[]` 是否配对正确。
   - **算法思想**：
     1. 从左到右扫描表达式。
     2. 遇到**左括号** (`(` 或 `[`)，则将其**入栈**。
     3. 遇到**右括号** (`)` 或 `]`)，则检查栈顶元素：
        - 若栈为空，说明右括号多余，不匹配。

        - 若栈顶元素与当前右括号不匹配（如栈顶是 `(`，当前是 `]`），则不匹配。
        - 若匹配，则将栈顶元素**出栈**。
    4. 表达式扫描结束后，若**栈为空**，则匹配成功；否则，说明左括号多余，不匹配。
3. **行编辑程序**：
    - **问题**：模拟一个简单的行编辑器，`#` 表示退格，`@` 表示退行（清空当前行）。
    - **算法思想**：用栈来存储当前行输入的字符。
        1. 遇到普通字符，**入栈**。
        2. 遇到 `#`，如果栈不空，则**出栈**一个字符。
        3. 遇到 `@`，则**清空栈**（`ClearStack`）。
        4. 遇到换行符或文件结束符，则将栈中所有内容输出。
4. **迷宫求解**：
    - **思想**：使用“穷举求解”的回溯法。栈是实现回溯的经典数据结构。
    - **基本过程**：
        1. 从入口出发，将当前位置压入栈中，标记为已访问。
        2. 探索下一个可通且未访问过的相邻方块，将其作为新的当前位置，重复步骤1。
        3. 如果当前位置所有方向都不可通，说明走入死胡同，则**回溯**：将栈顶位置**弹出**，并以新的栈顶作为当前位置，继续探索其他未尝试的方向。
        4. 如果栈顶是出口位置，则求解成功，栈中路径即为解。
        5. 如果栈为空，说明所有路径都已尝试过，迷宫无解。
5. **表达式求值（后缀表达式/逆波兰式）**：
    - **后缀表达式**：操作符在操作数之后，如 `a b +`。它无需括号，运算顺序唯一，适合计算机处理。
    - **求值算法**：
        1. 从左到右扫描后缀表达式。
        2. 遇到**操作数**，将其**入栈**。
        3. 遇到**运算符**，则从栈中**弹出两个**操作数（注意次序，先弹出的是右操作数），进行运算，并将结果**入栈**。
        4. 扫描结束后，栈中唯一的元素就是表达式的结果。
    - **中缀转后缀**：这个转换过程也需要用到栈，用于存放运算符。
6. **实现递归**：
    - **原理**：函数调用本身就是一个栈式过程。当一个函数调用另一个函数（或自身）时，系统会创建一个“**活动记录**”（也称栈帧），包含参数、局部变量、返回地址等信息，并将其压入“**调用栈**”。函数返回时，该记录出栈。
    - **递归工作栈**：递归函数执行过程中，每一层递归调用都对应一个活动记录，这些记录构成了递归工作栈。

3.2.3 栈的实现

- **顺序栈 (Sequential Stack)**：
    - **结构**：基于数组实现。通常包含三个成员：`base` (栈底指针), `top` (栈顶指针), `stacksize` (容量)。
    - `S.top == S.base` 表示栈空。
    - `*--S.top` 用于弹栈，`*S.top++ = e` 用于压栈。
    - 当`S.top - S.base >= S.stacksize`时，栈满，需要用`realloc`扩容。
    - **C语言定义及操作**：

```c
#define STACK_INIT_SIZE 100
#define STACK_INCREMENT 10
#define OK 1
#define ERROR 0
#define OVERFLOW -2
typedef int SElemType;
typedef int Status;

typedef struct {
    SElemType *base;     // 栈底指针
    SElemType *top;      // 栈顶指针 (指向栈顶元素的下一个位置)
    int       stacksize;// 当前已分配的存储容量
} SqStack;

// 入栈操作
//先存数，再指针+1
Status Push(SqStack *S, SElemType e) {
    // 如果栈满，追加存储空间
    if (S->top - S->base >= S->stacksize) {
        S->base = (SElemType *)realloc(S->base, (S->stacksize + STACK_INCREMENT) * sizeof(SElemType));
        if (!S->base) exit(OVERFLOW); // 存储分配失败
        // 更新栈顶指针和容量
        S->top = S->base + S->stacksize;
        S->stacksize += STACK_INCREMENT;
    }
    // 将元素e压入栈顶，然后栈顶指针上移
    *(S->top) = e;
    S->top++;
    // 等价于  *S->top++ = e;
    return OK;
}

// 出栈操作
Status Pop(SqStack *S, SElemType *e) {
    // 如果栈为空，返回错误
    if (S->top == S->base) return ERROR;
    // 栈顶指针先下移，再取值
    S->top--;
    *e = *(S->top);
    return OK;
}
```

- **链栈 (Linked Stack)**:
  - **结构**：基于链表实现，通常**不带头结点**，用栈顶指针直接指向链表的头部。
  - **特点**：不存在栈满的问题，插入和删除操作仅限于链表头部。
  - **操作**:
    - **入栈**：相当于在链表头部插入一个新结点。
    - **出栈**：相当于删除链表的第一个结点。

3.3 队列 (Queue)

3.3.1 抽象数据类型 (ADT) 定义

```c
ADT Queue {
    数据对象：D = { $a_i$ | $a_i$ ∈ ElemSet, i=1, 2, ..., n, n ≥ 0 }
    数据关系：R1 = { <$a_{i-1}$, $a_i$> | $a_{i-1}$, $a_i$ ∈ D, i=2, ..., n }
    // 约定 $a_1$ 端为队头， an 端为队尾。

    基本操作：
        InitQueue(&Q);       // 构造一个空队列 Q
        DestroyQueue(&Q);    // 销毁队列 Q
        ClearQueue(&Q);      // 将 Q 清为空队列
        QueueEmpty(Q);       // 判断 Q 是否为空
        QueueLength(Q);      // 返回 Q 的元素个数
        GetHead(Q, &e);      // 用 e 返回 Q 的队头元素
        EnQueue(&Q, e);      // 插入元素 e 为 Q 的新的队尾元素 (入队)
        DeQueue(&Q, &e);     // 删除 Q 的队头元素，并用 e 返回其值 (出队)
        QueueTraverse(Q, visit()); // 遍历队列
} ADT Queue
```

3.3.2 队列的实现

- **链队列 (Linked Queue)**:
  - **结构**：基于链表实现，为了方便操作，需要同时设置**队头指针** `front` 和**队尾指针** `rear`。通常带有一个头结点。
  - `front` 指向头结点，`rear` 指向最后一个元素结点。
  - **队空条件**：`Q.front == Q.rear`。
  - **入队**：在链表尾部插入，时间复杂度 $O(1)$。
  - **出队**：在链表头部删除，时间复杂度 $O(1)$。
  - **C语言定义**:

```c
typedef int QElemType;

// 链队列结点
typedef struct QNode {
    QElemType     data;
    struct QNode  *next;
} QNode, *QueuePtr;

// 链队列结构
typedef struct {
    QueuePtr front; // 队头指针，指向头结点
    QueuePtr rear;  // 队尾指针，指向最后一个元素
} LinkQueue;
```

- **循环队列 (Circular Queue)**:
  - **目的**：解决顺序队列的“假溢出”问题（即数组尾部已满，但头部还有空间）。
  - **结构**：基于数组实现，将数组的头尾逻辑上连接起来，形成一个环。

- **指针**：包含 `base`(基地址), `front`(头序号), `rear`(尾序号)。`front` 指向队头元素，`rear` 指向队尾元素的**下一个位置**。
- **队空条件**：`Q.front == Q.rear`。
- **队满条件**：$(Q.rear+1) \pmod{MAXQSIZE} == Q.front$。为了区分队空和队满，有意牺牲一个存储单元。
- **长度计算**：$(Q.rear - Q.front + MAXQSIZE) \pmod{MAXQSIZE}$。
- **指针移动**：
  - 入队：`Q.rear = (Q.rear + 1) % MAXQSIZE;`
  - 出队：`Q.front = (Q.front + 1) % MAXQSIZE;`
- **C语言定义及操作**：

```c
#define MAXQSIZE 100

typedef struct {
    QElemType *base;   // 动态分配存储空间
    int       front; // 头序号
    int       rear;  // 尾序号
} SqQueue;

// 入队操作
Status EnQueue(SqQueue *Q, QElemType e) {
    // 判断队列是否已满
    if ((Q->rear + 1) % MAXQSIZE == Q->front) return ERROR;
    // 将元素e存入队尾
    Q->base[Q->rear] = e;
    // 队尾指针后移，取模运算实现循环
    Q->rear = (Q->rear + 1) % MAXQSIZE;
    return OK;
}

// 出队操作
Status DeQueue(SqQueue *Q, QElemType *e) {
    // 判断队列是否为空
    if (Q->front == Q->rear) return ERROR;
    // 取出队头元素
    *e = Q->base[Q->front];
    // 队头指针后移，取模运算实现循环
    Q->front = (Q->front + 1) % MAXQSIZE;
    return OK;
}
```

第三章 栈和队列

基本概念

- **栈(Stack)**：后进先出(LIFO)，插入/删除仅在栈顶进行

- **队列(Queue)**：先进先出(FIFO)，插入在队尾，删除在队头
- **操作对比**：

线性表操作	栈操作	队列操作
`Insert(L,i,x)`	`Push(S,x)`	`EnQueue(Q,x)`
`Delete(L,i)`	`Pop(S)`	`DeQueue(Q)`
随机访问	仅访问栈顶	仅访问队头/队尾

栈的实现

顺序栈（数组实现）

```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10

typedef struct {
    SElemType *base;  // 栈底指针
    SElemType *top;   // 栈顶指针
    int stacksize;    // 当前分配空间
} SqStack;

Status InitStack(SqStack *S) {
    S->base = (SElemType*)malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if(!S->base) exit(OVERFLOW);
    S->top = S->base;
    S->stacksize = STACK_INIT_SIZE;
    return OK;
}

Status Push(SqStack *S, SElemType e) {
    if(S->top - S->base >= S->stacksize) { // 栈满
        S->base = (SElemType*)realloc(S->base,
                    (S->stacksize + STACKINCREMENT) * sizeof(SElemType));
        if(!S->base) return ERROR;
        S->top = S->base + S->stacksize;
        S->stacksize += STACKINCREMENT;
    }
    *S->top++ = e;
    return OK;
}

Status Pop(SqStack *S, SElemType *e) {
    if(S->top == S->base) return ERROR;
    *e = *--S->top;
    return OK;
}
```

链栈（链表实现）

```
typedef struct StackNode {
    SElemType data;
    struct StackNode *next;
} StackNode, *LinkStack;

Status Push(LinkStack *S, SElemType e) {
    StackNode *p = (StackNode*)malloc(sizeof(StackNode));
    p->data = e;
    p->next = *S;
    *S = p;  // 新节点成为栈顶
    return OK;
}

Status Pop(LinkStack *S, SElemType *e) {
    if(*S == NULL) return ERROR;
    StackNode *p = *S;
    *e = p->data;
    *S = (*S)->next;
    free(p);
    return OK;
}
```

队列的实现

循环队列（数组实现）

```
#define MAXQSIZE 100

typedef struct {
    QElemType *base;  // 动态分配空间
    int front;        // 队头索引
    int rear;         // 队尾索引（下一个位置）
} SqQueue;

Status InitQueue(SqQueue *Q) {
    Q->base = (QElemType*)malloc(MAXQSIZE * sizeof(QElemType));
    if(!Q->base) exit(OVERFLOW);
    Q->front = Q->rear = 0;
    return OK;
}

Status EnQueue(SqQueue *Q, QElemType e) {
    if((Q->rear+1) % MAXQSIZE == Q->front) // 队满
        return ERROR;
    Q->base[Q->rear] = e;
    Q->rear = (Q->rear+1) % MAXQSIZE;  // 循环后移
    return OK;
}

Status DeQueue(SqQueue *Q, QElemType *e) {
    if(Q->front == Q->rear) return ERROR; // 队空
```

```
    *e = Q->base[Q->front];
    Q->front = (Q->front+1) % MAXQSIZE;  // 循环后移
    return OK;
}
```

链队列（链表实现）

```
typedef struct QNode {
    QElemType data;
    struct QNode *next;
} QNode, *QueuePtr;

typedef struct {
    QueuePtr front;  // 队头指针
    QueuePtr rear;   // 队尾指针
} LinkQueue;

Status EnQueue(LinkQueue *Q, QElemType e) {
    QueuePtr p = (QueuePtr)malloc(sizeof(QNode));
    p->data = e;
    p->next = NULL;
    Q->rear->next = p;
    Q->rear = p;
    return OK;
}

Status DeQueue(LinkQueue *Q, QElemType *e) {
    if(Q->front == Q->rear) return ERROR;
    QueuePtr p = Q->front->next;
    *e = p->data;
    Q->front->next = p->next;
    if(Q->rear == p) Q->rear = Q->front; // 最后一个元素
    free(p);
    return OK;
}
```

栈的应用实例

1. 数制转换

原理：$N = (N \div d) \times d + N \mod d$

```c
void conversion(int N, int d) {
    SqStack S;
    InitStack(&S);
    while(N) {
        Push(&S, N % d);
        N = N / d;
    }
    while(!StackEmpty(S)) {
        SElemType e;
        Pop(&S, &e);
        printf("%d", e);
    }
}
```

示例：$(1348)_{10} = (2504)_8$

```
N       N/8     N%8
1348    168      4
168     21       0
21      2        5
2       0        2
```

2. 括号匹配检验

算法思想：

1. 左括号入栈
2. 右括号出现时：
   - 栈空 → 多余右括号
   - 栈顶匹配 → 出栈
   - 不匹配 → 错误
3. 结束时栈空则正确

```c
Status bracketMatching(char *exp) {
    SqStack S;
    InitStack(&S);
    int i = 0;
    while(exp[i] != '\0') {
        switch(exp[i]) {
            case '(': case '[':
                Push(&S, exp[i]);
                break;
            case ')':
                if(StackEmpty(S) || GetTop(S) != '(')
                    return ERROR;
                Pop(&S, NULL);
                break;
            case ']':
                if(StackEmpty(S) || GetTop(S) != '[')
                    return ERROR;
                Pop(&S, NULL);
                break;
        }
```

```
        i++;
    }
    return StackEmpty(S) ? OK : ERROR;
}
```

3. 行编辑程序

特殊字符:

- `#`: 退格符
- `@`: 退行符

```
void LineEdit() {
    SqStack S;
    InitStack(&S);
    char ch = getchar();
    while(ch != EOF) {
        while(ch != EOF && ch != '\n') {
            switch(ch) {
                case '#': Pop(&S, NULL); break;     // 退格
                case '@': ClearStack(&S); break;    // 退行
                default: Push(&S, ch); break;       // 正常字符
            }
            ch = getchar();
        }
        // 输出有效行
        ClearStack(&S);  // 清空栈
        if(ch != EOF) ch = getchar();
    }
}
```

4. 迷宫求解（回溯法）

算法思想:

```
Status MazePath(MazeType maze, PosType start, PosType end) {
    SqStack S;  // 存储路径
    InitStack(&S);
    PosType curpos = start;
    do {
        if(Pass(curpos)) {        // 当前位置可通过
            Push(&S, curpos);     // 加入路径
            if(curpos == end) return OK;  // 到达终点

            curpos = NextPos(curpos, 1); // 向东探索
        } else {  // 当前位置不可通
            if(!StackEmpty(S)) {
                Pop(&S, &curpos);
                while(!StackEmpty(S) && curpos.direction == 4) {
                    Mark(curpos);  // 标记死路
                    Pop(&S, &curpos);
                }
                if(curpos.direction < 4) {
                    curpos.direction++;
```

```
                    Push(&S, curpos);
                    curpos = NextPos(curpos, curpos.direction);
                }
            }
        }
    } while(!StackEmpty(S));
    return ERROR;  // 无解
}
```

5. 表达式求值

表达式类型:

- 前缀式（波兰式）：$+\times ab\times -c/def$
- 中缀式：$a\times b+(c-d/e)\times f$
- 后缀式（逆波兰式）：$ab\times cde/-f\times +$

中缀转后缀算法:

```
void transformInfixToSuffix(char *infix, char *suffix) {
    SqStack S;
    InitStack(&S);
    Push(&S, '#');
    int i = 0, j = 0;
    while(infix[i] != '\0') {
        if(!isOperator(infix[i])) {  // 操作数
            suffix[j++] = infix[i];
        } else {
            switch(infix[i]) {
                case '(':
                    Push(&S, infix[i]);
                    break;
                case ')':
                    while(GetTop(S) != '(') {
                        Pop(&S, &suffix[j++]);
                    }
                    Pop(&S, NULL);  // 弹出'('
                    break;
                default:
                    while(precedence(GetTop(S)) >= precedence(infix[i])) {
                        Pop(&S, &suffix[j++]);
                    }
                    Push(&S, infix[i]);
            }
        }
        i++;
    }
    while(GetTop(S) != '#') {
        Pop(&S, &suffix[j++]);
    }
    suffix[j] = '\0';
}
```

6. 实现递归（汉诺塔）

```c
void hanoi(int n, char x, char y, char z) {
    if(n == 1) {
        move(x, 1, z);  // 移动编号1的圆盘
    } else {
        hanoi(n-1, x, z, y);  // x->y (z辅助)
        move(x, n, z);        // 移动编号n的圆盘
        hanoi(n-1, y, x, z);  // y->z (x辅助)
    }
}

// 移动函数实现
void move(char from, int id, char to) {
    printf("Move disk %d from %c to %c\n", id, from, to);
}
```

队列应用实例

1. 杨辉三角计算

递推关系：$b[j] = a[j-1] + a[j]$

```c
void YangHuiTriangle(int n) {
    SqQueue Q;
    InitQueue(&Q);
    EnQueue(&Q, 0);  // 边界哨兵
    EnQueue(&Q, 1);

    for(int i = 0; i < n; i++) {
        EnQueue(&Q, 0);  // 行结束标记
        int s = 0;
        for(int j = 0; j < i+2; j++) {
            int t;
            DeQueue(&Q, &t);
            EnQueue(&Q, s+t);  // 计算下一行元素
            s = t;
            if(j < i+1) printf("%d ", s);
        }
        printf("\n");
    }
}
```

2. 划分无冲突子集

问题描述：将集合划分为最少的子集，使同一子集中元素无冲突

```c
void divideSet(int R[][N], int n, int group[]) {
    int clash[N] = {0};  // 冲突标记数组
    SqQueue Q;
    InitQueue(&Q);
    for(int i=0; i<n; i++) EnQueue(&Q, i);

```

```
    int pre = -1, groupID = 0;
    while(!QueueEmpty(Q)) {
        int i;
        DeQueue(&Q, &i);

        if(i <= pre) {  // 需要新分组
            groupID++;
            memset(clash, 0, sizeof(clash));
        }

        if(!clash[i]) {  // 无冲突可入组
            group[i] = groupID;
            for(int j=0; j<n; j++)
                clash[j] |= R[i][j];  // 标记冲突
        } else {
            EnQueue(&Q, i);  // 重新入队
        }
        pre = i;
    }
}
```

本章要点

1. 掌握栈和队列的特性及应用场景
2. 熟练掌握栈的两种实现及栈满/空条件
3. 熟练掌握循环队列和链队列的实现
4. 理解递归算法执行时的栈状态变化
5. 掌握典型应用问题的解决方法

> 作业：3.21, 3.32（K阶斐波那契数列）

K阶斐波那契数列公式：

$$f(m)=\begin{cases}0 & m\le 0\\ 1 & m=1\\ \sum_{i=1}^{k} f(m-i) & m>1\end{cases}$$

递推优化：$f(m)=2f(m-1)-f(m-k-1)$