栈

介绍

限定仅在表尾进行插入和删除操作的线性表
逻辑结构：和线性表相同
存储结构：顺序栈和链栈
运算规则：先进后出

表尾：栈顶
表头：栈底

ADT Stack {
数据对象：$D=\{a_i|a_i\in ElemSet,i=1,2,\cdots,n,n\geq 0\}$
数据关系：$R=\{<a_{i-1},a_i>|a_{i-1},a_i\in D,i=2,\cdots,n\}$
约定a_n端为栈顶，a_1端为栈底
基本操作：
InitStack(&S)
DestroyStack(&S)
ClearStack(&S)
StackEmpty(&S)
StackLength(&S)
GetTop(S, &e)
Push(&S, e)
Pop(&S, &e)
StackTraverse(S, visit()) // 从栈底（表头）到栈顶（表尾）
} ADT Stack

顺序栈

栈底一般在低地址端

- Top指针：指向栈顶元素的下一个位置
- Base指针：指向栈底
- stacksize：栈可使用的最大容量（即索引最大到stacksize-1）

判断空栈的标准：base == top
判断栈满的标准：top - base == stacksize

容易产生的溢出：

FxMarkBrown's Blog

- 上溢（Overflow）：栈已满还有元素要入栈
- 下溢（Underflow）：栈已空还有元素要出栈

定义

```C++
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10

typedef struct {
   SElemType* base;
   SElemType* top;
   int stacksize;
} SqStack;
```

基本操作

初始化

```C++
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;
```

返回栈顶元素

```C++
Status GetTop(SqStack S,SElemType &e) {
   // 若栈不为空，用e返回S的栈顶元素
   if(S.top == S.base) return ERROR;
   e = *(S.top-1);

   return OK;
}
```

FxMarkBrown's Blog

入栈

```C++
Status Push(SqStack &S,SElemType e) {
    if(S.top - S.base >= S.stacksize) {
        S.base = (SElemType*)realloc(
            S.base,
            (S.stacksize + STACKINCREMENT)*sizeof(SElemType)
        );
        if(!S.base) exit(OVERFLOW);
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top = e;
    S.top++;
    return OK;
}
```

出栈

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

链栈

一个没有头结点的单链表
指针域

- S：指向栈顶（表尾）

判断空栈的标准：S == NULL

FxMarkBrown's Blog

基本操作

入栈

```C++
Status Push(SLinkList &S, SElemType e) {
    LNode* p = (LNode*)malloc(sizeof(LNode));
    if(!p) exit(OVERFLOW);
    p->data = e;
    p->next = S;
    S = p;

    return OK;
}
```

出栈

```C++
Status Pop(SLinkList &S, SElemType e) {
    if(!S) return ERROR;
    e = S->data;
    LNode* p = S;
    S = S->next;
    free(p);

    return OK;
}
```

应用

对四则运算表达式求值

规则：

1. 先乘除后加减
2. 同级从左到右
3. 先括号内后括号外

表达式由操作符、运算符、界限符组成，运算符和界限符统称为算符（OP）

两个相继出现算符的优先关系

- $\theta_1 < \theta_2$：θ_1优先级低于θ_2
- $\theta_1 = \theta_2$：...
- $\theta_1 > \theta_2$：...

使用两个工作栈

FxMarkBrown's Blog

- OPTR：寄存运算符
- OPND：寄存操作数

思路为：

1. 首先置操作数栈为空栈，表达式起始符“#”为运算符栈的栈底元素
2. 依次读入表达式中的每个字符，若是操作数则进OPND，若是运算符则和OPTR栈顶运算符比较优先权后作相应操作，直到整个表达式求值完毕（即OPTR栈的栈顶元素和当前读入的字符串均为“#”）

```C++
OperandType EvaluateExpression() {
    InitStack(OPTR); Push(OPTR, '#');
    InitStack(OPND); c = getchar();
    while(c != '#' || GetTop(OPTR) != '#') {
        if(!In(c, OP)) Push(OPND, c); c = getchar(); // 不是运算符则进OPND
        else {
            switch(Precede(GetTop(OPTR), c)) {
                case '<': // 栈顶元素优先权低
                    Push(OPTR, c); c = getchar();
                    break;
                case '=': // 脱括号并接受下一字符
                    Pop(OPTR, x); c = getchar();
                    break;
                case '>': // 退栈并将操作结果入栈
                    Pop(OPTR, theta);
                    Pop(OPND, b); Pop(OPND, a);
                    Push(OPND, Operate(a, theta, b));
                    break;
            }
        }
    }
    return GetTop(OPND);
}
```

函数递归的实现

递归函数的运行过程类似多个函数的嵌套调用，只是调用函数和被调用函数是同一个函数

通常当在一个函数的运行期间调用另一个函数时，在运行被调用函数之前，系统需要完成

1. 将所有的实参、返回地址等信息传递给被调用函数保存
2. 为被调用函数的局部变量分配存储区
3. 将控制转移到被调用函数的入口

而从被调用函数返回调用函数之前，系统需要完成

4. 保存被调函数的计算结果
5. 释放被调函数的数据区

FxMarkBrown's Blog

6. 依照被调函数保存的返回地址将控制转移到调用函数

依照此规则可以使用栈实现，每当调用一个函数时就为它在栈顶分配一个存储区，当其退出时就释放其存储区，则当前正在运行的函数的数据区必在栈顶

汉诺塔问题：
有x, y, z三个塔座，在x上有n个直径不同的圆盘，按直径从小到大叠放，编号$1, 2, 3, \cdots, n$。现要将n个圆盘从x移到z，叠放顺序不变，移动过程中遵循下列原则：

- 每次只能移动一个圆盘
- 圆盘可在三个塔座上任意移动
- 任何时刻，每个塔座上不能将大盘压到小盘上

即：将n个圆盘，从x座借助于y座移到z座

```C++
void hanoi(int n, char x, char y, char z) {
    if(n == 1)
        move(1, x, z); // 如果圆盘只有一个，则直接从起始柱x移动到目标柱z
    else {
        hanoi(n-1, x, z, y); // 将n-1个圆盘由起始柱x移动到辅助柱y，中间借助目标柱z
        move(n, x, z); // 将起始柱的最后一个大圆盘直接从起始柱x移动到目标柱z
        hanoi(n-1, y, x, z); // 将剩余n-1个在辅助柱y上的圆盘移动到目标柱z，中间借助起始柱x
    }
}
```

队列

队列定义

限定只能挂在表的一端进行插入，在表的另一端进行删除的数据表
逻辑结构：和线性表相同
存储结构：顺序队和链队，循环队最常见
运算规则：先进先出

ADT Queue {
数据对象：$D = \{a_i | a_i \in ElemSet, i = 1, 2, \cdots, n\ n \geq 0\}$
数据关系：$R1 = \{< a_{i-1}, a_i > | a_{i-1}, a_{\in} D\}$
基本操作：
InitQueue(&Q); DestroyQueue(&Q);
EnQueue(&Q, e); Dequeue(&Q, &e);
} ADT Queue

FxMarkBrown's Blog

链队

定义

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

typedef struct {
    QueuePtr front;
    QueuePtr rear;
} LinkQueue;
```

初始化

```C++
Status InitQueue(LinkQueue &Q) {
    Q.front = Q.rear = (QueuePtr) malloc(sizeof(QNode));
    if(!Q.front) exit(OVERFLOW);
    Q.front->next = NULL:
}
```

销毁

```C++
Status DestoryQueue(LinkQueue &Q) {
    while(Q.front) {
        Q.rear = Q.front ->next;
        free(Q.front);
        Q.front = Q.rear;
    }
    return OK;
}
```

FxMarkBrown's Blog

基本操作

入队

```C++
Status EnQueue(LinkQueue &Q, QElemType e) {
    p = (QueuePtr)malloc(sizeof(QNode));
    if(!p) exit(OVERFLOW);
    p->data= e; p->next = NULL;
    Q.rear->next = p; 
    Q.rear = p;
}
```

出队

```C++
Status DeQueue(LinkQueue &Q, QElemType &e) {
    if(Q.front == Q.rear) return ERROR;
    p = Q.front->next;
    e = p->data;
    Q.front->next= p->next;
    //需要判断尾指针
    if(Q.rear == p) Q.rear = Q.front;
    free(q);
    return OK;
}
```

顺序队

定义

```C++
#define MAXQSIZE 100

typedef struct {
    QElemType* base;
    int front;
    int rear;
} SqQueue;
```

空队rear == front
解决假溢出问题：引入循环队列

- 在入队列时：Q.base[Q.rear] = e; Q.rear = (Q.rear + 1) % MAXQSIZE;
- 在出队列时：e = Q.base[Q.front]; Q.front = (Q.front + 1) % MAXQSIZE;
- 队空：Q.front = Q.rear

FxMarkBrown's Blog

- 队满：Q.front = Q.rear
- 解决队空和队满条件相同的问题
  - 额外维护一个标志
  - 少用一个空间，用(Q.rear + 1) % MAXSIZE == Q.front判断队满
  - 此时队列长度判断可用 (Q.rear - Q.front + MAXSIZE) % MAXSIZE

基本操作

用第二种方法解决循环队列队满和队空条件一致的问题

入队

```C++
Status EnQueue(SqQueue &Q, QElemType e) {
    if((Q.rear + 1) % MAXQSIZE == Q.front) return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear + 1) % MAXQSIZE;
    return OK;
```

出队

```C++
Status DeQueue(SqQueue &Q, &e) {
    if(Q.front == Q.rear) return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front + 1) % MAXQSIZE;
    return OK;
}
```

FxMarkBrown's Blog