栈

概念

先进后出，限制在表的一端进行插入删除

设栈S=(a1, a2, ..., an), a1为栈底元素，an为栈顶元素。退栈的第一个元素为栈顶元素。
元素在进栈（压栈）过程中可以随时出栈（弹栈），所以出栈顺序不一定。

出栈排列组合个数：1/(n+1)*Cn_2n*
e.g. 对于4个元素的栈：1/(5)((8765)/(4321)) = 14

顺序栈（分为动态和静态）

动态顺序栈用一维数组存储，栈的大小可以增加，实现复杂

静态顺序栈不能增大存储空间，实现简单

顺序栈用的比较多，因为顺序栈在增删时无需移动元素，避开了顺序表插入元素时需要移动大量元素的缺点，同时顺序表的随机存储特性能提高效率。

动态顺序栈

bottom表示栈底指针，固定不变；top表示栈顶指针，随着进栈、退栈变化

空栈：top和bottom都指向第一个位置；
元素a进栈：bottom指向a，top指向a的下一个位置；
出栈：top指向栈顶元素，然后栈顶元素取出。

动态顺序栈存储的临界条件

设开辟n个空间存储栈元素：

1. 空栈：bottom == top
2. 满栈：|bottom - top| >= n
3. 元素个数：|bottom - top| 个
4. 对于指针移动操作
    - 若入栈时先移动指针再入栈，则在出栈时先出栈再移动指针
5. top的指向位置：
    - 若一开始指向合法位置：指向栈顶元素下一位置【不做特殊说明时，此项为默认】
    - 不合法：指向栈顶元素

具体情况：

1. 初始化时，top = bottom = -1
    1. bottom == -1
    2. 空栈：bottom == t == -1
    3. 入栈：t++; push(e)
    4. top指向栈顶元素

5. 出栈：pop(e); t--
6. 满栈：top == n-1
7. 个数：(top - bottom) 个

2. 初始化时：top = bottom = 0
    1. bottom == 0
    2. 空栈：bottom == top
    3. 入栈：push(e); top++
    4. top指向栈顶元素下一位
    5. 出栈：top--; pop(e)
    6. 满栈：(top-bottom) >= n
    7. 个数：(top-bottom) 个
3. 初始化时：top = bottom = n

   从上往下入栈
    1. bottom == n
    2. 空栈：bottom == top == n
    3. 入栈：t--; push(e)
    4. top指向栈顶元素
    5. 出栈：pop(e); t++
    6. 满栈：(bottom - top) >= n
    7. 个数：(bottom - top) 个
4. 初始化时：top = bottom = n-1

   从上往下入栈
    1. bottom == n-1
    2. 空栈：bottom == top == n-1
    3. 入栈：push(e); t--
    4. top指向栈顶元素的下一空位
    5. 出栈：t++; pop(e)
    6. 满栈：(bottom - top) >= n
    7. 个数：(bottom - top) 个

动态的定义和初始化，进栈和出栈

```
#define STACK_SIZE 100      //初始大小
#define STACKINCREMENT 10   //存储空间分配增量

/*定义*/
typedef struct sqstack{
    int *bottom;
    int *top;
    int stacksize;  //当前已分配空间，以元素为单位
}SqStack;

/*初始化*/
int Init_Stack(void){
    SqStack S;
    S.bottom = (int *)malloc(STACK_SIZE *sizeof(int));
    if(! S.bottom) return 0;
    S.top = S.bottom;
```

```
    S.stacksize = STACK_SIZE;
    return 1;
}

/*进栈*/
int push(SqStack S, int e){
    if(S.top->S.bottom > S.stacksize){  //如果满栈，追加存储空间
        S.bottom = (int *)realloc((S.STACKINCREMENT + STACK_SIZE)*sizeof(int));
        if(! S.bottom) return 0;
        S.top = S.bottom + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top = e;
    S.top++;
    return 1;
}   //如果在追加存储空间时，没有那么大的连续存储空间，容易报错

/*出栈*/
int pop(SqStack S, int *e){
    if(S.top == S.bottom) return 0;     //判空
    S.top--;
    *e = *S.top;
    return 1;
}
```

静态顺序栈

静态的定义和初始化，进栈和出栈

```
#define MAX_STACK_SIZE 100      //栈向量大小

/*定义*/
typedef struct sqstack{
    int stack_array[MAX_STACK_SIZE];
    int topl
}SqStack;

/*初始化*/
SqStack Iint_Stack(void){
    Sqstack S;
    S.top = ;
    return(S);
}

/*进栈*/
int push(SqStack S, int e){
    if(S.top == MAX_STACK_SIZE) return 0;   //判满
    S.Stack_array[S.top] = e;
    S.top++;
    return 1;
}
```

```
/*出栈*/
int pop(SqStack S, int *e){
    if(S.top == 0) return 0;    //判空
    S.top--;
    *e=S.stack_array[S.top];
    return *e;
}
```

对顶栈

【408还没出现（截至2022），北京大学考过】

若内存不足，可以考虑把两个栈共享一片空间

两个栈共享同一片空间；
把两个栈的栈底设在数组的两端；
|top1 - top2| == 1 表示栈满

链栈

存储结构为链式存储的栈，一种运算首先的单链表

比起顺序栈，它不会出现满栈的情况

定义和初始化，入栈和出栈

```
/*定义*/
typedef struct Stack_Node{
    int data;
    struct Stack_Node *next;
}STack_Node;

/*初始化*/
Stack_Node *Init_Link_Stack(void){
    Stack_Node *top;
    top = (Stack_Node *)malloc(sizeof(Stack_Node));
    top->next = NULL;
    return(top);
}

/*入栈*/
int push(Stack_Node *top, int e){
    Stack_Node *p;
    p = (Stack_Node *)malloc(sizeof(Stack_Node));
    if(!p) return 0;    //空间申请失败，结点未创建
    p->data = e;
    p->next = top->next;
    top->next = p;  //钩链
    return 1;
}
```

```
/*出栈*/
int pop(Stack_Node *top, int *e){
    Stack_Node *p;
    int e;
    if(top->next == NULL) return 0; //栈空，返回错误
     p = top->next;
     *e = p->data;  //取出栈顶元素
     top->next = p->next;   //修改栈顶指针
     free(p);
     return 1;
}
```

应用

考试中可以直接使用，无需定义：

top(), bottom(), push(), pop(), initStack()

数学运算中的括号匹配

读到左括号，pop()，读到右括号，push与读到的左括号匹配
匹配成功，继续读入；反之返回FALSE

```
int Match_Brackets(){
    char ch, x;
    scanf("%c", &ch);
    while(asc(ch) != 13){   //程序到回车结束
        if(ch == '(' || ch == '[') push(S, ch);
        else if(ch == ']'){
            x = pop(S);
            if(x != '[') {
                printf("括号不匹配");
                return 0;
            } else if(ch == ')'){
                x = pop(S);
                if(x != '('){
                    printf("括号不匹配");
                    return 0;
                }
            }
            if(S>top != 0){
                printf("括号数量不匹配");
                return 0;
            } else return 1;
        }
    }
}
```

进制转换（辗转相除法）

e.g. 十进制转八进制：
(D/8)%8，得到的余数进栈，
随后(D/8/8)%8，得到的余数进栈，
...
依此类推，直到D==0，将栈中元素逐个出栈，即可得到八进制数

```
void conversion(int D, int d){
    //将十进制D转为d进制数
    SqStack S;
    int k, *e;
    S = Init_Stack();
    while(D>0){
        k = D%d;
        push(S, k);
        D = D/d;
    }
    while(S.top != 0){
        pop(S, e);
        printf("%d", *e);
    }
}
```

将递归调用用栈实现

递归的最后一次调用会先进行处理（top），贴合于栈的后进先出的特性，可以借助栈来转换为非递归算法

表达式求值

e.g. 对于b-(a+5)*3 //这是一种中缀表达式*
后缀表达式：ba5+3- //使用最多 前缀表达式：-b*+a53
注意：数据顺序不限，变化的是运算符号的位置和顺序

在使用栈进行表达式求值时，设立两个栈：数栈、符栈。当符栈push进高优先级的运算符号时，数栈进行相应运算。

中缀转后缀

中缀转后缀同样需要两个栈，根据运算符的优先级，将符栈的元素pop并push进数栈