第三章 栈和队列

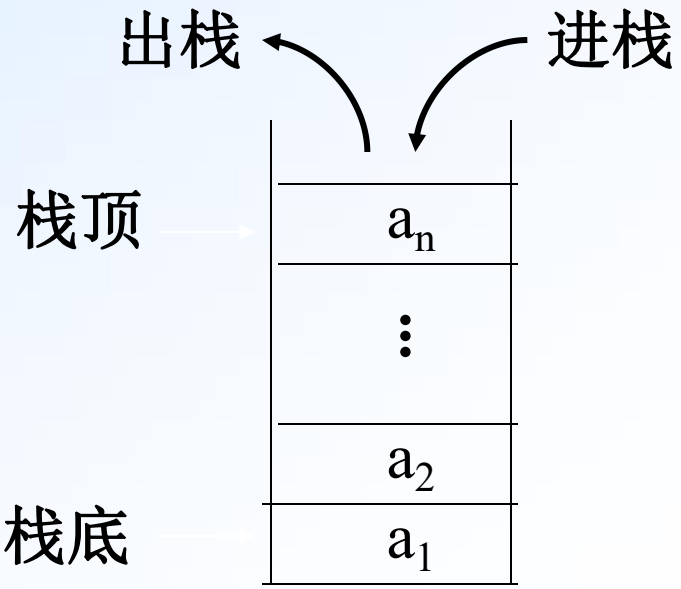
- 栈和队列也是线性表，只不过是两种**操作受限**的线性表。

- 如果从数据类型角度看，他们是和线性表大不相同的两类重要的抽象数据类型。

- 栈和队列广泛应用于各种软件系统中。

3.1 栈的基本概念

- **栈 -- 是限制仅在线性表的一端进行插入和删除运算的线性表。**
  - **栈顶（top）-- 允许插入和删除的一端。**
  - **栈底（bottom) -- 不允许插入和删除的一端。**
  - **空栈 -- 表中没有元素。**



- 栈的特点：

  后进先出（LIFO）

【例3-1】 假定有4个元素A，B，C，D，按所列次序进栈，试写出所有可能的出栈序列。注意，每一个元素进栈后都允许出栈，如**ACDB**就是一种出栈序列。

解：可能的出栈序列有ABCD，ABDC，ACBD，ACDB，ADCB，BACD，BADC，BCAD，BCDA，BDCA，CBAD，CBDA，CDBA，DCBA。

可能的出栈序列个数可用**Catalan**列计算，即 $\frac{1}{n+1}C_{2n}^{n}$

栈的基本运算

1. **初始化** — 创建一个空栈；
2. **判栈空** — 判断栈是否为空栈；
3. **进栈** — 往栈中插入（或称推入）一个元素；
4. **退栈** — 从栈中删除（或称弹出）一个元素；
5. **取栈顶元素**

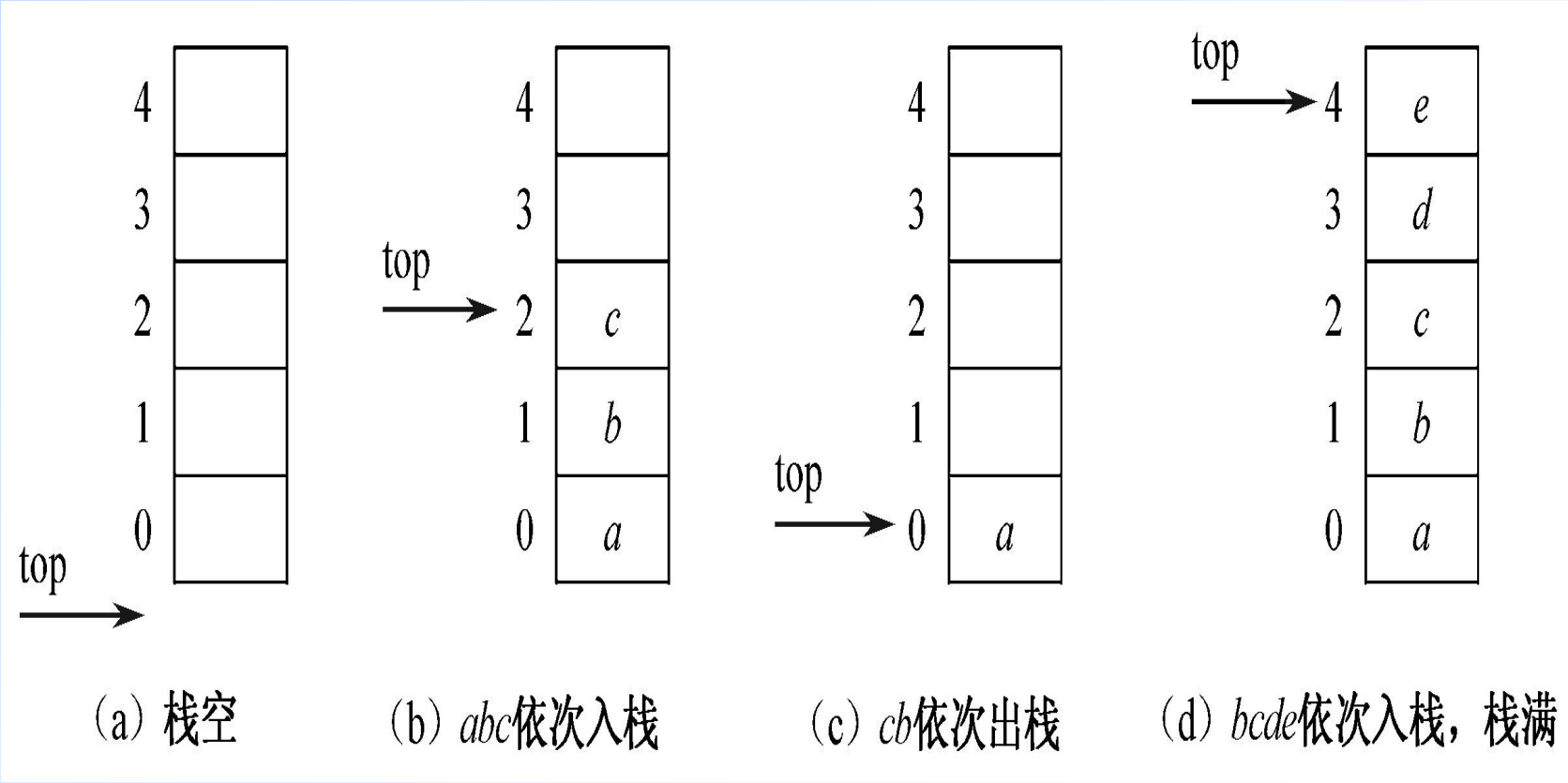
栈的存储结构

- 栈的顺序存储结构（顺序栈）
- 栈的链式存储结构（链栈）

顺序栈

- 栈的顺序存储结构简称为顺序栈。可用数组来实现顺序栈。
- 因为栈底位置是固定不变的，所以可以将栈底位置设置在数组的两端的任何一个端点；
- 栈顶位置是随着进栈和退栈操作而变化的，故需用一个整型变量top表示栈顶位置。

栈操作的示意图



顺序栈的类模板定义：

```
template <class T, int MaxSize >
class SeqStack{
    T data[MaxSize];        //存放栈元素的数组
    int top;                //栈顶指针
  public:
    SeqStack( ) ;           //构造函数
    void Push(T x);         //入栈
    T Pop( );               //出栈
    T Top( );               //取栈顶元素
    bool Empty( );          //判断栈是否为空
};
```

顺序栈的基本操作的实现

1、构造函数

```
template <class T,int MaxSize>
SeqStack<T,MaxSize>::SeqStack( )
{
        top=-1;

}
```

2、入栈操作

```
template <class T,int MaxSize>
void SeqStack<T,MaxSize>::Push( )
{
        if (top== MaxSize-1)
            {cerr<<"上溢";  exit(1);}
        top++;
        data[top]=x;
}
```

3、退栈操作

```
template <class T,int MaxSize>
T SeqStack<T,MaxSize>::Pop( )
{
        if (top==-1)
          { cerr<<"下溢"; exit(1); }
        x=data[top];
        top--;
        return x;
}
```

4、取栈顶元素

```
template <class T,int MaxSize>
T SeqStack<T,MaxSize>::Top( )
{
        if (top==-1)
          { cerr<<"下溢"; exit(1); }
        return data[top];
 }
```

5、判栈空操作

```
template <class T,int MaxSize>
bool SeqStack<T,MaxSize>::Empty( )
{
        return top==-1;
 }
```

6、栈遍历

```
template <class T,int MaxSize>
void SeqStack<T,MaxSize>:: StackTraverse( )
{
        for( i=0; i<=top; i++)
            cout<<data[i];
 }
```

链 栈

- 栈的链式存储结构称为链栈。

- 链栈的设置---

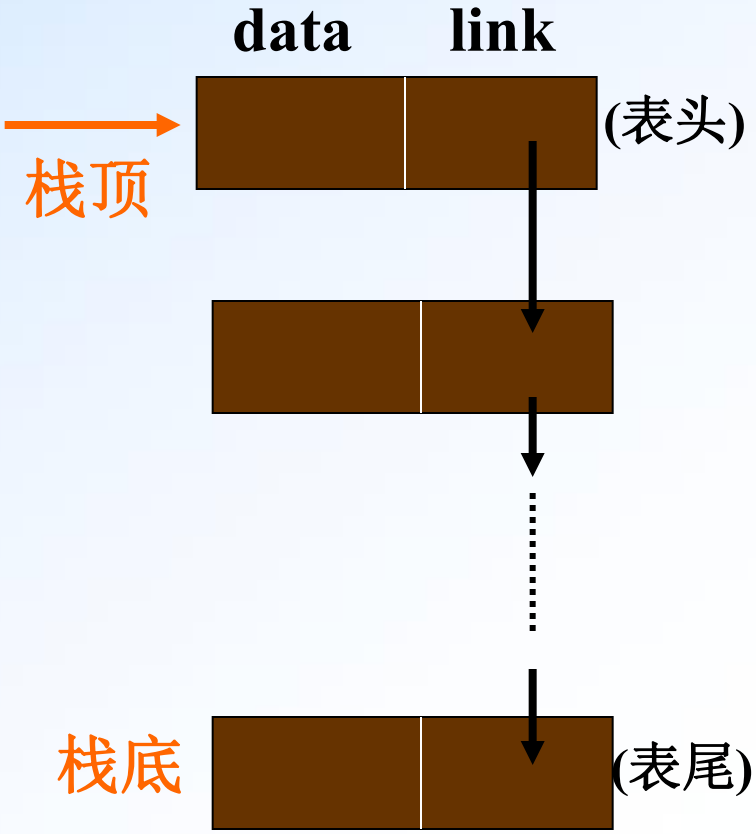
（1）栈顶位置放在哪里合适？

栈顶指向表头位置

进栈/退栈仅在表头进行

（2）是否需要附加头结点？

不需要！



链栈的类模板定义：

```
template <class T, int MaxSize >
class LinkStack{
    Node<T> *top;          //栈顶指针
  public:
    LinkStack( );          //构造函数
    ~LinkStack( );         //析构函数
    void Push(T x);        //入栈
    T Pop( );              //出栈
    T Top( );              //取栈顶元素
    bool Empty( );         //判断链栈是否为空栈
};
```

链栈的基本操作的实现

1、构造函数

```
template <class T,int MaxSize>
LinkStack<T, MaxSize>::LinkStack( )
{
        top=NULL;

}
```

2、入栈操作

```
template <class T,int MaxSize>
void LinkStack<T, MaxSize>::Push(T x )
{
      s=new Node<T>;
      s->data = x;
      s->next = top;
      top=s;
}
```

3、退栈操作

```
template <class T, int MaxSize>
T LinkStack<T, MaxSize>::Pop( )
{
    if (top==NULL) {cerr<<"下溢"; exit(1);}
    x=top->data;
    p=top;
    top=top->next;
    delete p;
    return x;
}
```

4、取栈顶元素

```
template <class T,int MaxSize>
T LinkStack<T, MaxSize>::Top( )
{
        if (top==NULL)
              {cerr<<"下溢"; exit(1);}
        return top->data;
 }
```

5、判栈空操作

```
template <class T,int MaxSize>
bool LinkStack<T, MaxSize>::Empty( )
{
        return top==NULL;
 }
```

6、栈遍历

```
template <class T, int MaxSize>
void LinkStack<T, MaxSize>::StackTraverse( )
{
        for( p=top; p; p=p->next)
            cout<<p->data;
 }
```

栈的应用举例

例1：括号匹配问题：

假设一个算术表达式中允许包含两种括号：圆括号与方括号，其嵌套的次序随意。请设计一个算法判断一个算术表达式中的括号是否匹配。

问题：

1. 括号匹配过程如何组织？

2. 在括号匹配过程中如何应用栈？

```
void check() {
   SeqStack<int,100> S;
   char ch[80], *p, e;
   printf("请输入表达式\n");
   gets(ch);
   p = ch;
   while( *p ) // 没到串尾
       switch(*p) {
         case '(':
         case '[':
                  S.Push(*p++);
                  break;      // 左括号入栈，且p++
```

```
case ')':
case ']': if( !S.Empty() ) {
                e=S.Pop();                  // 弹出栈顶元素
                if(*p==')' && e!='(' || *p==']' && e!='['){
                    printf("左右括号不配对\n");
                    exit(1);
                }
                else{
                   p++;
                   break;    // 跳出switch语句
                }
            }   // Empty
            else {   // 栈空
                 printf("缺乏左括号\n");
                 exit(1);
            }
```

```
        default: p++;      // 其它字符不处理，指针向后移
    }
    if(S.Empty() )  // 字符串结束时栈空
        printf("括号匹配\n");
    else
        printf("缺乏右括号\n");
}
```

- 例2： 表达式求值问题：

  输入包含+、－、*、/、圆括号和正整数组成的中缀算术表达式，以“@”作为表达式结束符。设计一个算法，对给定的中缀算术表达式进行求值。

  2 + 4 * 6 – 8 / (5 - 3) + 7 @

- 假定采用“算符优先法”对中缀表达式进行求值。
  - 一个表达式是由操作数、运算符、界限符组成。
  - 任何两个相继出现的算符之间的优先级关系为> 、= 、< 。
  - 算符之间的优先级关系完全决定了操作数的运算次序。

算符间的优先关系

θ_1 \ θ_2	+	−	×	/	(	)	@
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
×	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
@	<	<	<	<	<		=

说明：

- 左括号‘(’的优先级最高
- 当θ1 = θ2 时，令

  θ1 > θ2（左结合性）
- 为了算法处理的方便，在表达式的最左边添加一个‘@’。
- 优先关系相等的情况只有‘(’=‘)’和‘@’=‘@’

•算术表达式求值示例：　　8/(5-3)@

- 基于算符优先法，如何组织和实现表达式求值过程？
- 如何利用栈的后进先出特性？

- **数据结构的设计**
  - 为实现算符优先算法，需引入两个栈：
    - **OPTR**：用于保存运算符
    - **OPND**，用于保存操作数

表达式求值算法的设计：

1、首先将操作数栈OPND置为空栈，将表达式起始符“@”压入运算符栈OPTR中作为栈底元素。

2、依次读入表达式的每个字符，直至当前字符与运算符栈的栈顶元素均为“@”时，结束下列循环：

- 若是操作数，则进操作数栈OPND ；
- 若是运算符,则进行以下判断：
  - 若当前运算符高于OPTR栈顶运算符，将当前运算符入栈OPTR ，继续读入下一字符；
  - 若当前运算符低于OPTR栈顶运算符，栈顶运算符退栈，并弹出操作数栈的两个栈顶操作数进行运算，再将运算结果压入操作数栈OPND中。
  - 若当前运算符等于栈顶运算符，表明栈顶运算符为“（”，当前运算符为“）”，则栈顶运算符退栈。

算术表达式求值过程示例：　**8 / (5 - 3) @**

```
int EvaluateExpression() {
    SeqStack<char,100> OPTR;
    SeqStack<int,100> OPND;
    OPTR.Push('@');
    c = getchar();
    while( c!=' @ ' || OPTR.Top()!='@ ' ){
        if( InOPTR(c, OP) )          // c是运算符
            processing…
        else if(c>='0'&&c<='9') {   // c是操作数
            OPND.Push(c);  c=getchar();
        }
        else {    // c是非法字符
            printf("ERROR\n");  exit(1);
        }
    }
    return OPND.Top();
}
```

```
x= OPTR.Top();
switch( Precede(x, c) ){
      case‘ < ‘ :      // 栈顶元素优先权低
          OPTR.Push(c);  c=getchar();
          break;
     case‘ = ‘ :      //脱括号并接收下一字符
          x=OPTR.Pop();  c=getchar();
          break;
    case ‘ > ‘ :     // 退栈并将运算结果入栈
          theta=OPTR.Pop();
          b=OPND.Pop();
          a=OPND.Pop();
          OPND.Push(Operate(a,theta,b));
          break;
}
```