第四章 数据结构

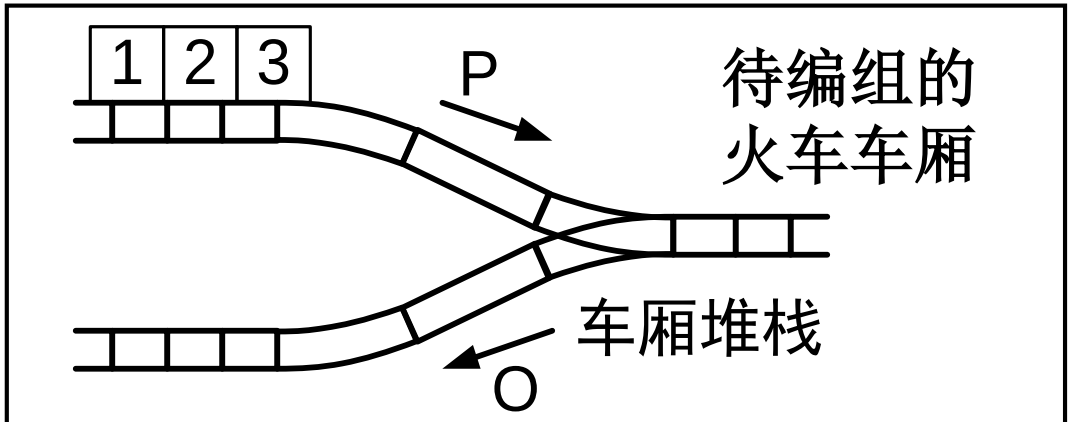
- 4.1 基本概念
- 4.2 线性表
- 4.3 栈和队列
  - 4.3.1 栈
  - 4.3.2 顺序栈
  - 4.3.3 链接栈
  - 4.3.4 队列
  - 4.3.5 顺序队列
  - 4.3.6 环形队列
  - 4.3.7 链接队列
- 4.4 树
- 4.5 二叉树
- 4.6 二叉树的线性表示及生成
- 4.7 任意次树与二叉树之间的转换

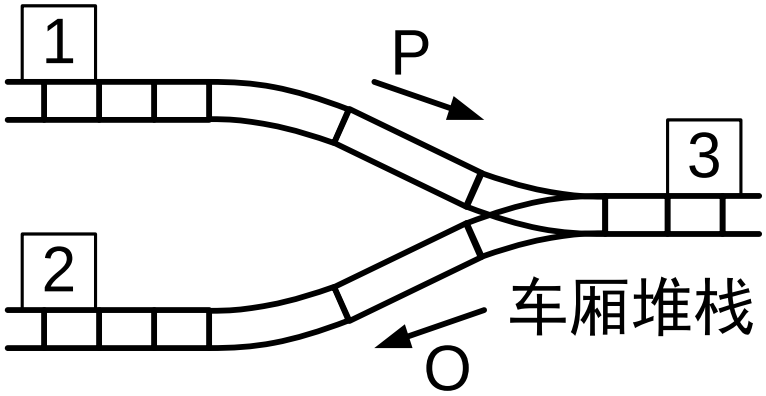
【例4-3.1】栈的应用示例：铁路车厢编组

下图中铁路的右侧是一个车厢的堆栈，规定从上边轨道进入堆栈的操作为P(push)，从堆栈退出进入下边轨道的操作为O(pop)。通过一系列的进栈和出栈操作，可以对火车车厢进行重新编组。

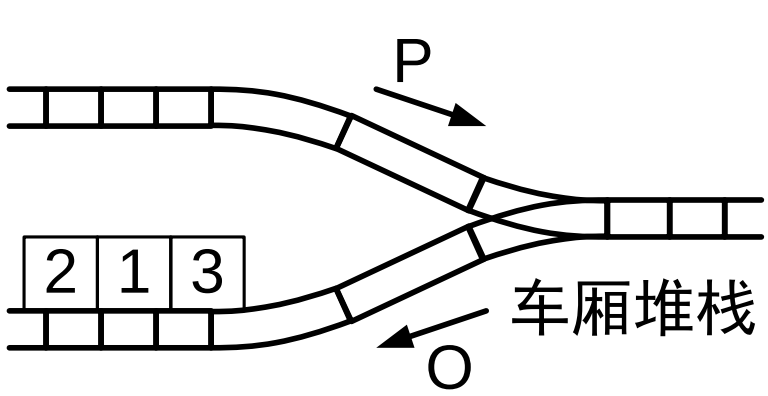
例如经过PPO的操作，原来按{1, 2, 3}编组的车厢被牵引到各条轨道。

而经过PPOPOO的操作，原来按{1, 2, 3}编组的车厢可以重新编组为{2, 1, 3}的序列。





经过PPO操作的车厢编组



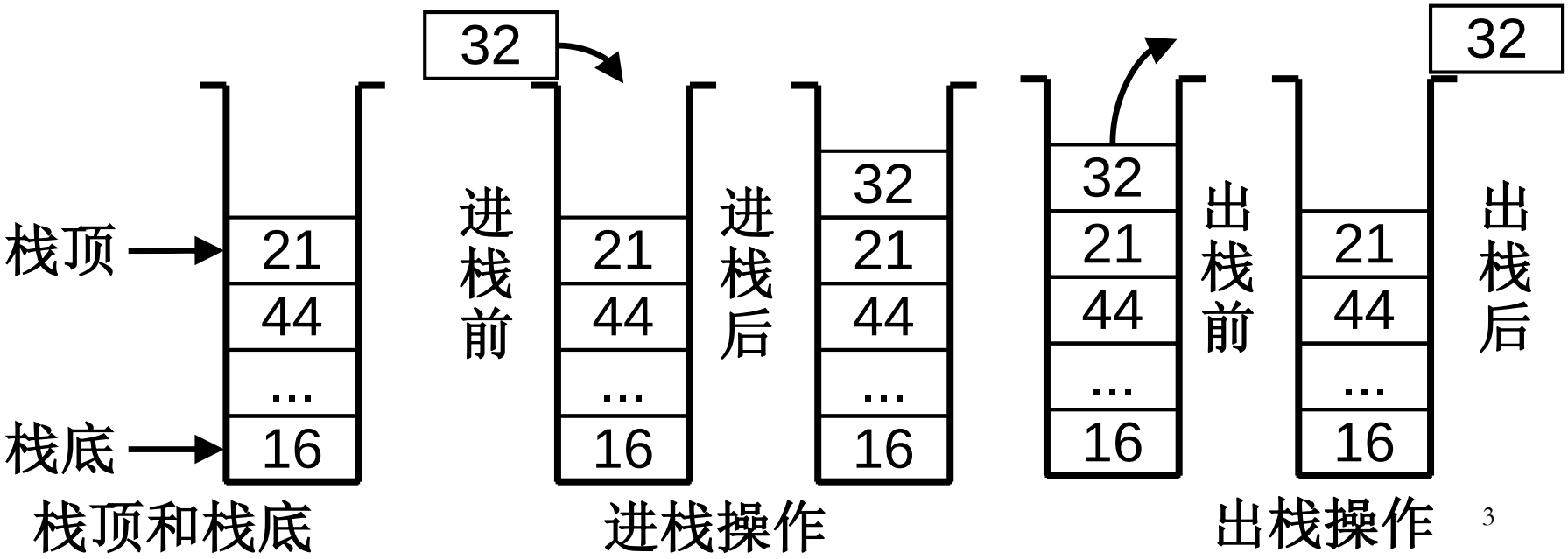
编组后的火车车厢

➢4.3 栈和队列

➢ 4.3.1 栈

➢ 定义

只能对始结点进行操作的线性表称为stack(栈，堆栈)。
栈的始端称为栈顶，栈的终端称为栈底。



➢4.3 栈和队列

➢ 4.3.1 栈

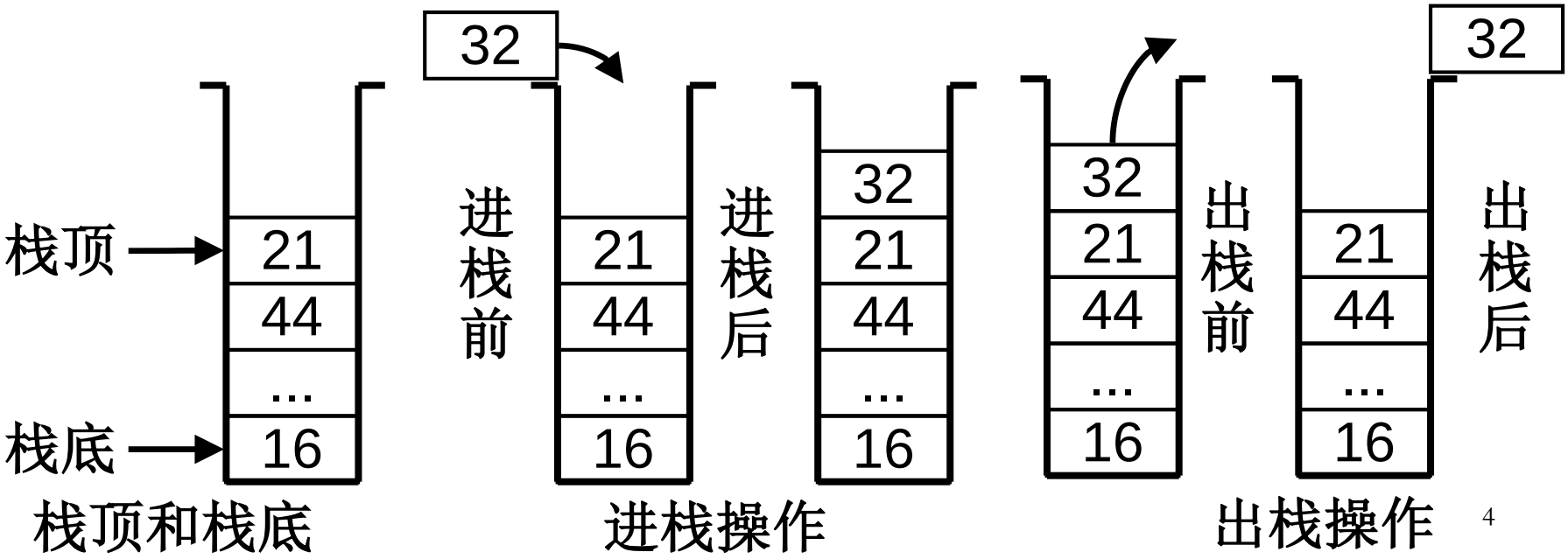
➢ 定义

栈的主要操作包括**push(进栈)**和**pop(出栈或者退栈)**。

进栈是指在栈顶添加一个结点，使原来的栈顶成为下一结点。

出栈是从栈内取出栈顶，并使原来的下一结点成为栈顶。

最先进栈的结点将最后出栈。因此，栈又被称为先进后出表，记为**FILO表(First In Last Out List)**。



➢栈的存储方式

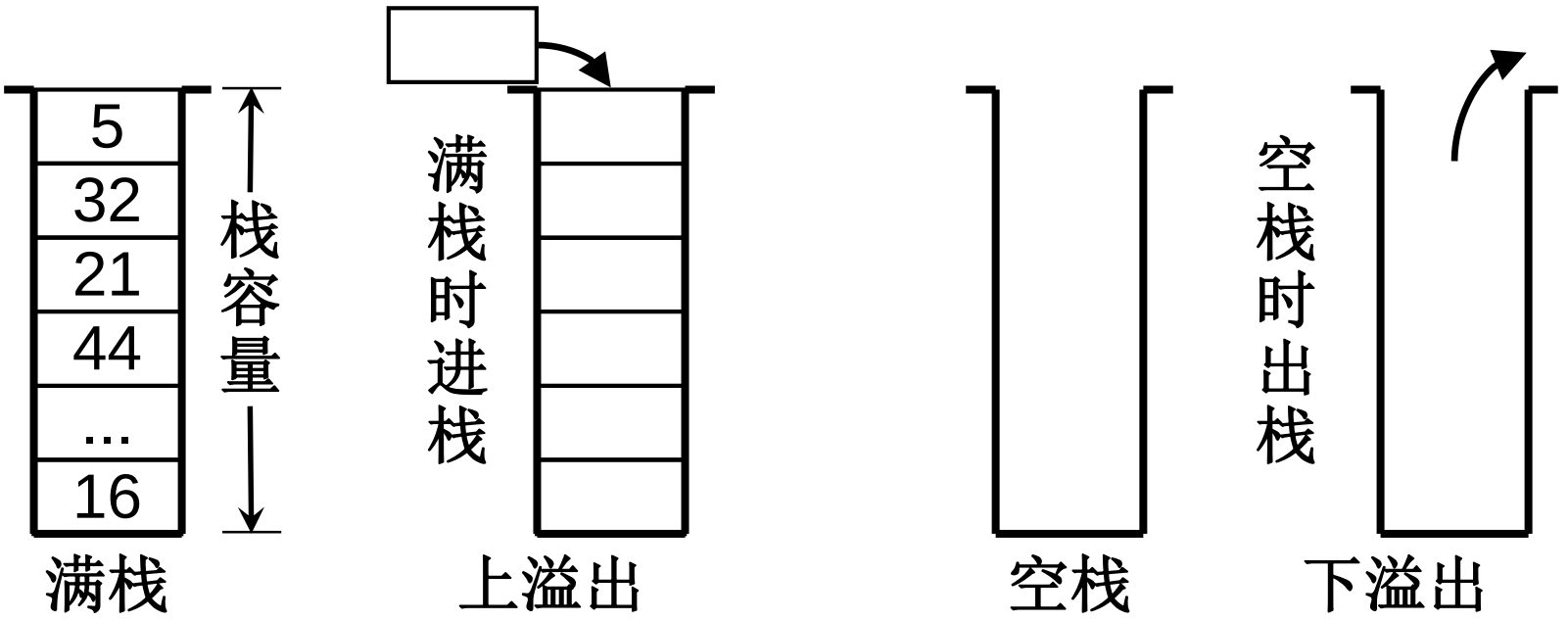
用顺序存储实现的栈称为顺序栈，通常用数组来构造顺序栈。

用链接存储实现的栈称为链接栈，通常用链表来构造链接栈。

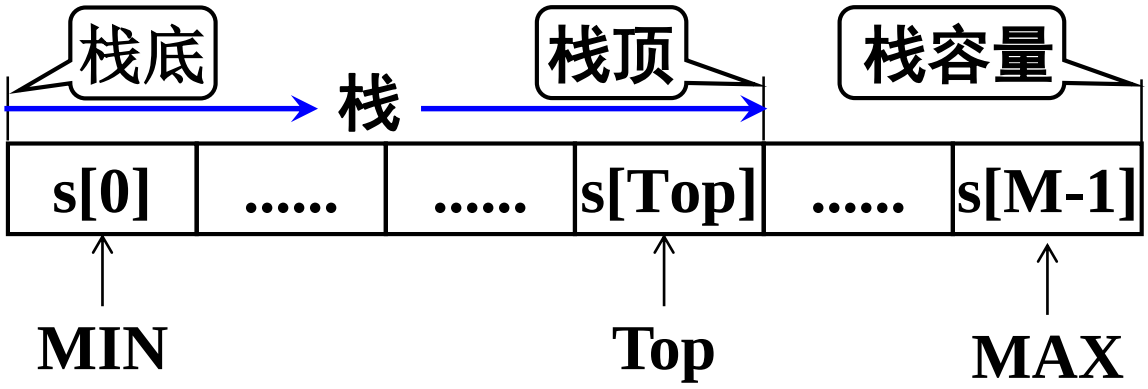
➢栈的基本操作和存储空间测试

设计一个栈，要给予分配一定的空间，因此在进栈或退栈操作时应注意到栈空间的变化情况。如果在满栈(结点已经占满栈的容量)时执行进栈操作，称为上溢出(overflow)。而如果在空栈(栈内没有结点)时执行出栈操作，称为下溢出(underflow)。

通常根据栈顶指针的变化来判断是否发生上溢出和下溢出。在程序实现时必须对上溢出和下溢出的情况加以处理，否则将引起出错。



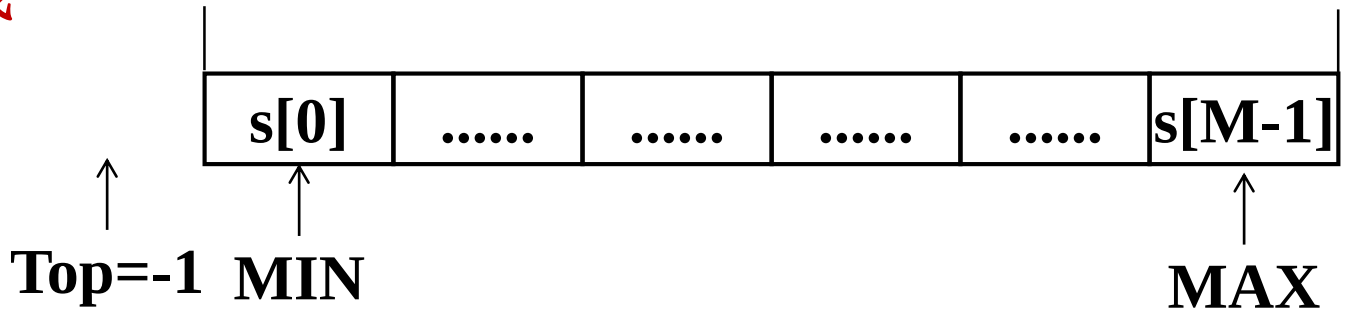
➢4.3.2 顺序栈



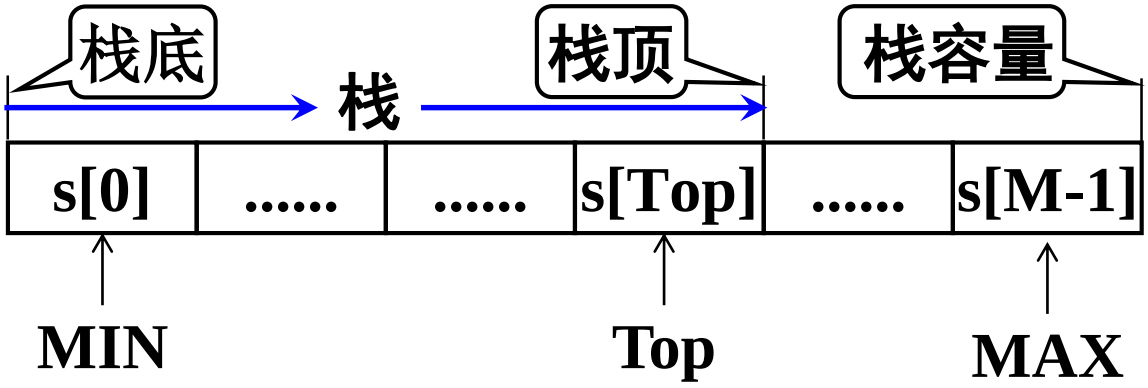
➢数据定义

```
#define   M      1000
#define   MAX    M-1      /* 栈容量                          */
#define   MIN    0        /* 栈底指针值                      */
short  Stack[M];          /* 用数组存放栈                    */
short  Top = MIN-1;       /* 栈顶指针，初始为空栈 */
```

空栈



➢4.3.2 顺序栈



➢ 进栈函数

```
void push(short key) /* 结点值*/
{
  if(Top>=MAX  /* overflow    */
  || Top<MIN-1) /* 非法指针值*/
       error();
  /* 结点进栈，指针加1 */
  Stack[++Top] = key;
}
```

➢ 出栈函数

```
short pop(void)
{
    if(Top<MIN   /* underflow   */
    || Top>MAX) /* 非法指针值*/
         error();
    /* 返回栈顶结点, 指针减1 */
    return(Stack[Top--]);
}
```

➢ 调用方法示例

```
push(key);
```

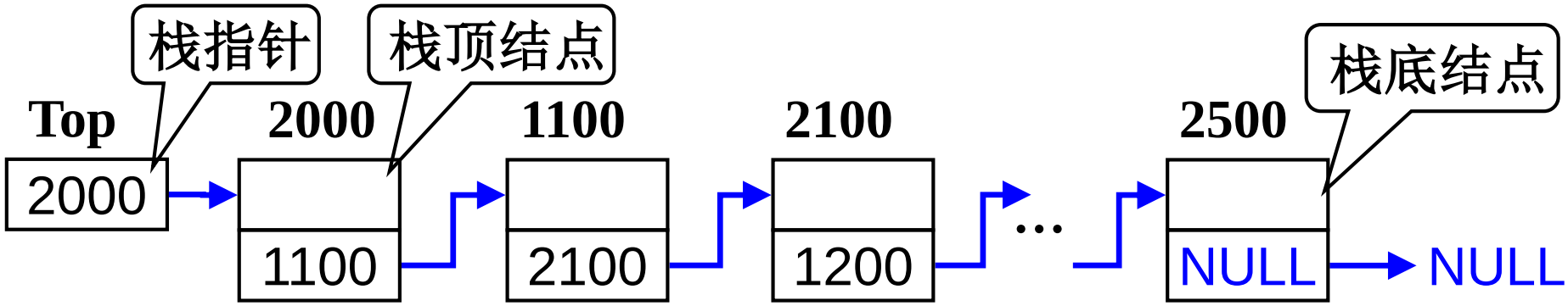
➢ 调用方法示例

```
key = pop( );
```

➢4.3.3 链接栈

➢数据定义

```
#define  NODE  struct node
NODE
{
   short  num;
   NODE   *next;
};      /* 用链表存放栈 */
NODE *Top=NULL;   /* 栈指针，初始为空栈 */
```



➢进栈函数

```
void push(short key)
{
   NODE   *node;
   node=(NODE *)
      malloc(sizeof(NODE));
   if(node == NULL)
      error();  /* 满栈(overflow) */
   /* 填入结点数据          */
   node->num = key;
   /* 插入结点              */
   node->next = Top;
   Top=node;
}
```

➢调用方法示例

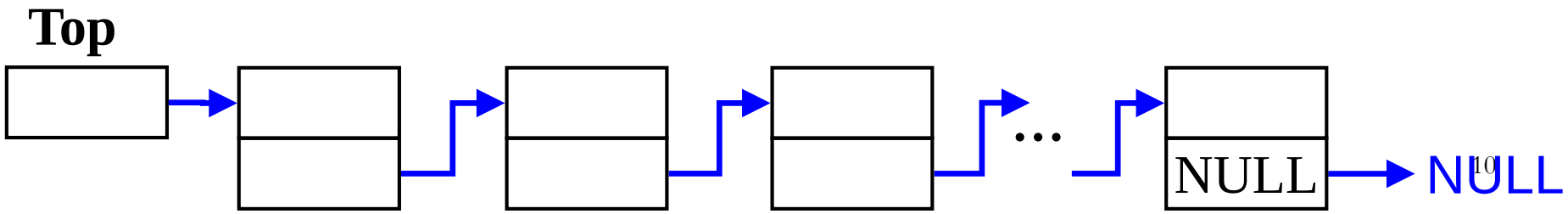
```
push(key);
```

➢出栈函数

```
short pop(void)
{
   short  key;
   NODE  *node;
   if(Top==NULL)
      error(); /* 空栈(underflow) */
   key = Top->number;
   /* 删除结点          */
   node = Top;
   Top = node->next;
   free(node);    /* 释放结点*/
   return(key);
}
```

➢调用方法示例

```
key = pop( );
```



➢栈的应用

- 函数递归调用（汉诺塔）

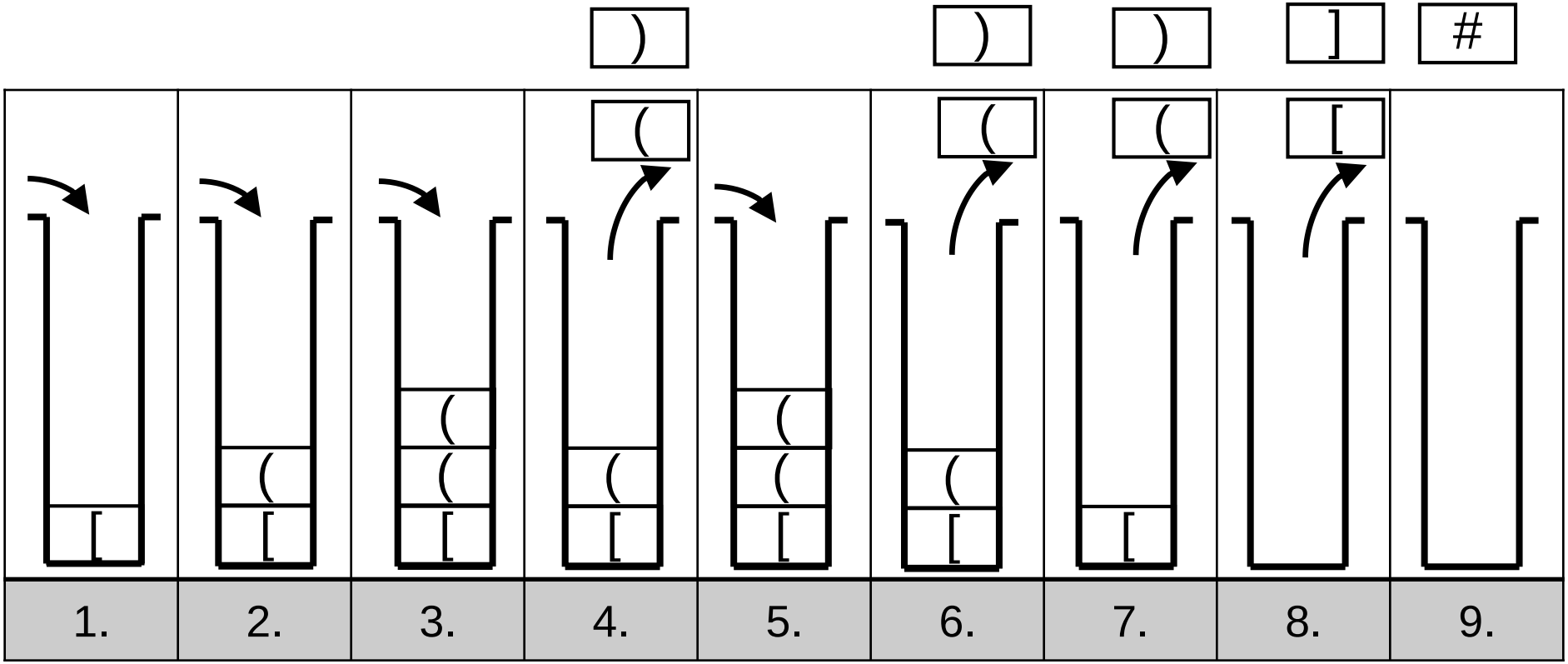
①调用函数时：系统将会为调用者构造一个由参数表和返回地址组成的活动记录，并将其压入到由系统提供的运行时刻栈的栈顶，然后将程序的控制权转移到被调函数。

②被调函数执行完毕时：系统将运行时刻栈栈顶的活动结构退栈，并根据退栈的活动结构中所保存的返回地址将程序的控制权转移给调用者继续执行。

➢栈的应用

- 括号匹配

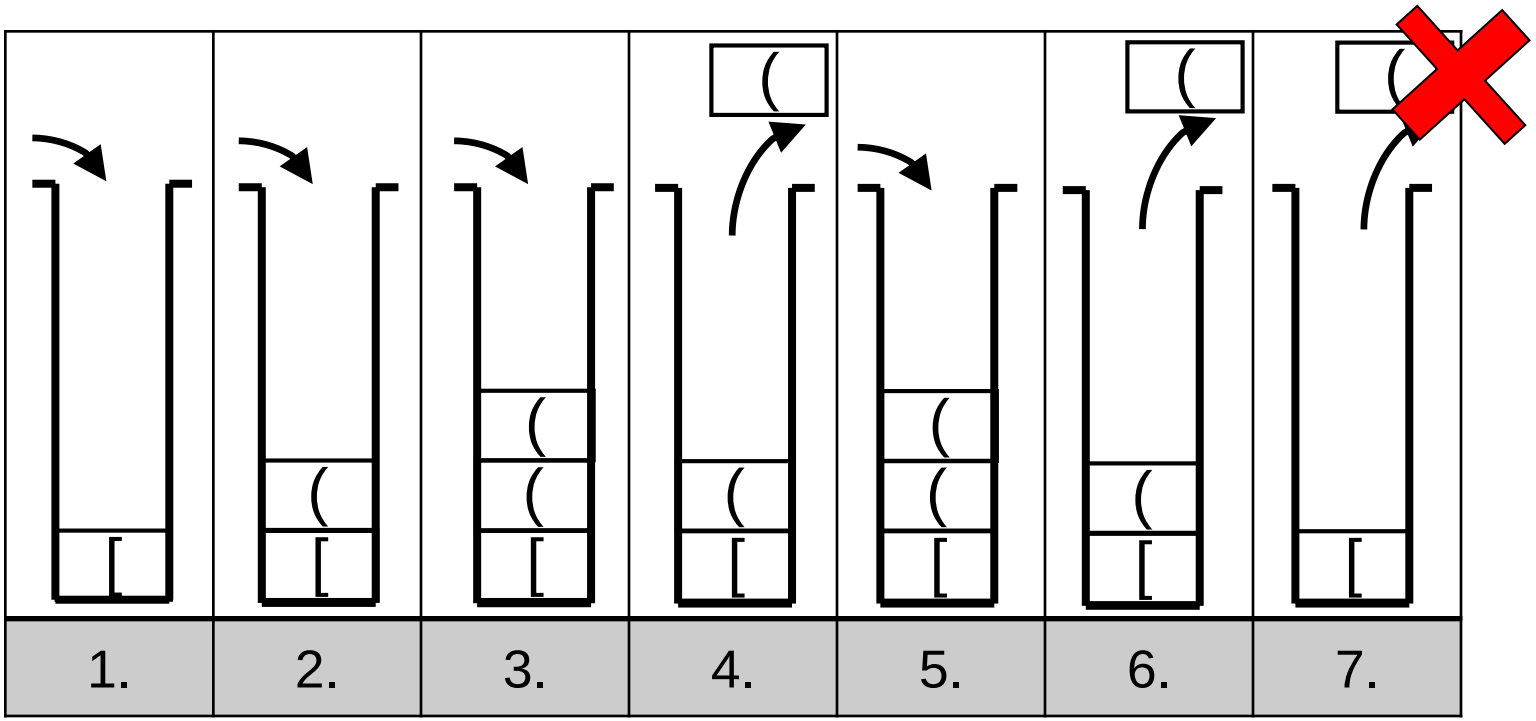
1.	2.			3.				4.		5.				6.	7.					8.			9.
[	(	a	+	(	b	+	c	)	*	(	d	+	e	)	)	*	d	+	e	]	/	h	#



➢栈的应用

- 括号匹配

1.	2.			3.				4.		5.				6.						7.			
[	(	a	+	(	b	+	c	)	*	(	d	+	e	)		*	d	+	e	]	/	h	#



【例4-3.2】栈的应用示例：PFE的计算

- 算法描述

PFE(无括号表达式，又称后缀表达式或者逆波兰表达式)的计算，是栈的一种应用。

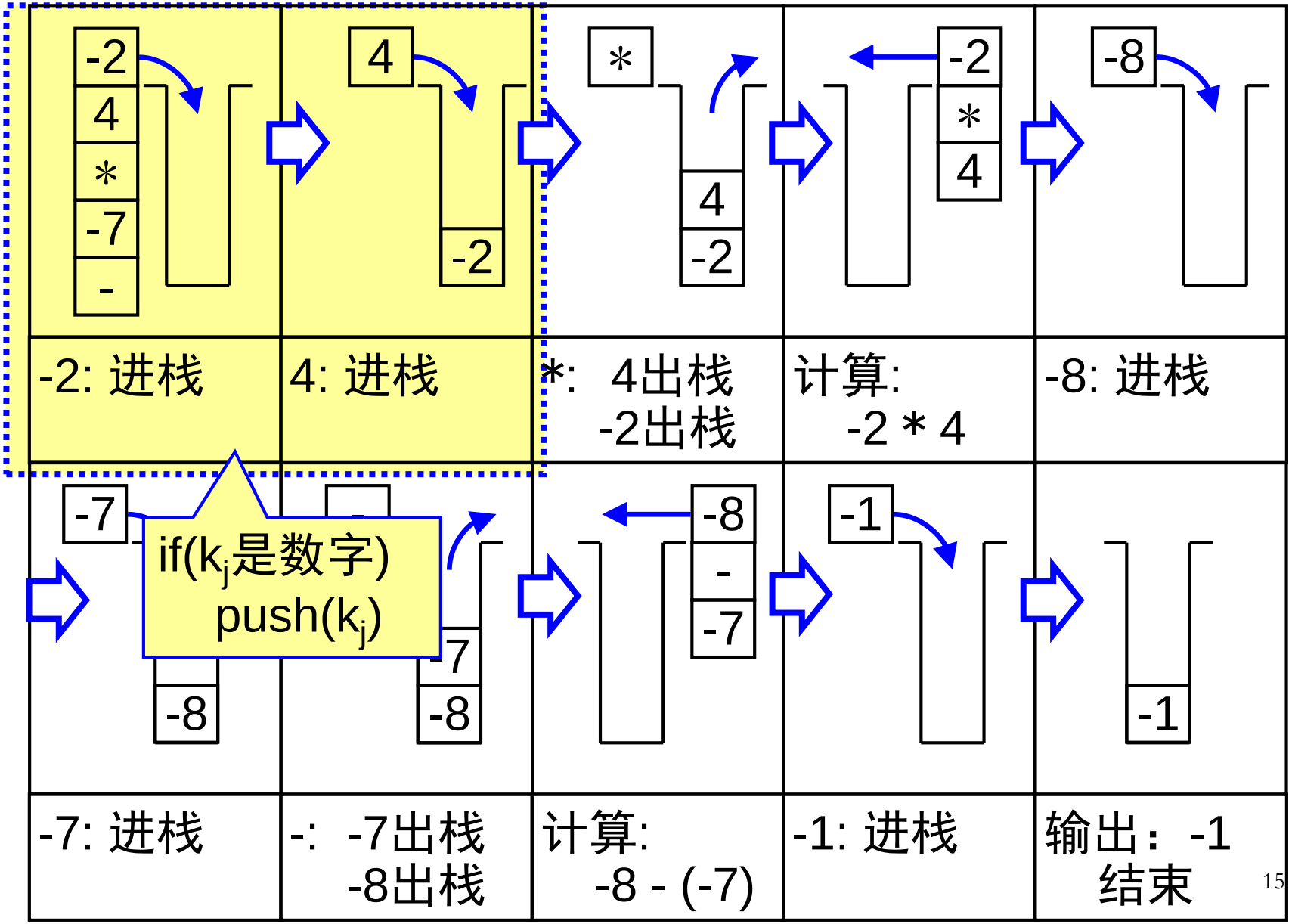
即K={k_j, (j=1, …, n | k_j=数字或算术运算符+, -, *, /)}。以此为程序的输入序列，求PFE的计算结果。

求解PFE计算结果的算法为：

```
for(j=1; j<=n; j++)
{
    if(kj是数字)
        push(kj)
    else if(kj是算术运算符op)
    {
        data = pop()
        result = pop()
        result = result op data
        push(result)
    }
}
输出result
```

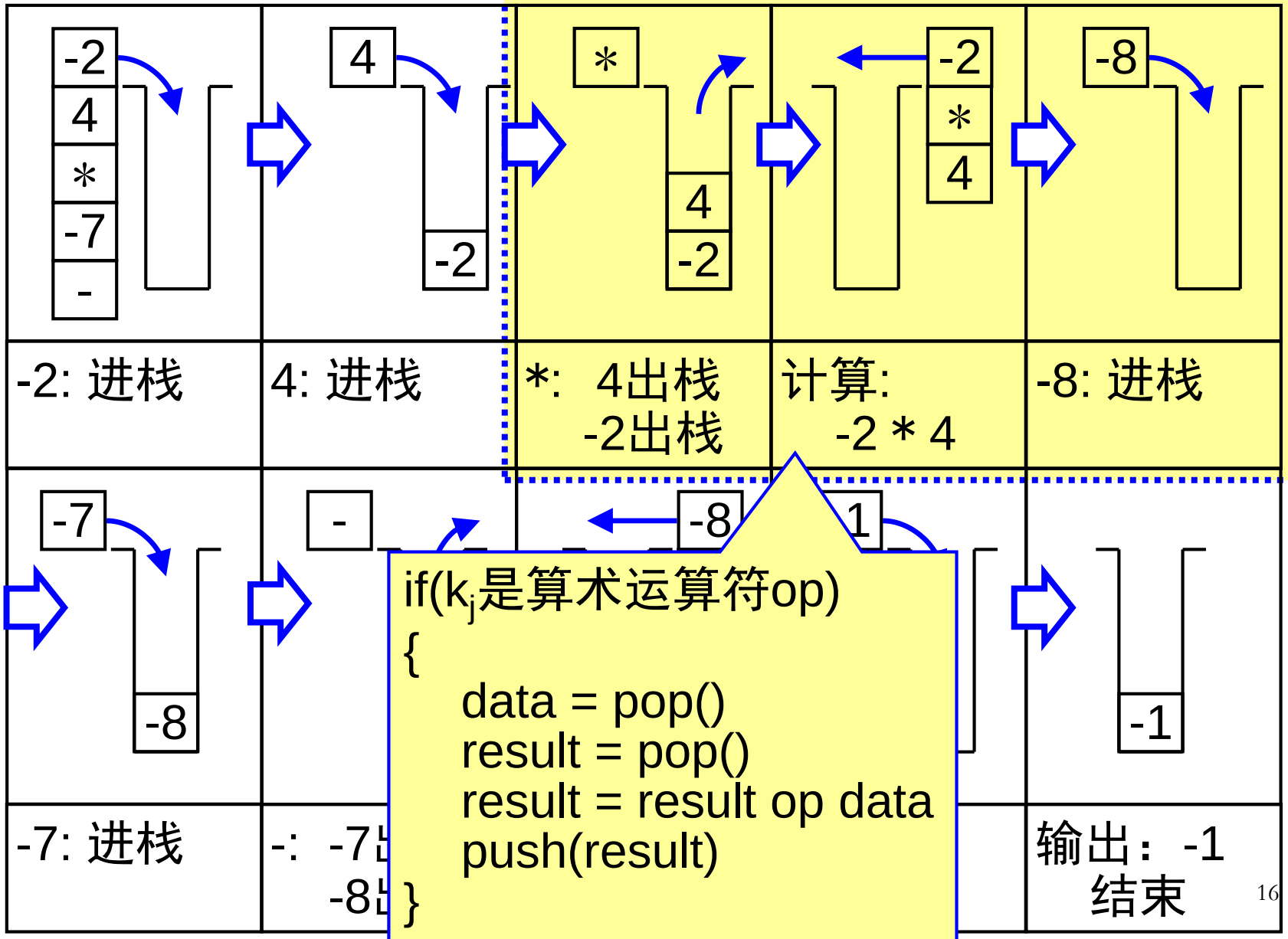
● 求解过程

以K={-2, 4, *, -7, -}为例，用图示法来演示PFE的求解过程。



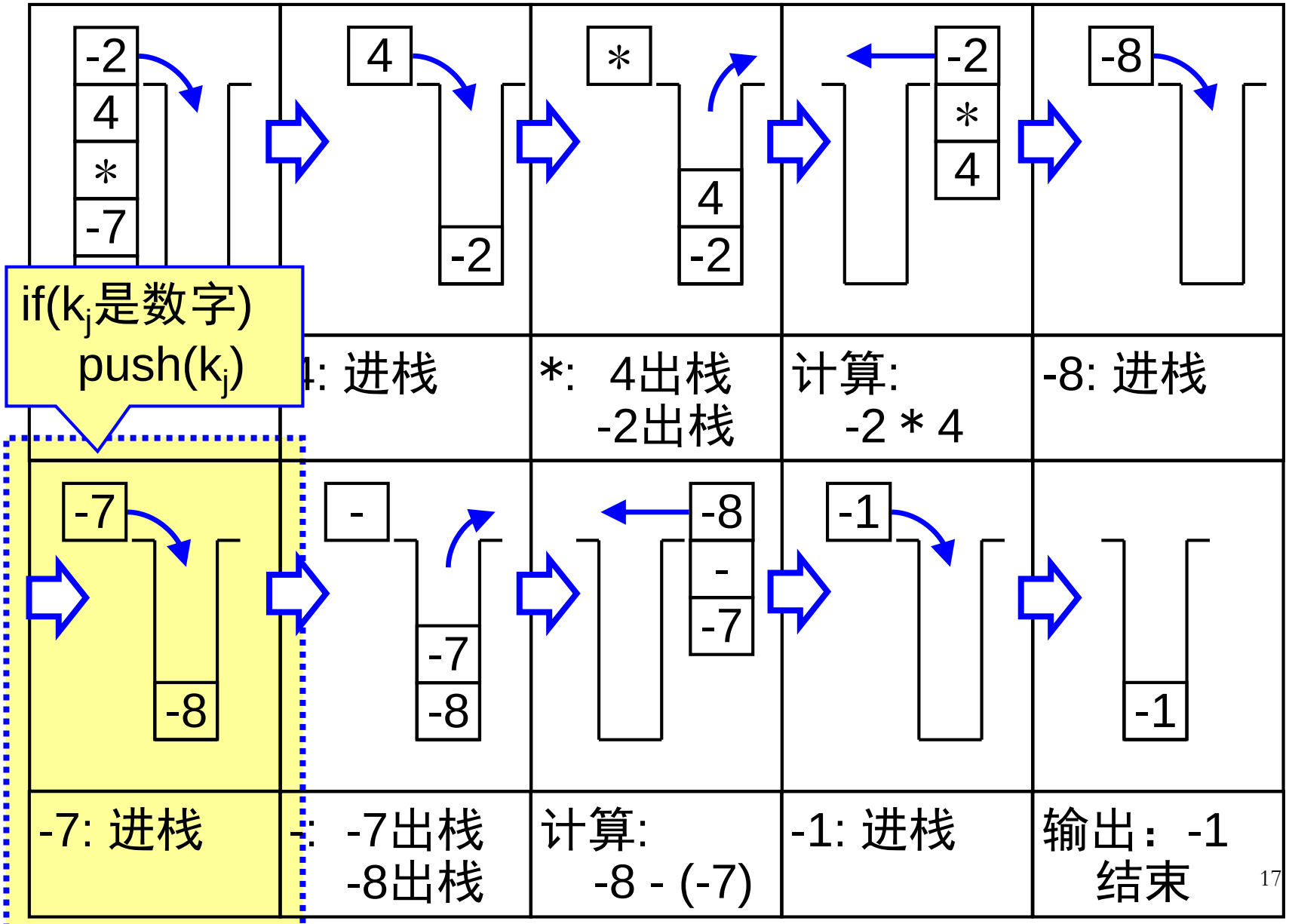
● 求解过程

以K={-2, 4, *, -7, -}为例，用图示法来演示PFE的求解过程。



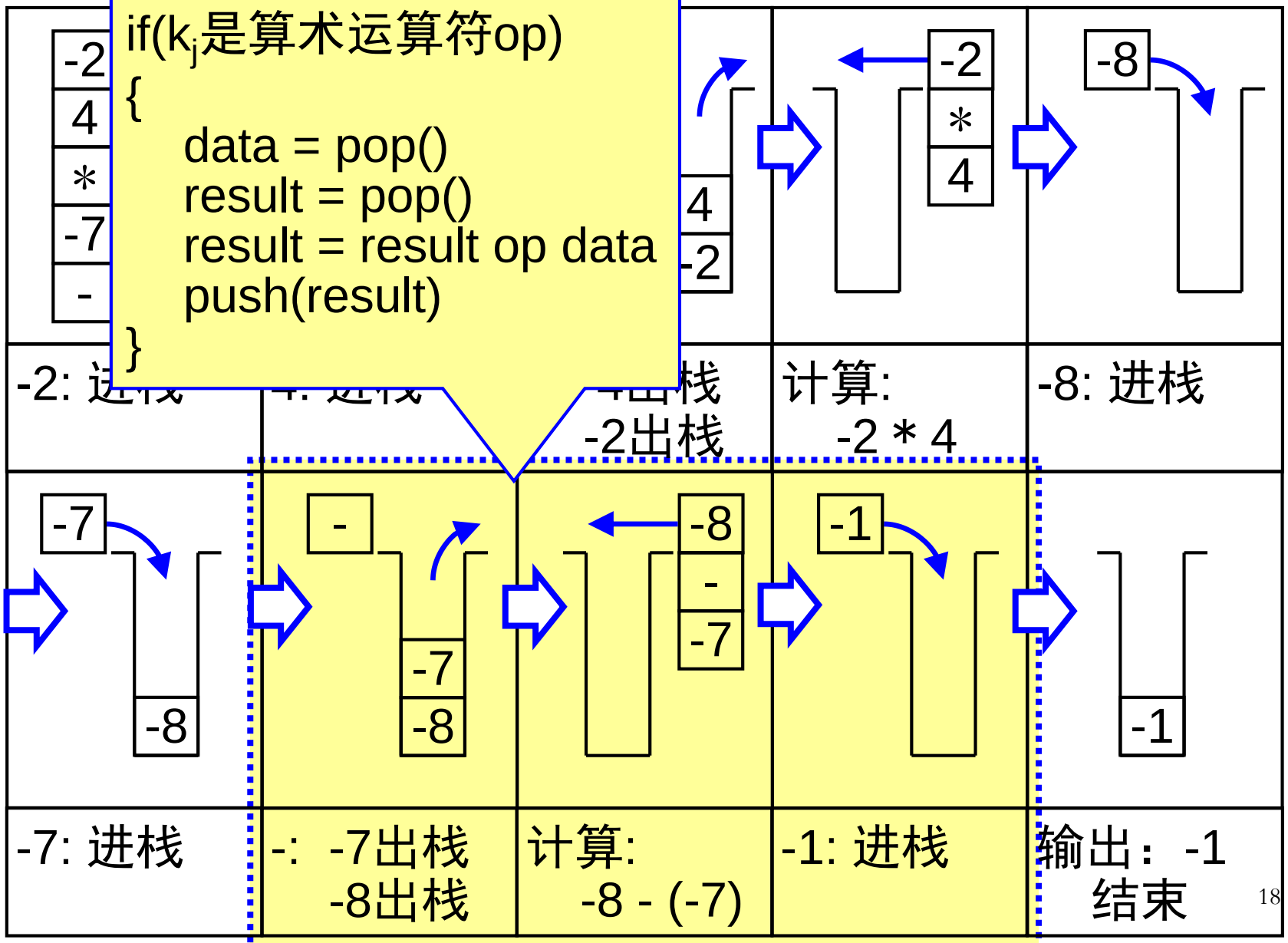
● 求解过程

以K={-2, 4, *, -7, -}为例，用图示法来演示PFE的求解过程。



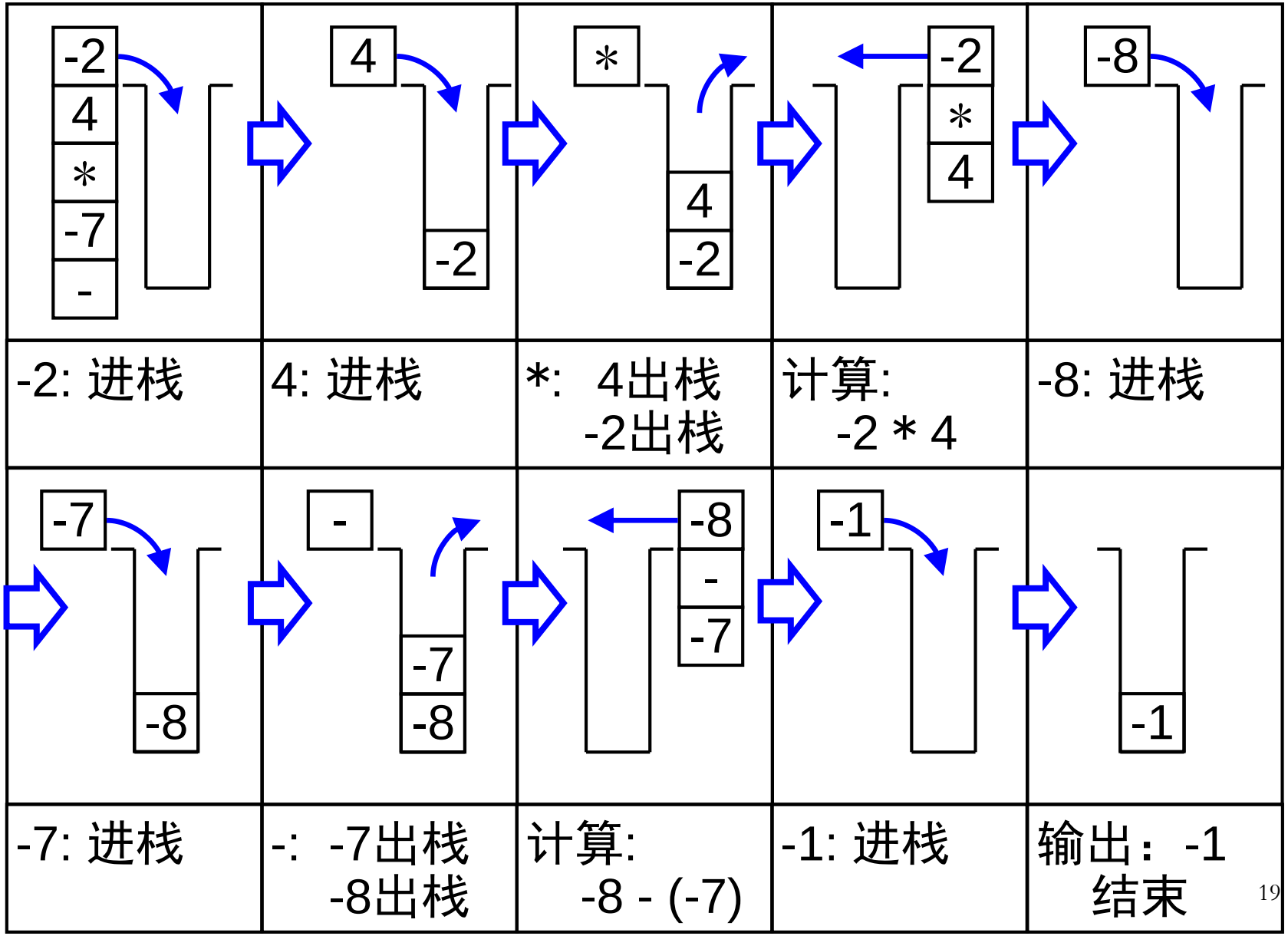
●求解过程

以K={-2, 4, *, -7, -}为例，用图示法来演示PFE的求解过程。



● 求解过程

以K={-2, 4, *, -7, -}为例，用图示法来演示PFE的求解过程。



➢空栈管理

在链接栈的进栈函数push()中需要调用malloc()申请空间，出栈函数pop()中需要调用free()释放空间。

反复调用malloc()和free()，实际上是增加了与系统打交道的时间。如果次数频繁，将降低程序的运行速度。

通常在实用的软件中可以采用空栈管理的办法。采用空栈管理的做法是集中申请一批存储单元，放在空栈内。需要使用结点(进栈)时从空栈取出结点，用完(出栈)后将结点放回空栈，从而提高软件的效率。

➢空栈管理

分别设计空栈管理的出栈函数**popfree()**和进栈函数**pushfree()**，其程序请参见教材第**252**页。

从而可以修改链接栈的进栈函数和出栈函数中的相关语句如下：

- 修改进栈函数申请空间的语句

  if((node=(NODE *)malloc(sizeof(NODE))) == NULL)

  改为：

  if((node= popfree()) == NULL)

- 修改出栈函数释放空间的语句

  free(node);

  改为：

  pushfree(node);

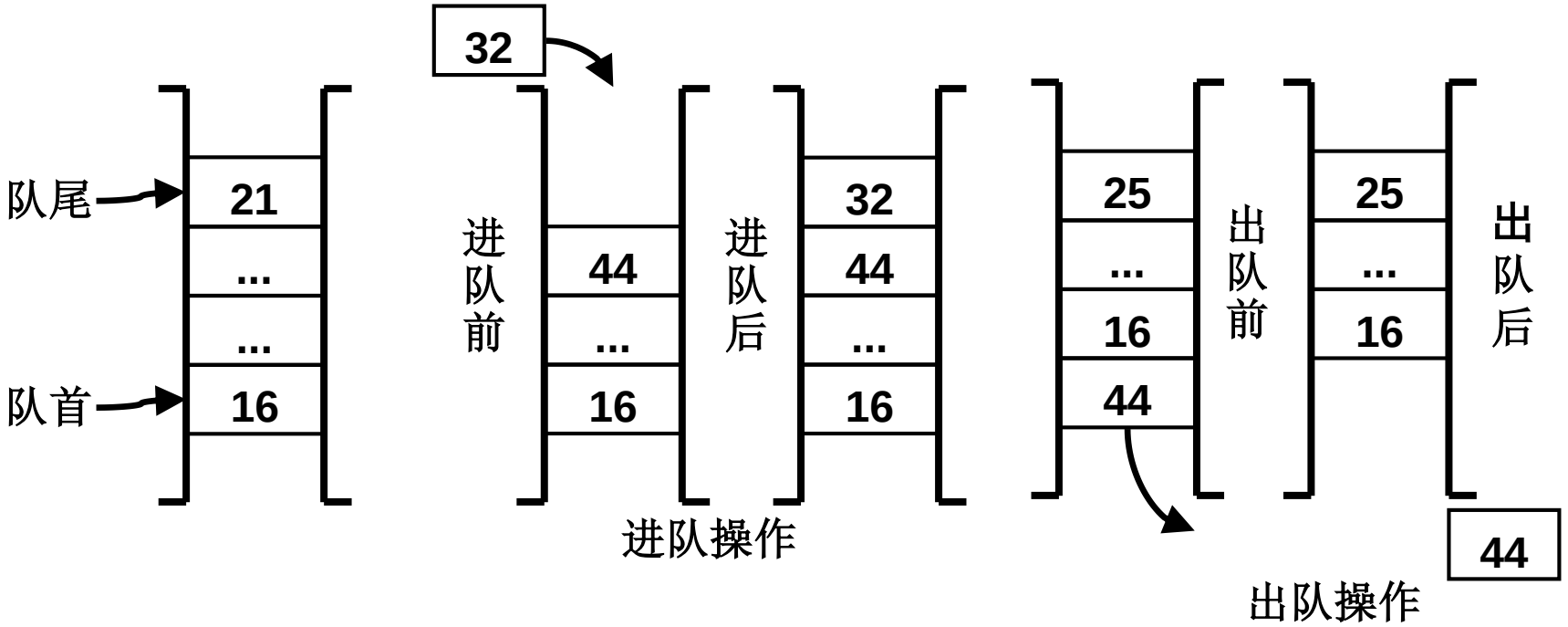
第四章 数据结构

- 4.1 基本概念
- 4.2 线性表
- 4.3 栈和队列
  - 4.3.1 栈
  - 4.3.2 顺序栈
  - 4.3.3 链接栈
  - 4.3.4 队列
  - 4.3.5 顺序队列
  - 4.3.6 环形队列
  - 4.3.7 链接队列
- 4.4 树
- 4.5 二叉树
- 4.6 二叉树的线性表示及生成
- 4.7 任意次树与二叉树之间的转换

➢4.3.4 队

➢ 定义

- 只能添加终结点和删除始结点的线性表称为queue(队列，队)。
- 队的终端称为队尾，队的始端称为队首。

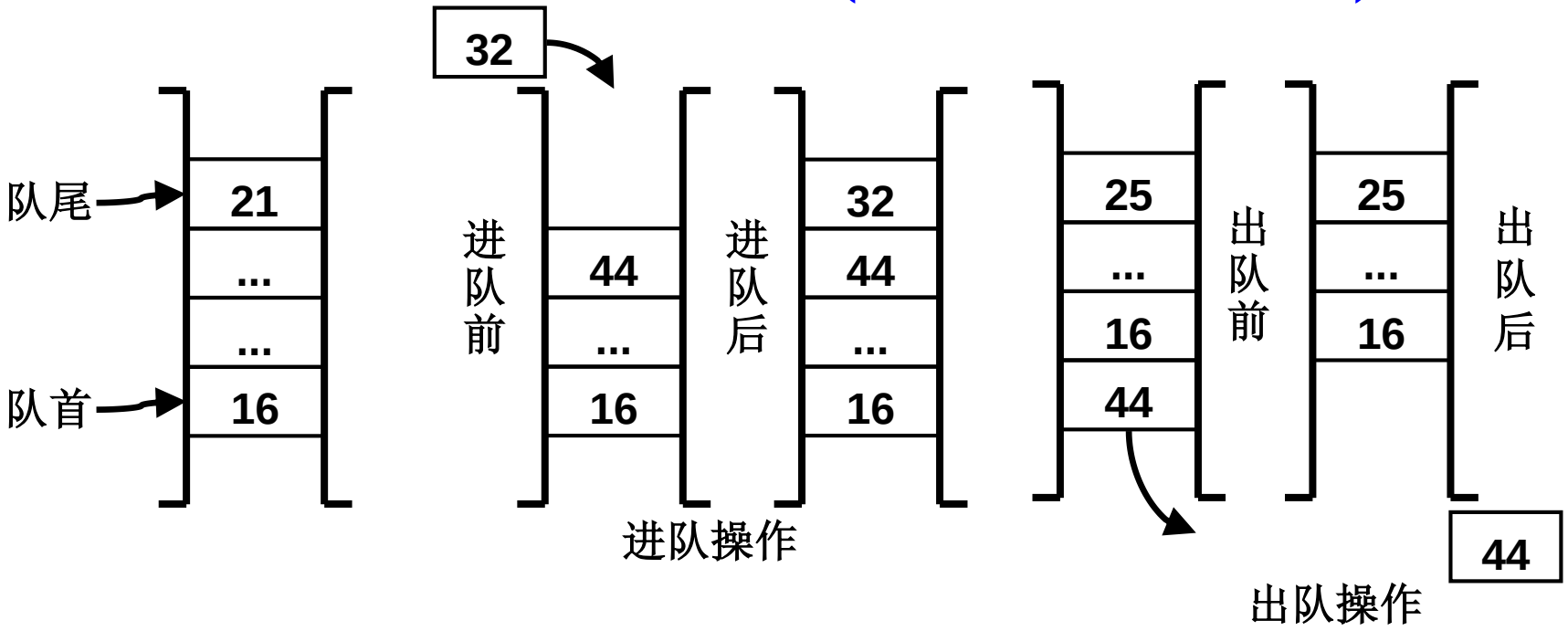


➢4.3.4 队

➢ 定义

- 队的主要操作包括**enqueue(进队)**和**dequeue(出队)**。
- **进队操作**是在队尾添加一个结点，并使新结点成为队尾结点。
- **出队操作**是在队首取出一个结点，并使原来的下一个结点成为队首结点。

  由此可以看到，最先进队的结点必定最先出队。因此，队又被称为先进先出表，记为**FIFO表(Fist In Fist Out List)**。

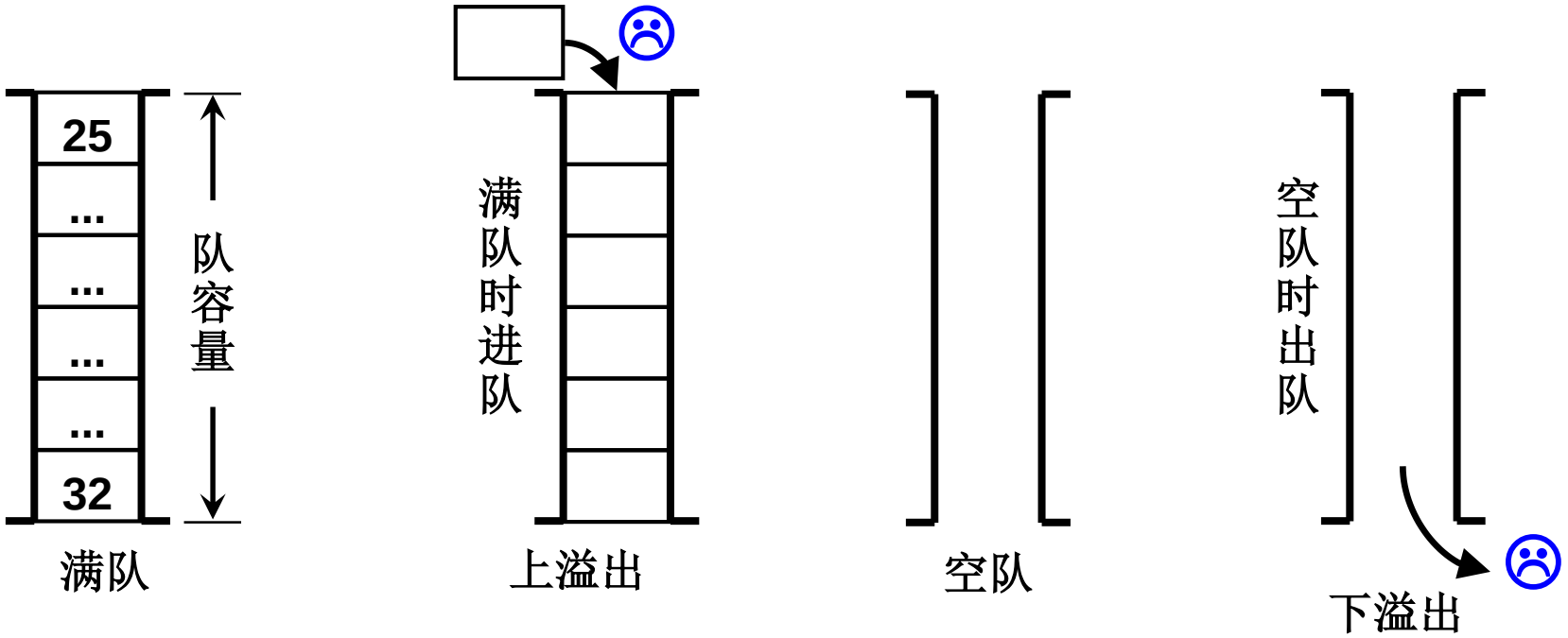


➢4.3.4 队

➢队的基本操作和存储空间测试

类似于栈的情况，设计一个队列，要给予分配一定的空间，因此在进队或出队操作时应注意到队列空间的变化情况。

如果在满队(结点已经占满队列的容量)时执行进队操作，称为上溢出(overflow)。



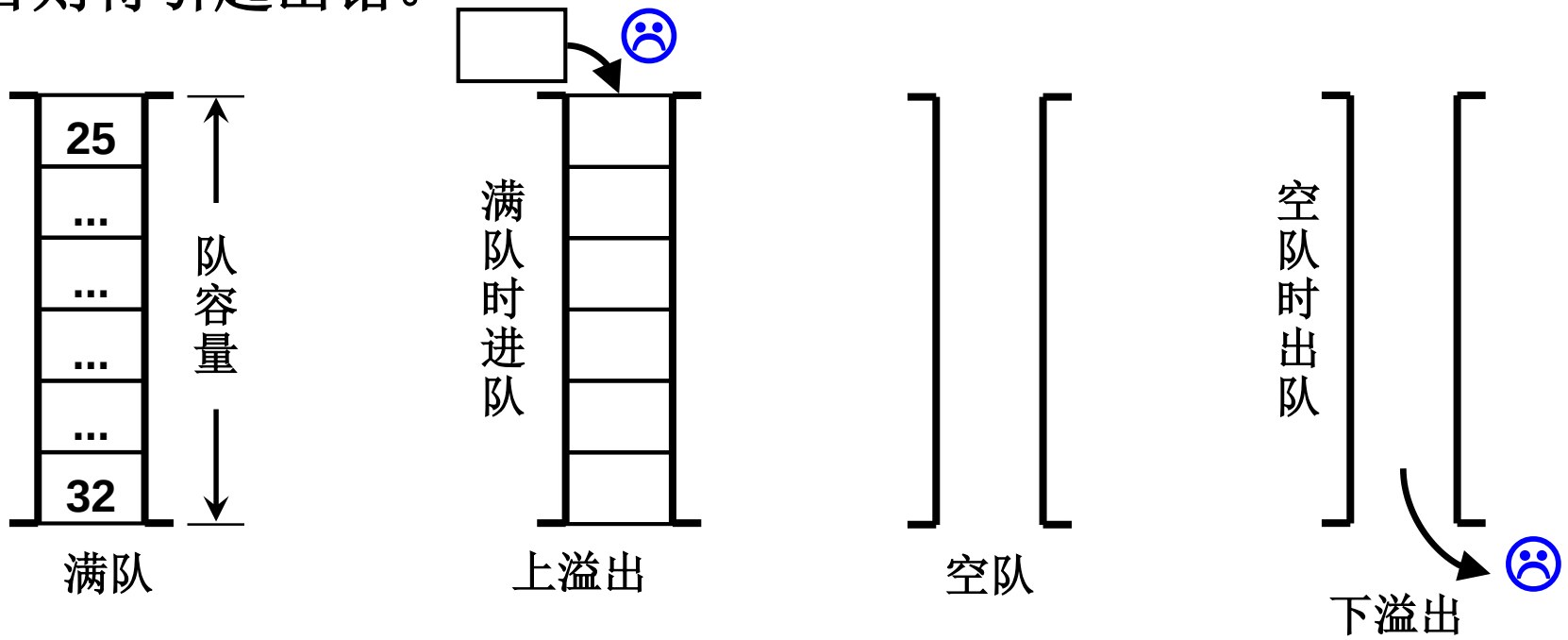
➢4.3.4 队

➢队的基本操作和存储空间测试

如果在空队(队内没有结点)时执行出队操作，称为**下溢出(underflow)**。

通常根据队首指针和队尾指针的变化来判断是否发生上溢出和下溢出。

在程序实现时，必须对上溢出和下溢出的情况加以处理，否则将引起出错。



➢4.3.5 顺序队列

➢数据定义

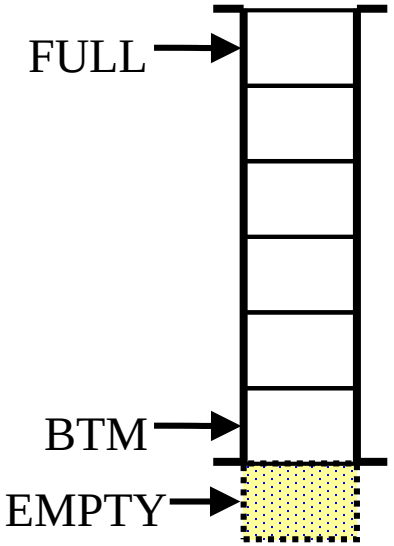
定义数组Q[M]，用Q[0]到Q[M-1]来存放队列的结点。设队首指针为Head，队尾指针为Tail，则队首结点为Q[Head]，队尾结点为Q[Tail]。初始时令Head和Tail都等于EMPTY，表示空队。

```
#define   M        1000
#define   FULL     M-1        /* 队尾指针的最大值          */
#define   BTM      0          /* 队首指针的最小值          */
#define   EMPTY    BTM-1      /* 空队指针值                */
short     Q[M];               /* 用数组存放队列            */
short     Head=EMPTY;         /* 队首指针，初始为空队      */
short     Tail=EMPTY;         /* 队尾指针，初始为空队      */
```

➢4.3.5 顺序队列

➢数据定义

定义数组Q[M]，用Q[0]到Q[M-1]来存放队列的结点。设队首指针为Head，队尾指针为Tail，则队首结点为Q[Head]，队尾结点为Q[Tail]。初始时令Head和Tail都等于EMPTY，表示空队。



➢顺序队列的出队操作

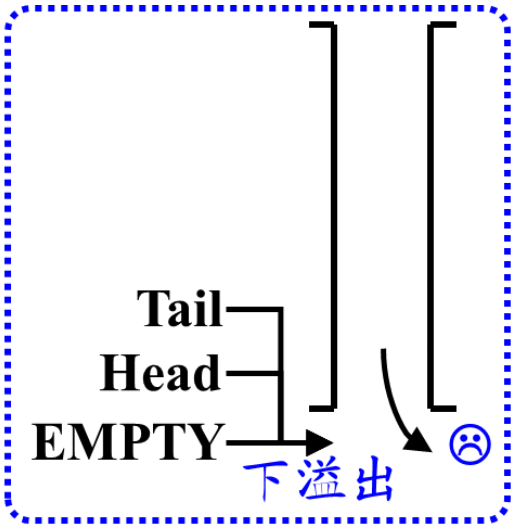
- 下溢出

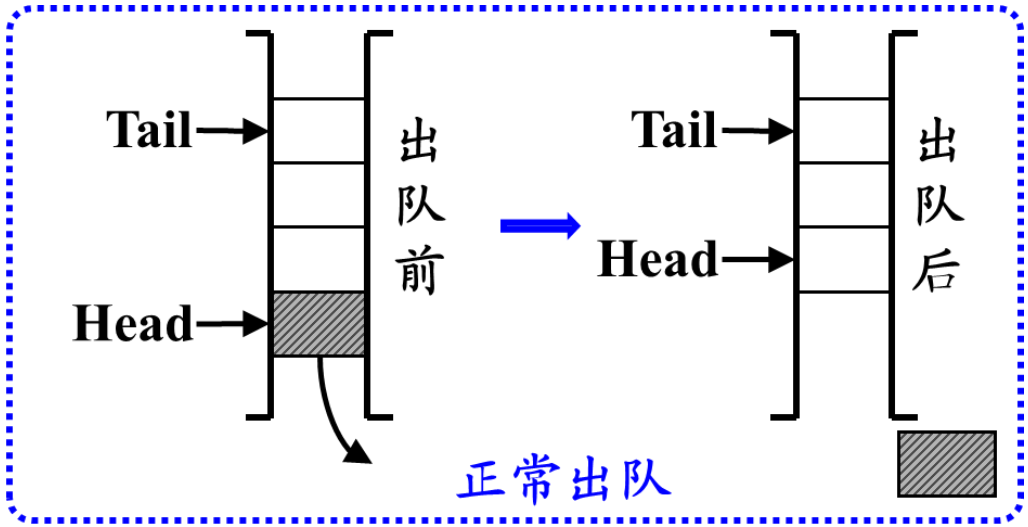
结点出队时，如果Head和Tail都等于EMPTY，表示需要在空队中取出结点，说明程序设计有误或者运行过程出错，称为下溢出。

- 正常出队

如果Head小于Tail，表示队内有多个结点，可执行正常出队的操作，返回队首结点Q[Head]的值，并令Head加1。程序语句为：

```
key = Q[Head];                /* 取结点值   */
if(Head < Tail)               /* 正常出队   */
   Head++;
return(key) ;
```



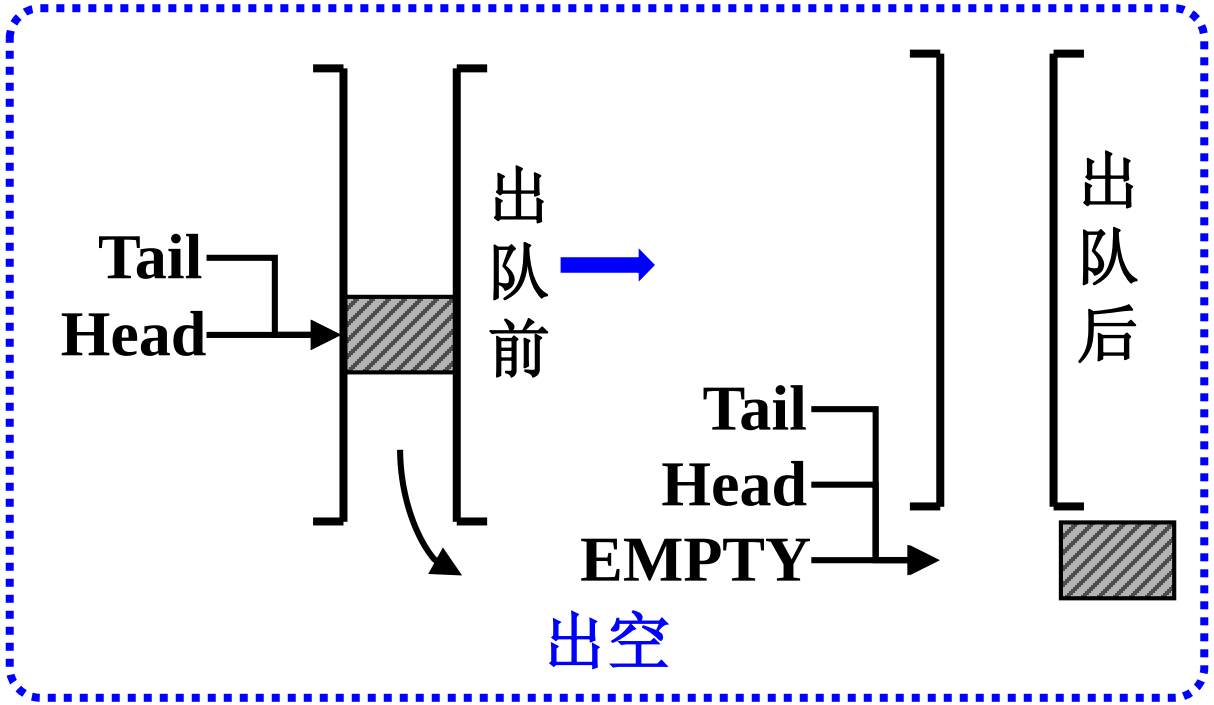


➢顺序队列的出队操作

- 出空

如果Head等于Tail而不等于EMPTY，表示队内只剩下1个结点，这种情况称为出空，则除了返回队首结点Q[Head]的值，还应该将Head和Tail都置为EMPTY。

```
key = Q[Head];
if(Head == Tail)   /* 出空 */
   Head = Tail = EMPTY;
return(key) ;
```



➢顺序队列的出队操作

●顺序队列的出队函数

```
short deQueue(void)
   {
      short key;
      if(Head< BTM|| Head>FULL|| Tail< BTM|| Tail>FULL )
         error();                  /* 非法  &  下溢出        */

      key=Q[Head];
      if(Head != Tail)       /* 正常出队      */
         Head++;
      else                          /* 出空          */
         Head = Tail = EMPTY;
      return(key);
   }
```

●调用方法示例

```
key=deQueue();
```

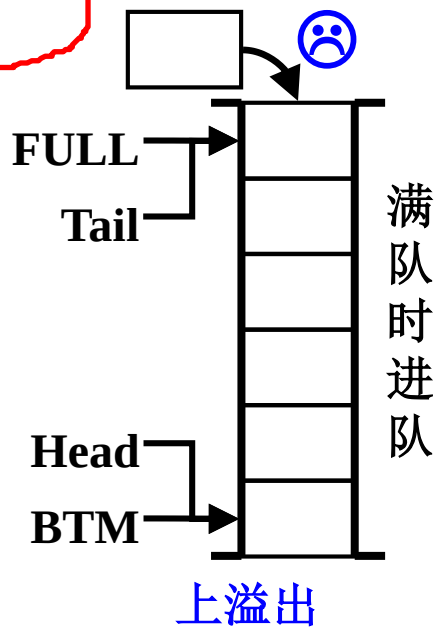
➢顺序队列的进队操作

结点进队时，只要队列中还有存储空间，就应该更新队尾指针Tail (以及队首指针Head)，并且将新结点key添加到Q[Tail]。

●上溢出

如果在满队情况下进队，即Tail等于FULL而且Head等于BTM，表示发生上溢出，说明队列容量不够。程序语句为：

```
if(Tail == FULL && Head == BTM)
   error();              /* 上溢出          */
```

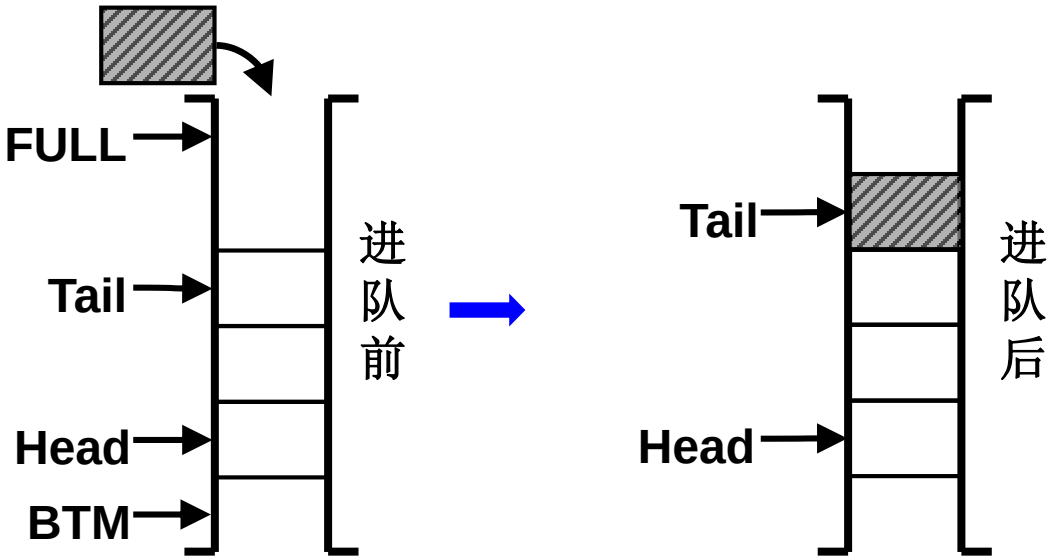


➢顺序队列的进队操作

●正常进队

如果BTM≤Head≤Tail<FULL，表示为正常进队情况，令队尾指针Tail加1，将新结点添加到Q[Tail]。程序语句为：

```
if(BTM <= Head || Head <= Tail || Tail < FULL)
   Q[++Tail] = key;        /* 正常进队          */
```



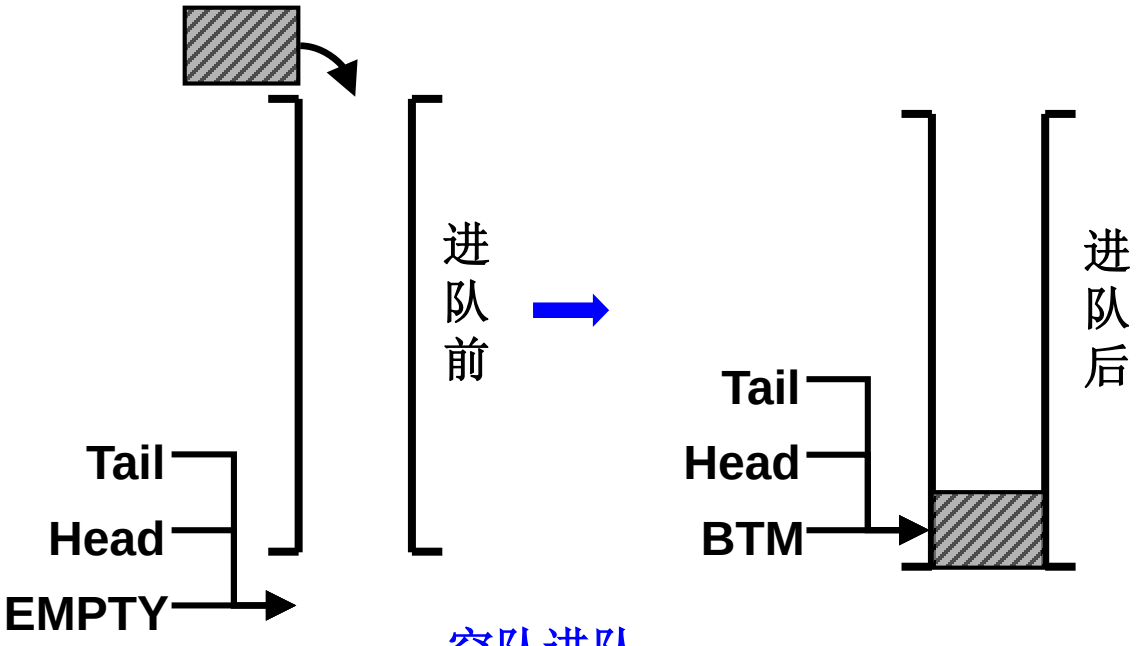
正常进队

➢顺序队列的进队操作

●空队进队

如果Tail和Head都等于EMPTY，表示空队进队，应该令Tail和Head都等于BTM。程序语句为：

```
if(Tail == EMPTY && Head == EMPTY)  /* 空队进队    */
   Head = Tail = BTM;               /* 或者Head++; Tail++; */
Q[Tail] = key;
```

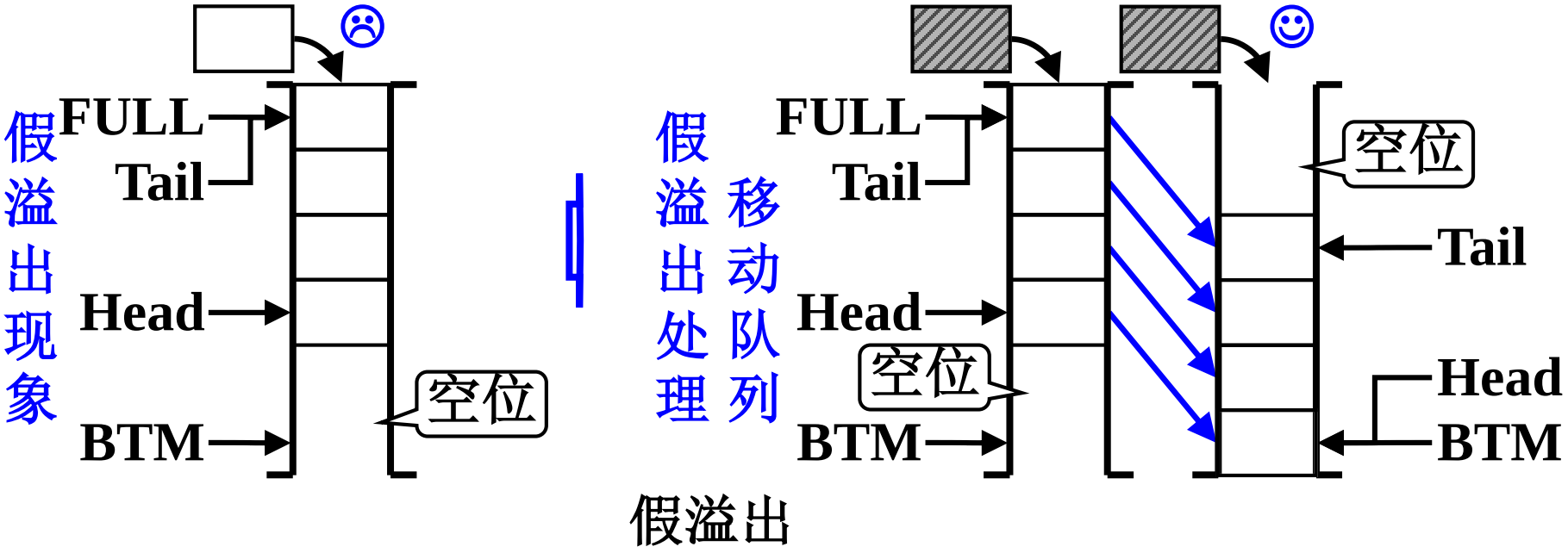


空队进队

➢顺序队列的进队操作

●假溢出

如果Tail等于FULL，而Head大于BTM，说明所有结点靠近队尾，队首方向还有空位，却不能在队尾添加结点，称为假溢出。



假溢出

➢顺序队列的进队操作

●假溢出

如果Tail等于FULL，而Head大于BTM，说明所有结点靠近队尾，队首方向还有空位，却不能在队尾添加结点，称为假溢出。

假溢出的处理办法是把所有结点向队首移动，使空位出现在队尾，然后更新首指针和尾指针，再在队尾添加结点。程序语句为：

```
if(Tail == FULL && Head > BTM)        /* 假溢出        */
{
   for(k=Head; k<=FULL; k++)          /* 移动队列      */
      Q[k-Head] = Q[k];
   Tail = Tail - Head + 1;            /* 更新指针      */
   Head = BTM;
   Q[Tail] = key;
}
```

➢顺序队列的进队函数

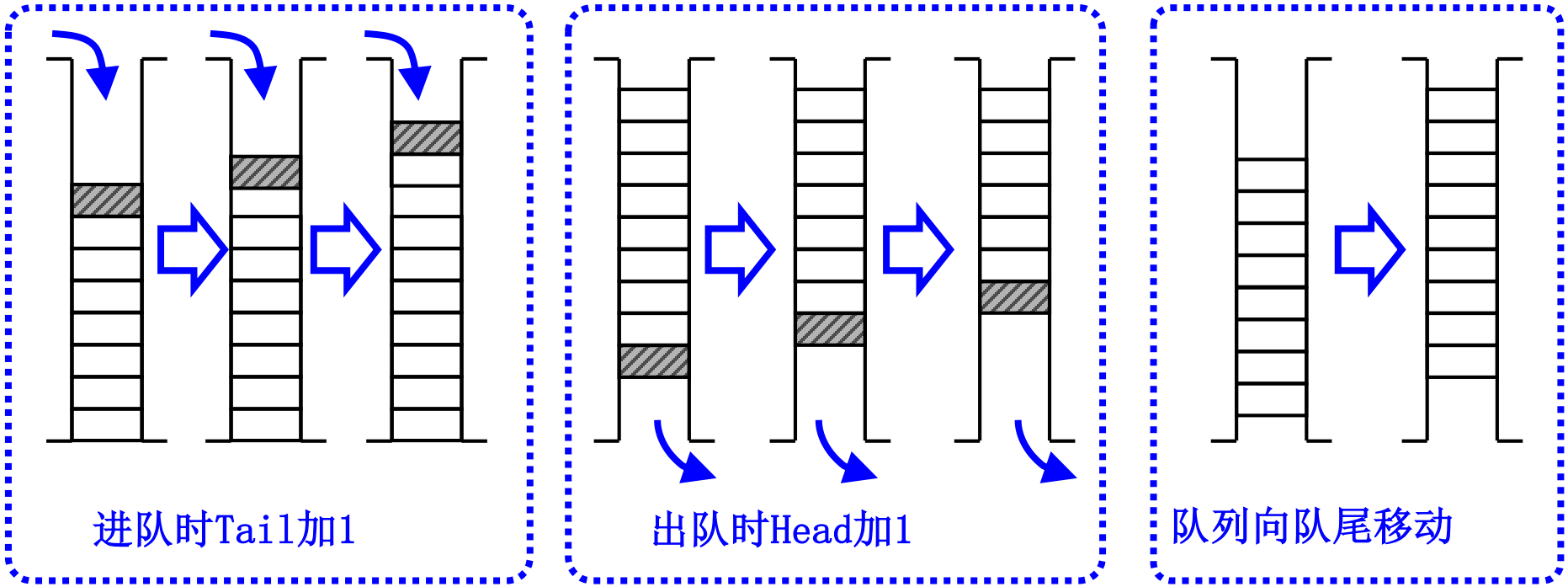
```
void enQueue(short key)
{
   short k;
   if(Head<EMPTY || Head>FULL || Tail<EMPTY || Tail>FULL)
      error();                                  /* 非法          */
   if(Tail==FULL && Head== BTM)
      error();                                  /* 上溢出        */
   if(Tail==FULL && Head>BTM)                   /* 假溢出        */
   {
      for(k=Head; k<=FULL; k++)                 /* 移动队列      */
         Q[k-Head] = Q[k];
      Tail = Tail - Head;                       /* 更新指针      */
      Head = BTM;
   }
   else if(Head==EMPTY)                         /* 空队进队      */
      Head++;
   Q[++Tail] = key;                             /* 包含正常进队    */
   return;
}
```

调用方法示例：**enQueue(key);**

➢ 顺序队列的问题

观察指针操作：进队时**Tail**加**1**，出队时**Head**加**1**。

说明通过进队和出队操作，整个队列将不断向队尾(数组的终端)方向移动。

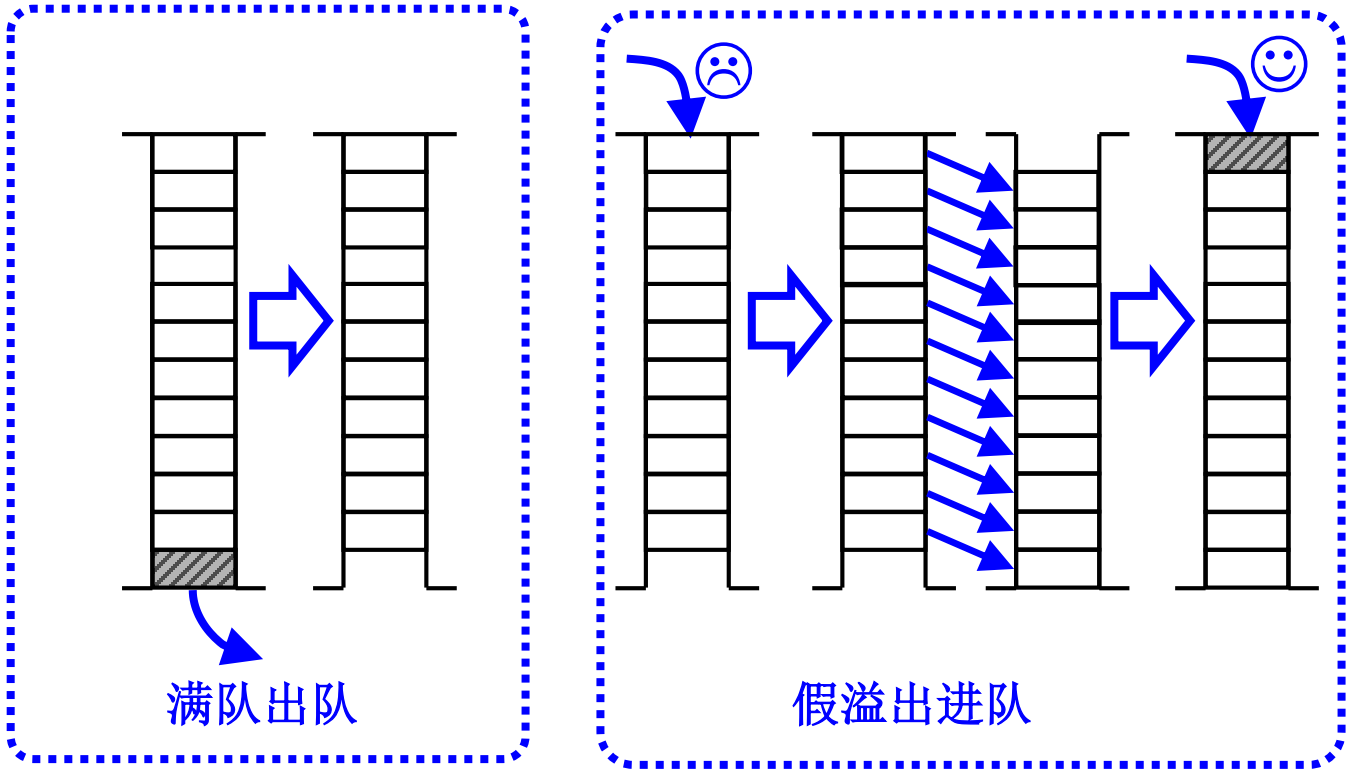


➢ 顺序队列的问题

观察指针操作：进队时Tail加1，出队时Head加1。

说明通过进队和出队操作，整个队列将不断向队尾(数组的终端)方向移动。

每当进队时遇到假溢出的情况，就需要执行整个队列向队尾移动的操作。特别地，在满队(Head==BTM&&Tail==FULL)时，交替地出队和进队，就将交替地发生满队出队和假溢出进队，每次都需要整体移动。显然，这种移动是很费时的，也是顺序队列的致命弱点。



➢4.3.6 环形队列

➢环形队列

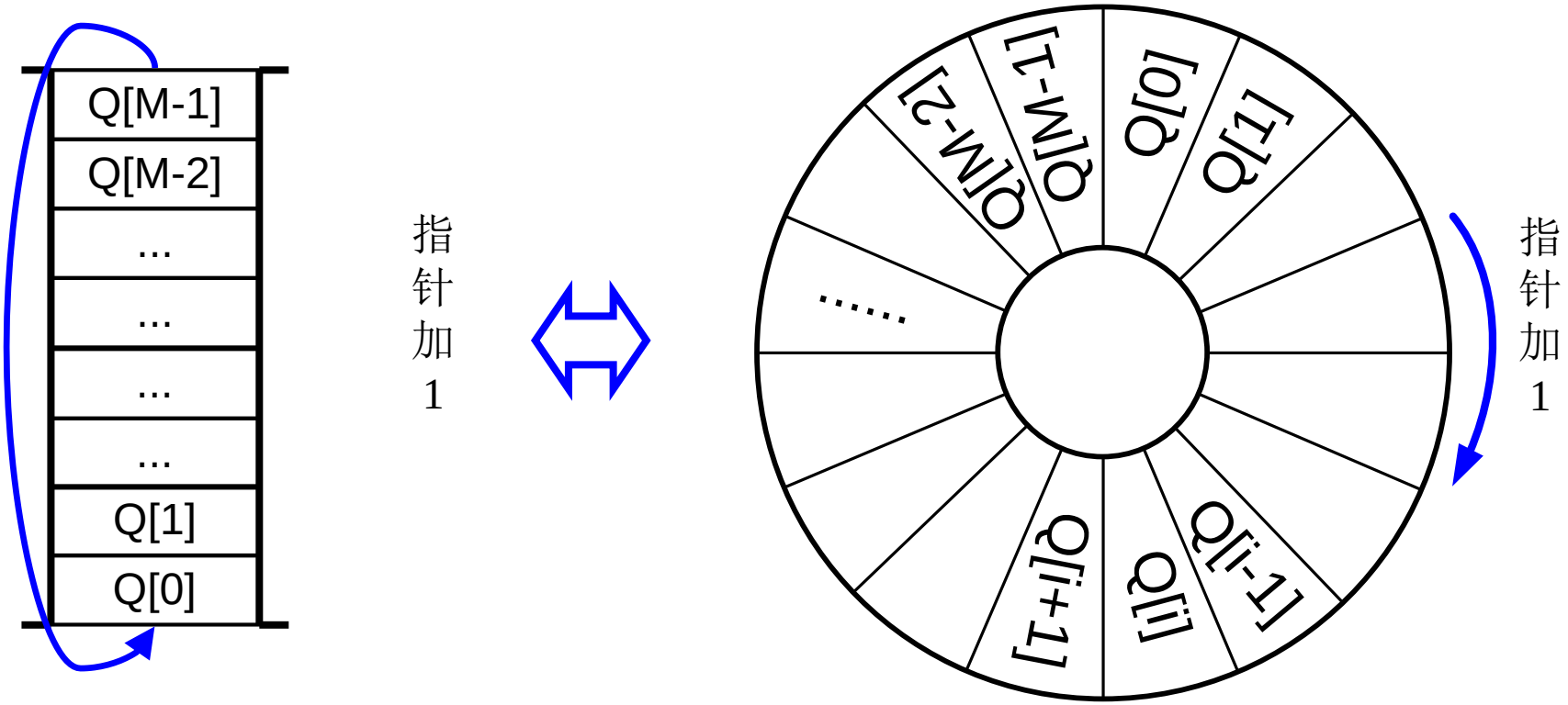
克服顺序队列反复移动的办法是引进环形队列，即在数组Q[M] 中将结点Q[0](Q[BTM])看成是结点Q[M-1](Q[FULL])的下一结点。

修改指针的加1操作：

将i++改为i=(i+1)%M，即把Head++改为Head=(Head+1)%M， 把Tail++改为Tail=(Tail+1)%M。则当指针i==M-1(i==FULL)时，(i+1)%M得0(BTM)。

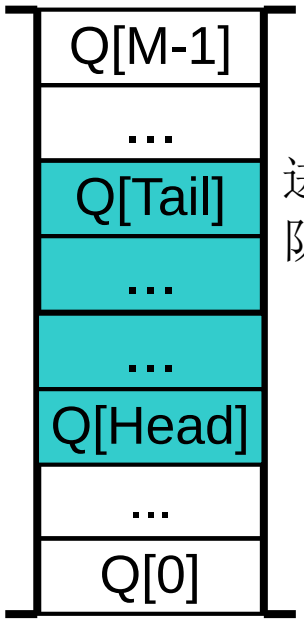
➢4.3.6 环形队列

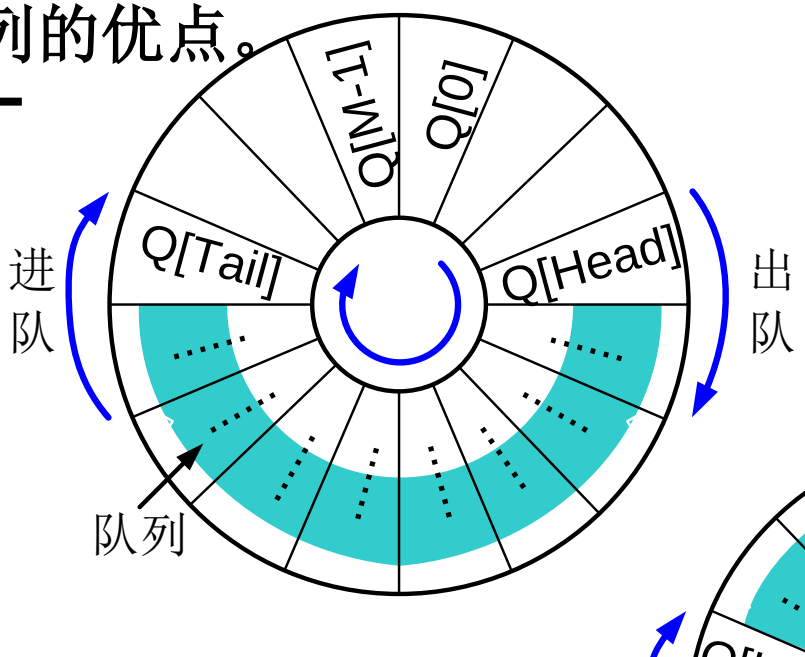
➢环形队列



➢环形队列的特点

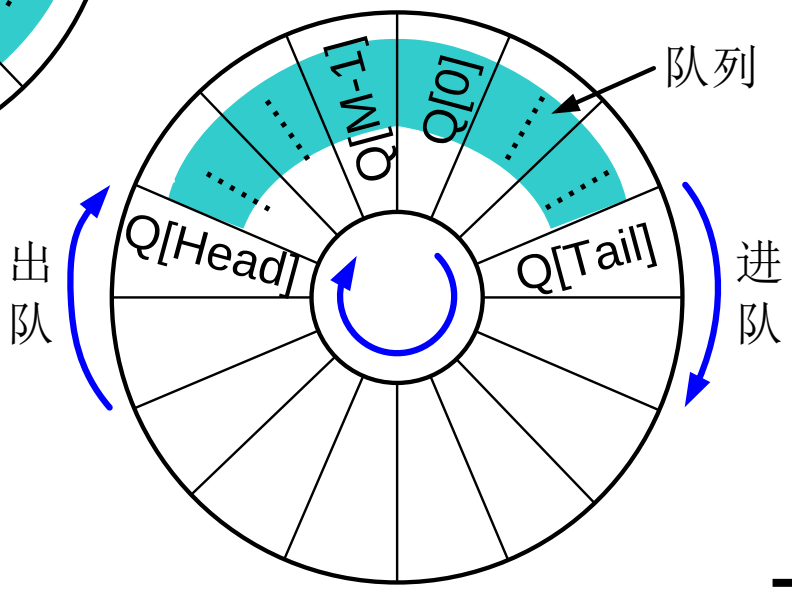
在环形队列中，**Head<Tail(左下图)和Head>Tail(右下图)的两种情况都有可能出现，把Head和Tail的加1操作改为加1后对M取模的操作，形成了数组的首尾相接，从而显示了环形队列的优点。**

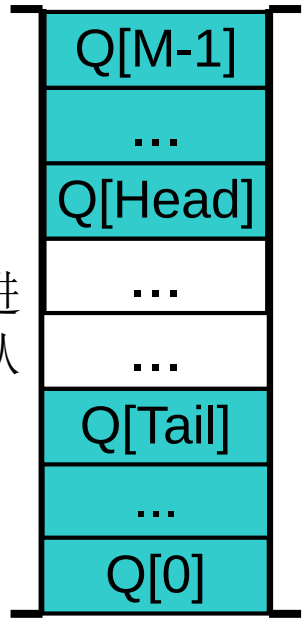




Head<Tail

Head>Tail

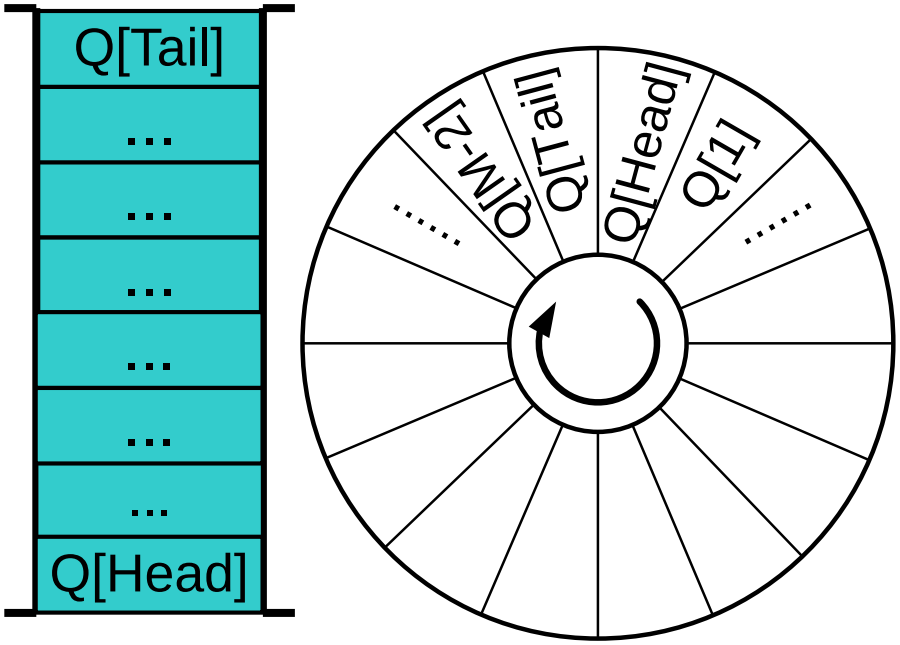




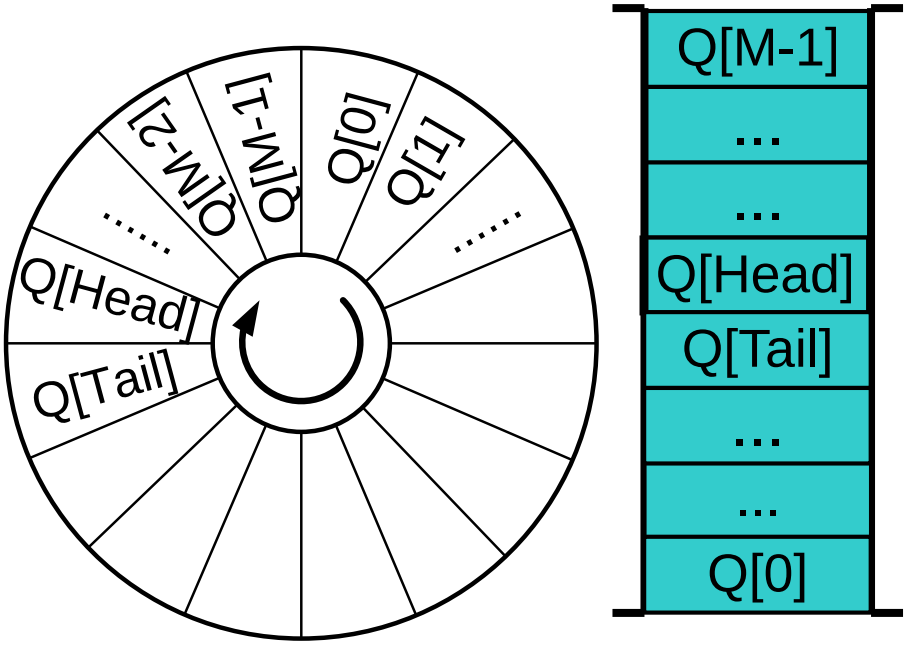
➢环形队列的操作

- 下溢出测试
  if(Head==EMPTY && Tail==EMPTY) /* 与顺序队列相同 */
- 上溢出测试
  无论是**Head>Tail**还是**Head<Tail**，测试条件相同：
  if((Tail+1) % M == Head)

Head<Tail
(Tail+1) % M == Head



Head>Tail
(Tail+1) % M == Head



➢环形队列的操作

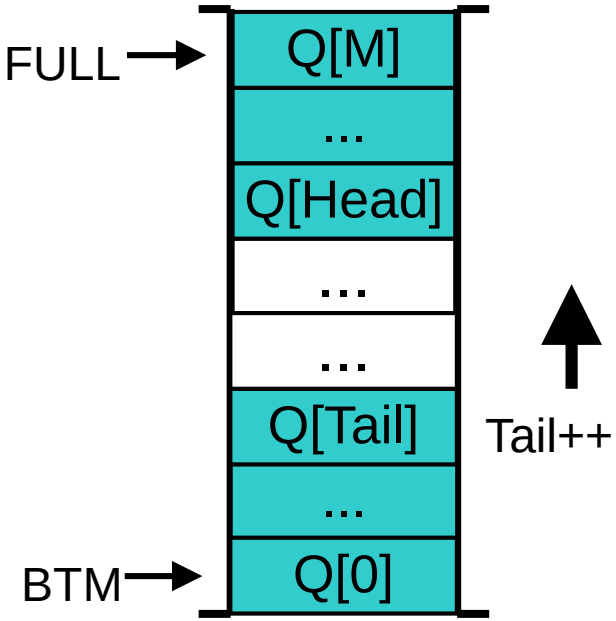
- 空队设置

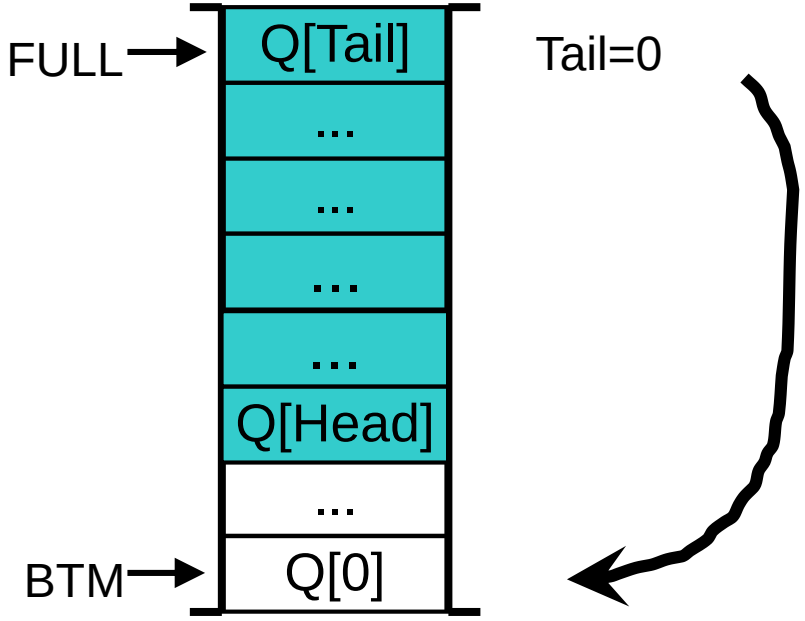
  Head = Tail = EMPTY;　　**/* EMPTY=-1;** 与顺序队列相同 ***/**

- 进队时的指针操作

正常(非空队)进队的指针操作：

Tail = (Tail + 1) % M;





空队进队的指针操作：

Head = Tail = BTM;

或者

Head = BTM;

Tail = (Tail + 1) % M;

➢环形队列的进队函数

```
void enQueue(short key)
{
   short k;
   if(Head<EMPTY || Head>FULL || Tail<EMPTY || Tail>FULL)
      error();                                  /* 非法          */
   if((Tail+1) % M == Head)
      error();                                  /* 上溢出        */
   if(Tail==FULL && Head>BTM)   /* 假溢出                      */
   {
      for(k=Head; k<=FULL; k++)  /* 移动队列                   */
        Q[k-Head] = Q[k];
      Tail = Tail - Head;                       /* 更新指针      */
      Head = BTM;
   }
   else if(Head==EMPTY)                         /* 空队进队      */
      Head = BTM;
   Tail = (Tail+1) % M;
   Q[Tail] = key;                               /* 包含正常进队  */
   return;
}
```

※ 调用方法示例

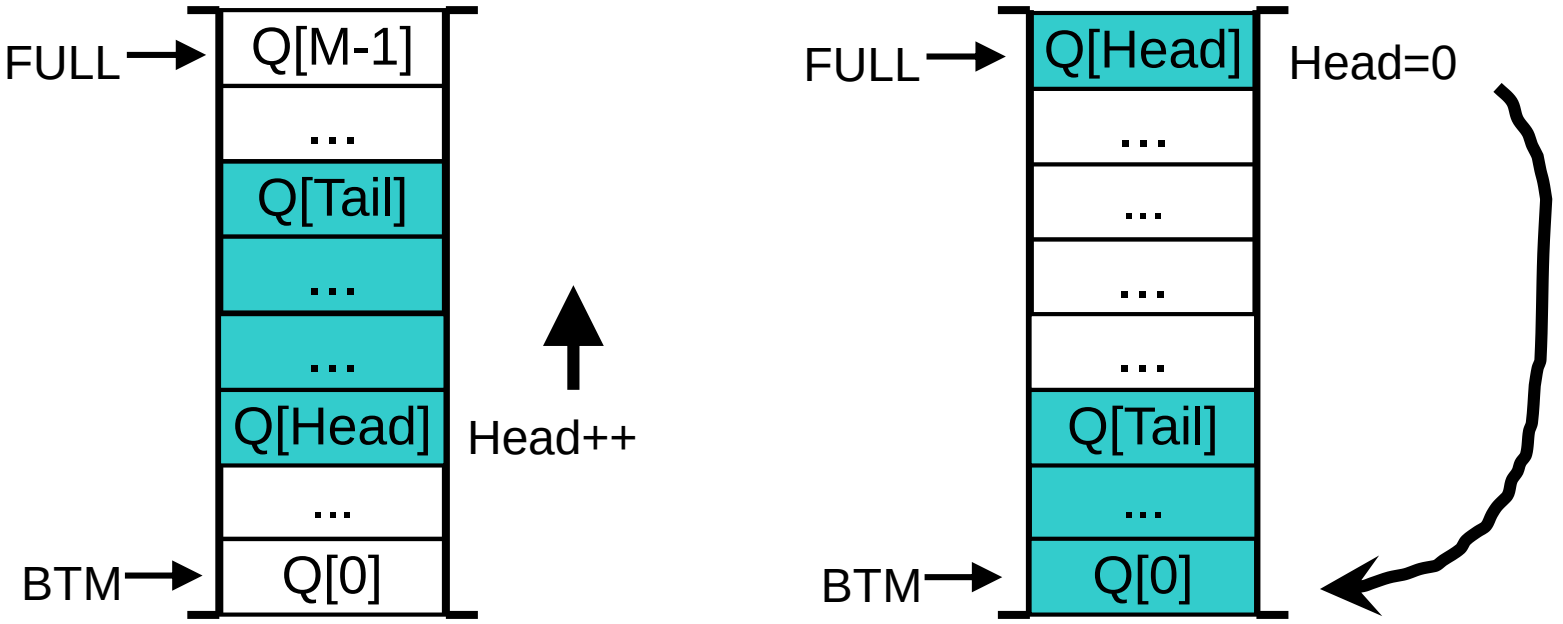
```
enQueue(key);
```

➢环形队列的操作

- 出队时的指针操作

正常出队时指针操作：

Head = (Head + 1) % M;



出空时的指针操作(置空)：

if(Head == Tail && Tail > EMPTY)

Head = Tail = EMPTY;

➢环形队列的出队函数

```
short deQueue(void)
{
   short key;
   if(Head< EMPTY || Head>FULL || Tail< EMPTY || Tail>FULL)
      error();                  /* 非法          */
   if(Head==EMPTY || Tail==EMPTY)
     error();                   /* 下溢出        */
   key=Q[Head];
   if(Head != Tail)        /* 正常出队      */
      Head = (Head + 1) % M;
   else                       /* 出空          */
      Head = Tail = EMPTY;
   return(key);
}
```

⊙调用方法示例

```
   key=deQueue();
```

➢4.3.7 链接队列

➢链接队列的设计

采用始端不含空结点的链表。采用空栈存放暂时不用的结点，使用教材252页的空栈出栈函数**popfree()**和进栈函数**pushfree()**。

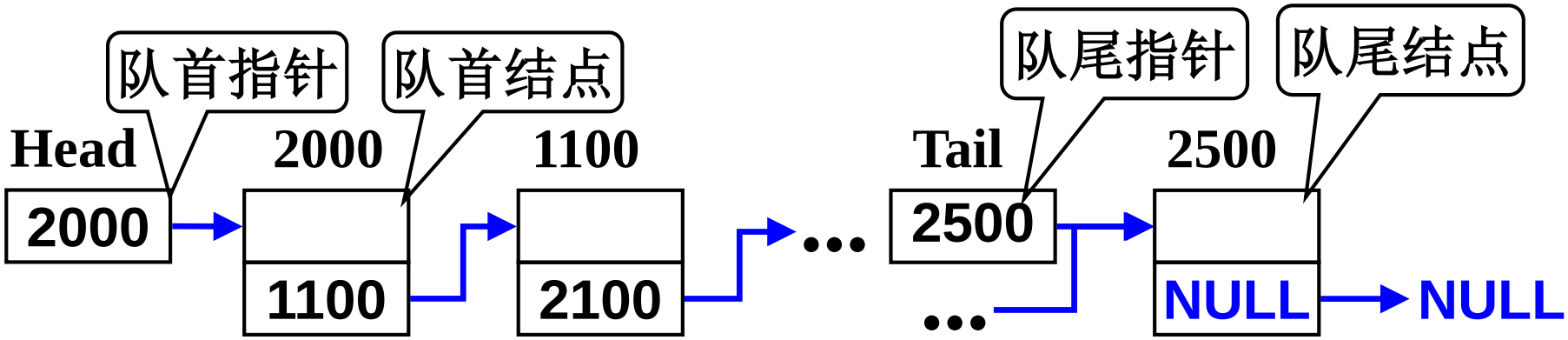
进队：采用表后插入，即在链表的尾端插入结点

出队：采用表前删除，即在链表的始端删除结点

➢4.3.7 链接队列

➢数据定义

```
#define   NODE  struct node
NODE
{
    short num;
    NODE *next;
};
NODE   *Head=NULL;      /* 队首指针，初始为空队 */
NODE   *Tail=NULL;      /* 队尾指针，初始为空队 */
```



➢链接队列的进队操作

- 空队时的进队操作

  Head = Tail = popFree();

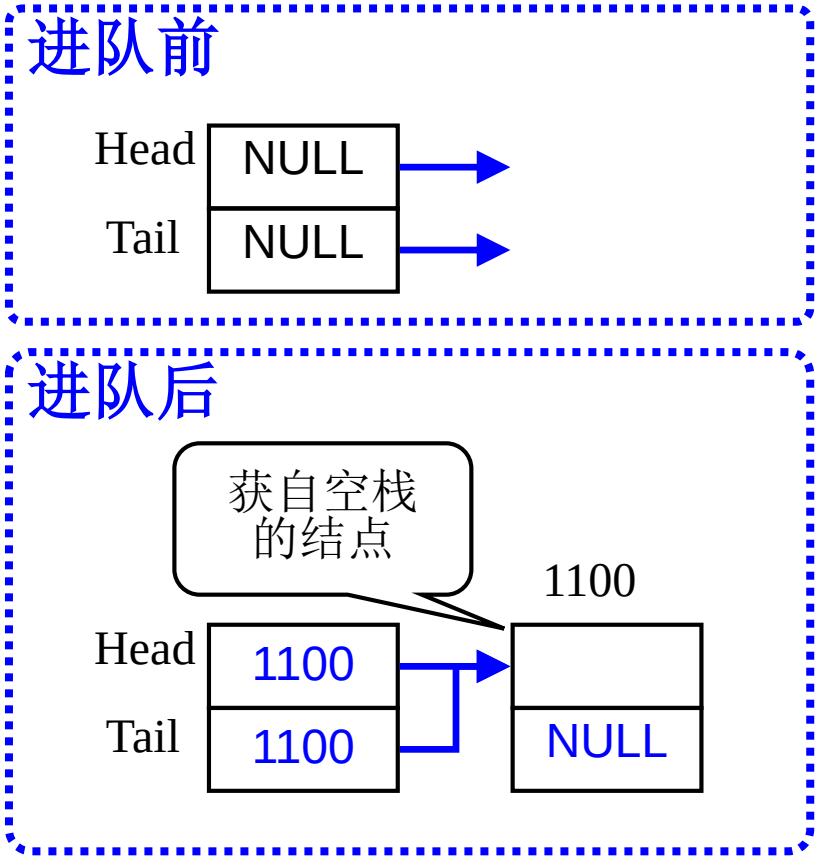
  Head->next = NULL;

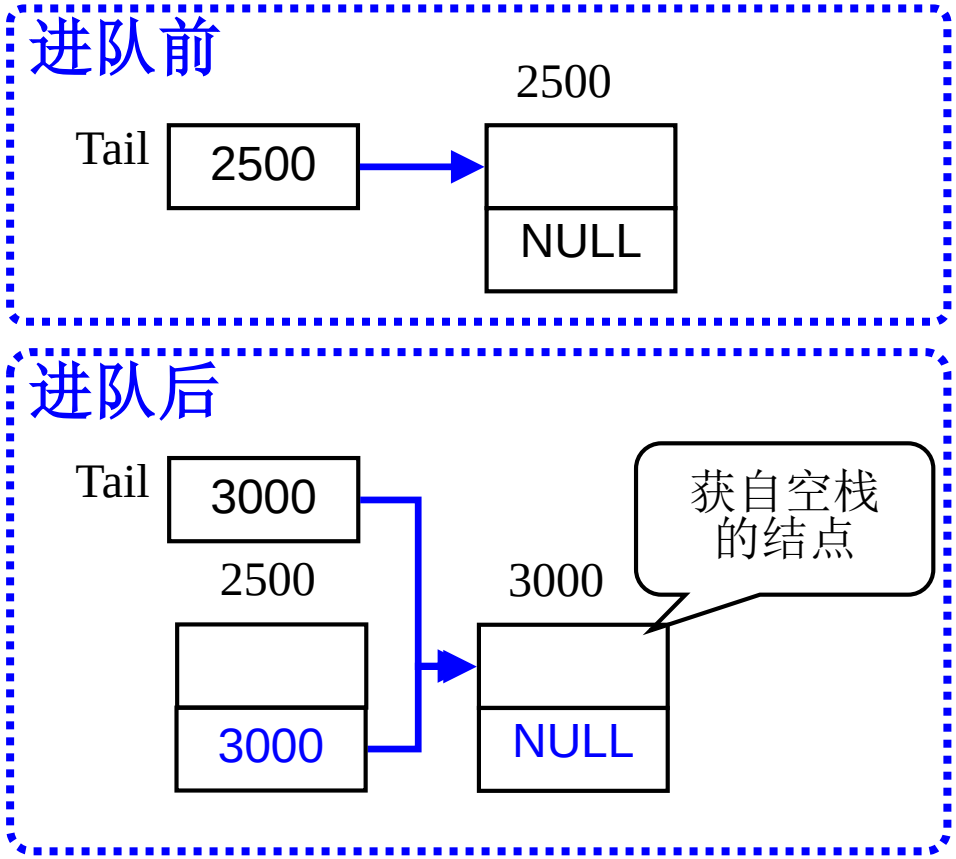
- 非空队的进队操作

  Tail->next = popFree();

  Tail = Tail->next;

  Tail->next = NULL;





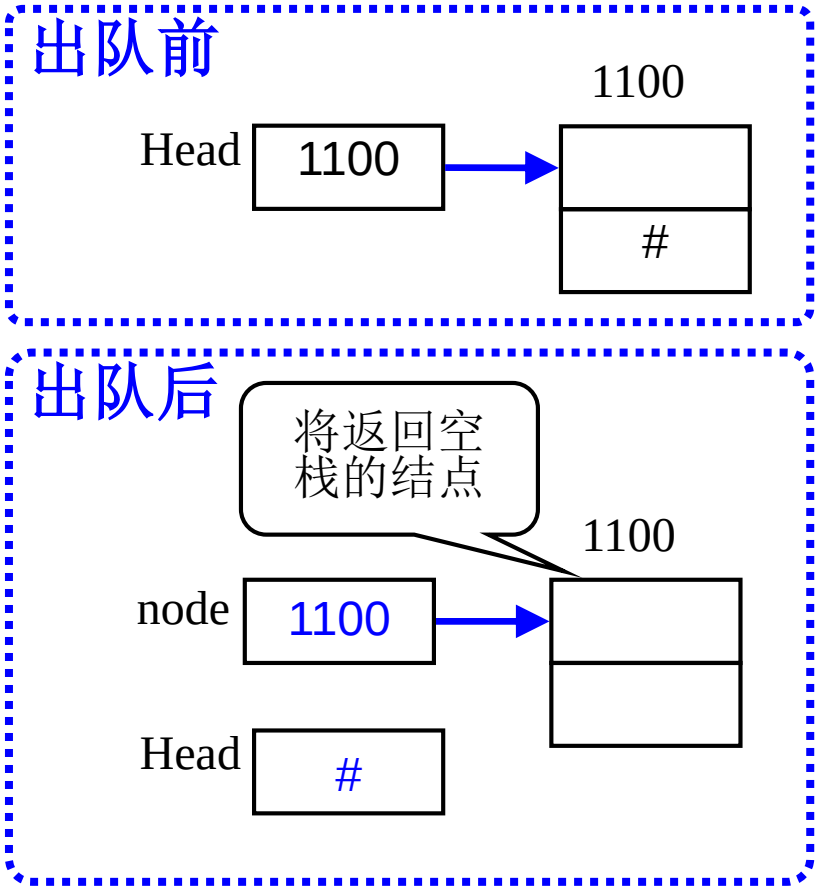
➢链接队列的出队操作

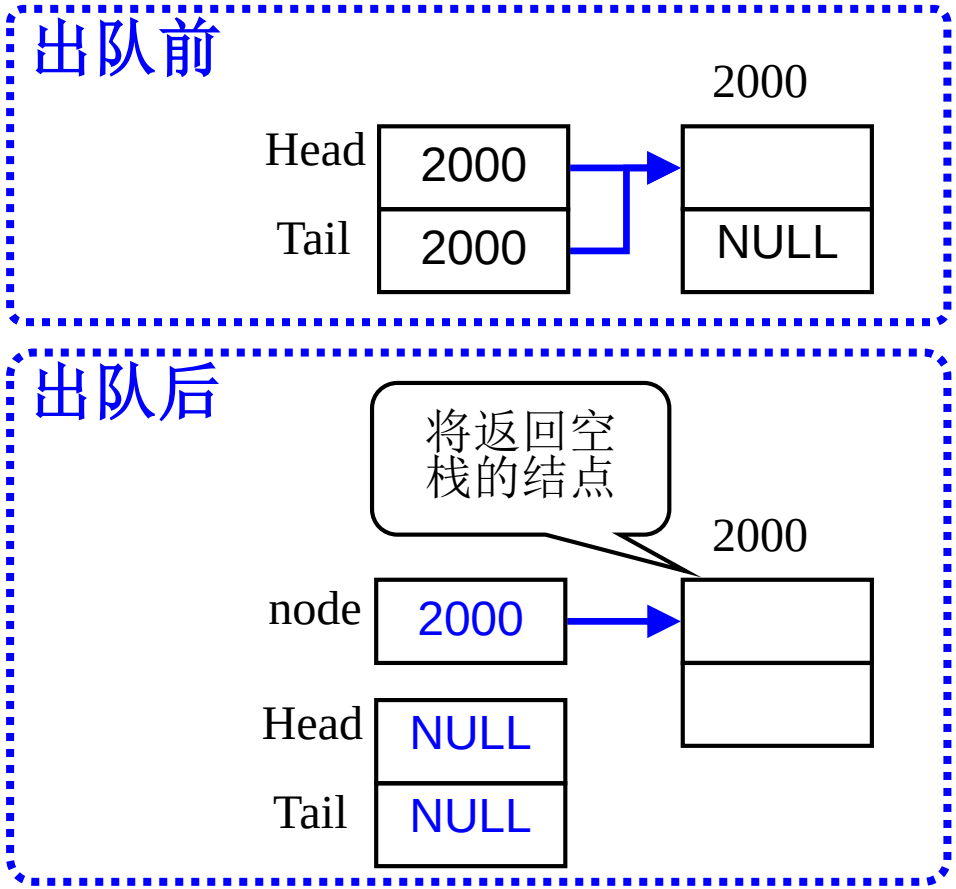
- 正常出队操作

```
node = Head;
Head = Head->next;
pushFree(node);
```

- 出空操作

```
node = Head;
Head = Head->next;
if (Head==NULL)
    Tail = NULL;
pushFree(node);
```





➢链接队列的函数实现

- 进队函数

```
void   enQueue(short key)
{
if(Tail==NULL)     /* 空队*/
    Head = Tail = popFree();
else   /* 非空队*/
{
    Tail->next = popFree();
    Tail=Tail->next;
}
Tail->num = key;
Tail->next = NULL;
}
```

- 调用方法示例

```
enQueue(key);
```

- 出队函数

```
short deQueue()
{
   NODE *node;
   short    key;
   if(Head==NULL)
   /*underflow*/
      error();
   key = Head->num;
   node = Head;
   Head = Head->next;
   if (Head==NULL)
      Tail = NULL;
   pushFree(node);
   return(key);
}
```

- 调用方法示例

```
key = deQueue( );
```

➢小结

栈

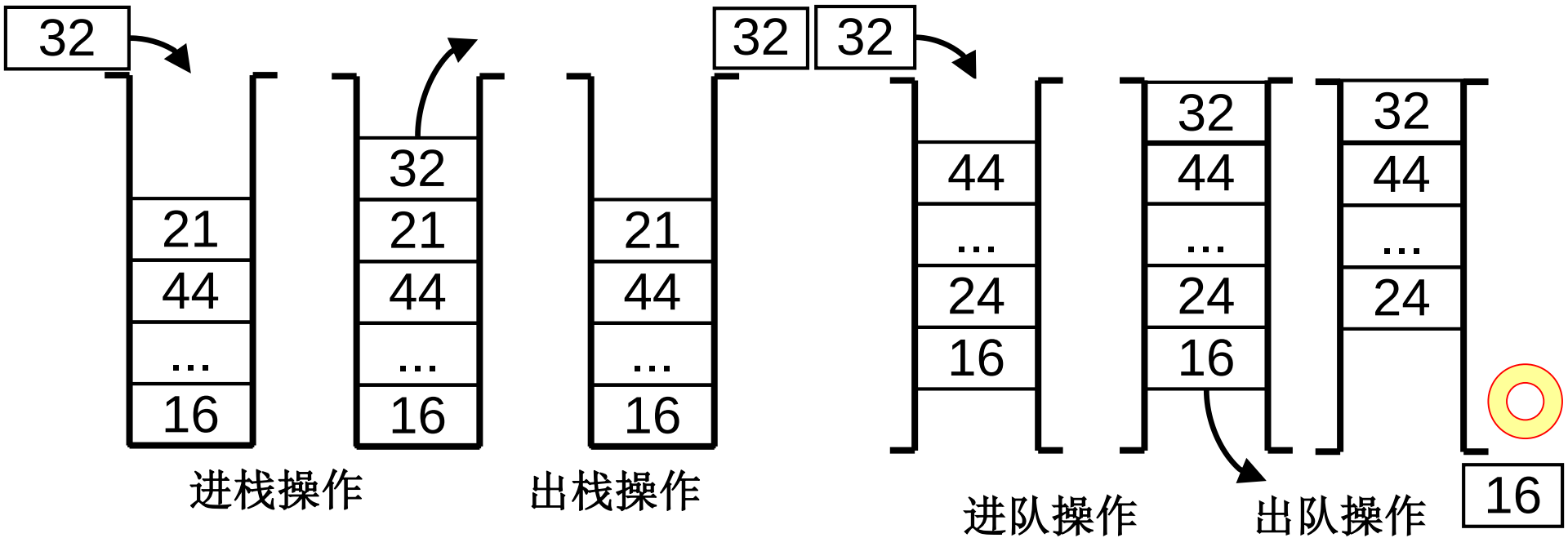
最先进栈的结点将最后出栈。因此，栈又被称为先进后出表，记为**FILO表(First In Last Out List)**。

⊙顺序栈、链接栈

队列

最先进队的结点必定最先出队。因此，队又被称为先进先出表，记为**FIFO表(Fist In Fist Out List)**。

⊙顺序队列、环形队列、链接队列



作　　业

- 习题　　4-15(逆波兰表达式),
  4-37(车厢编组),
  4-16(选择题)