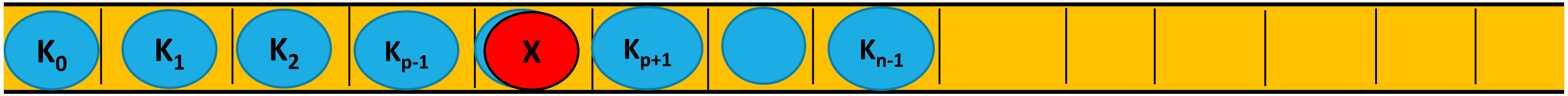
2.3 顺序表插入和删除

$(k_0, k_1, \ldots, k_{p-1}, k_p, \ldots, k_{n-1})$

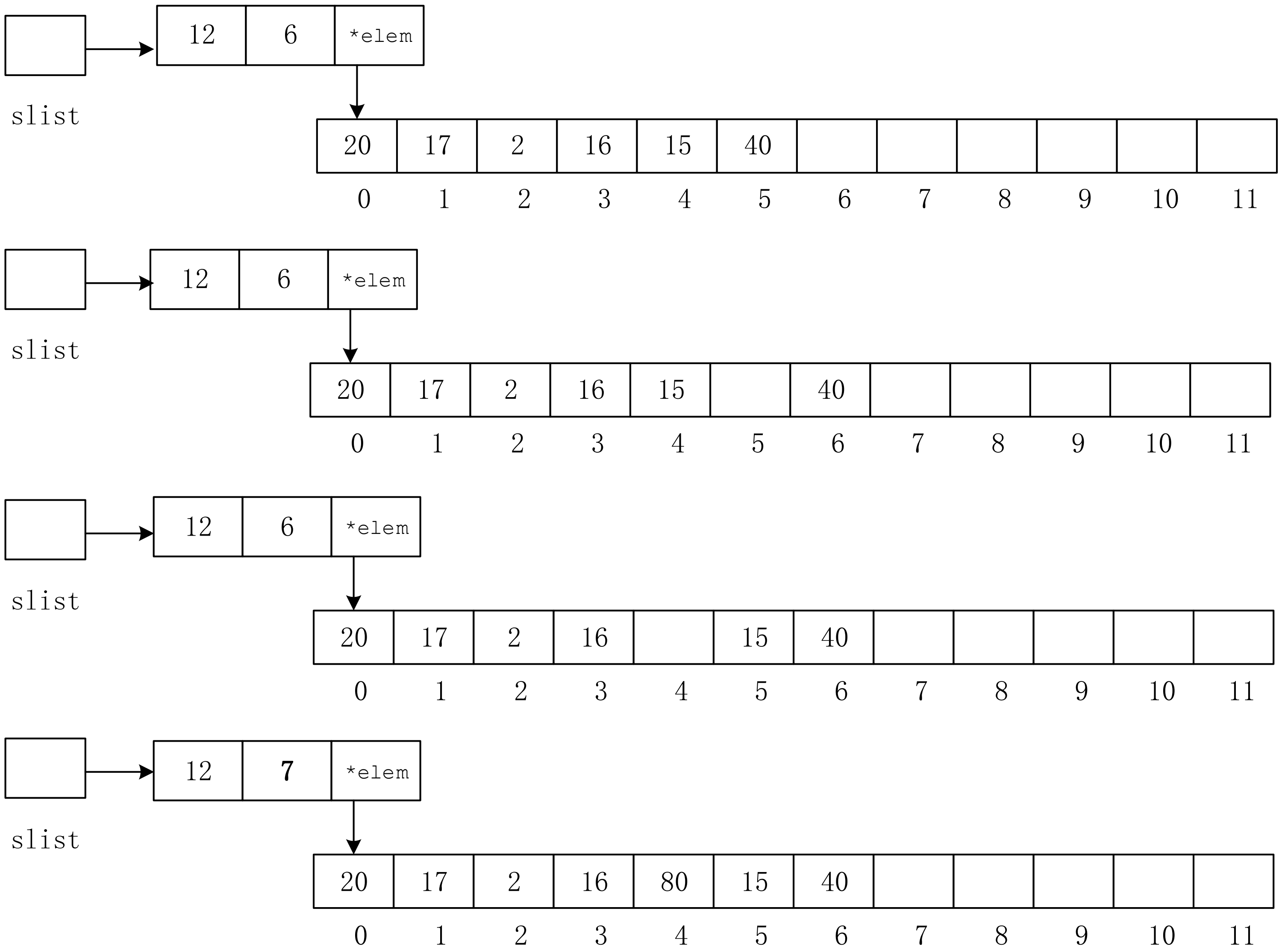
$(k_0, k_1, \ldots, k_{p-1}, \color{red}{x}, k_p, \ldots, k_{n-1})$



1.移动结点　2.插入结点　3.增加表长

- **先检查表空间是否满了，在表满的情况下不能再做插入，否则产生溢出错误**
- **要检验插入位置的有效性，这里 p 的有效范围是：0<=p<=n，其中 n 为原表长**
- **注意数据的移动方向：从下标大的元素开始**

顺序表插入举例



12
6
*elem
slist
20 17 2 16 15 40
0 1 2 3 4 5 6 7 8 9 10 11
12
6
*elem
slist
20 17 2 16 15 40
0 1 2 3 4 5 6 7 8 9 10 11
12
6
*elem
slist
20 17 2 16 15 40
0 1 2 3 4 5 6 7 8 9 10 11
12
7
*elem
slist
20 17 2 16 80 15 40
0 1 2 3 4 5 6 7 8 9 10 11

插入算法　　算法2-6

```
int InsertPre_seq(SeqList slist, int p, DataType x)
{//在线性表slist的p位置之前插入x，成功返回1，否则返回0
	int q;
	if(slist->n >= slist->Max){ //顺序表满溢出
		printf("overflow");
		return(0);
		}
	if(p<0 || p>slist->n){  //不存在下标为p的元素
		printf("not exist!\n");
		return(0);
		}
	for (q = slist->n - 1; q >= p; q--)//插入位置以及之后的元素后移
		slist->elem[q+1] = slist->elem[q];
	slist->elem[p] = x;  //插入元素x
	slist->n = slist->n + 1;  //顺序表长度加1
	return(1);
}
```

插入算法时间复杂度

$(k_0, k_1, \ldots, k_{i-1}, k_i, \ldots, k_{n-1})$

$(k_0, k_1, \ldots, k_{i-1}, x, k_i, \ldots, k_{n-1})$

算法的时间主要花费在结点的移动上，在表中第i个位置上插入一个结点的移动次数为（n - i)

当i=n时，无须移动结点；

$$M_i = \sum_{i=0}^{n}(n-i)P_i \qquad P_i = \frac{1}{n+1}$$

最好时间复杂度O(1)

当i=0时，须移动表中所有结点；

$$= \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{n}{2}$$

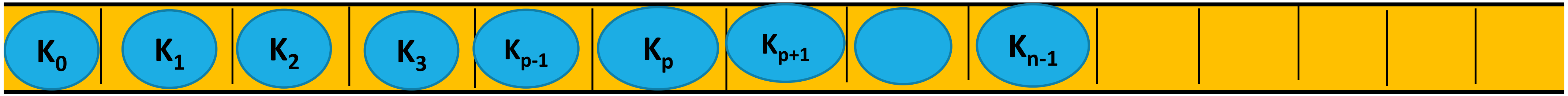
最坏时间复杂度O(n)

平均时间复杂度O(n)

2.3 顺序表插入和删除

$(k_0,\ k_1,\ \dots,\ k_{p-1},\ k_p,\ k_{p+1}\ \dots,\ k_{n-1})$

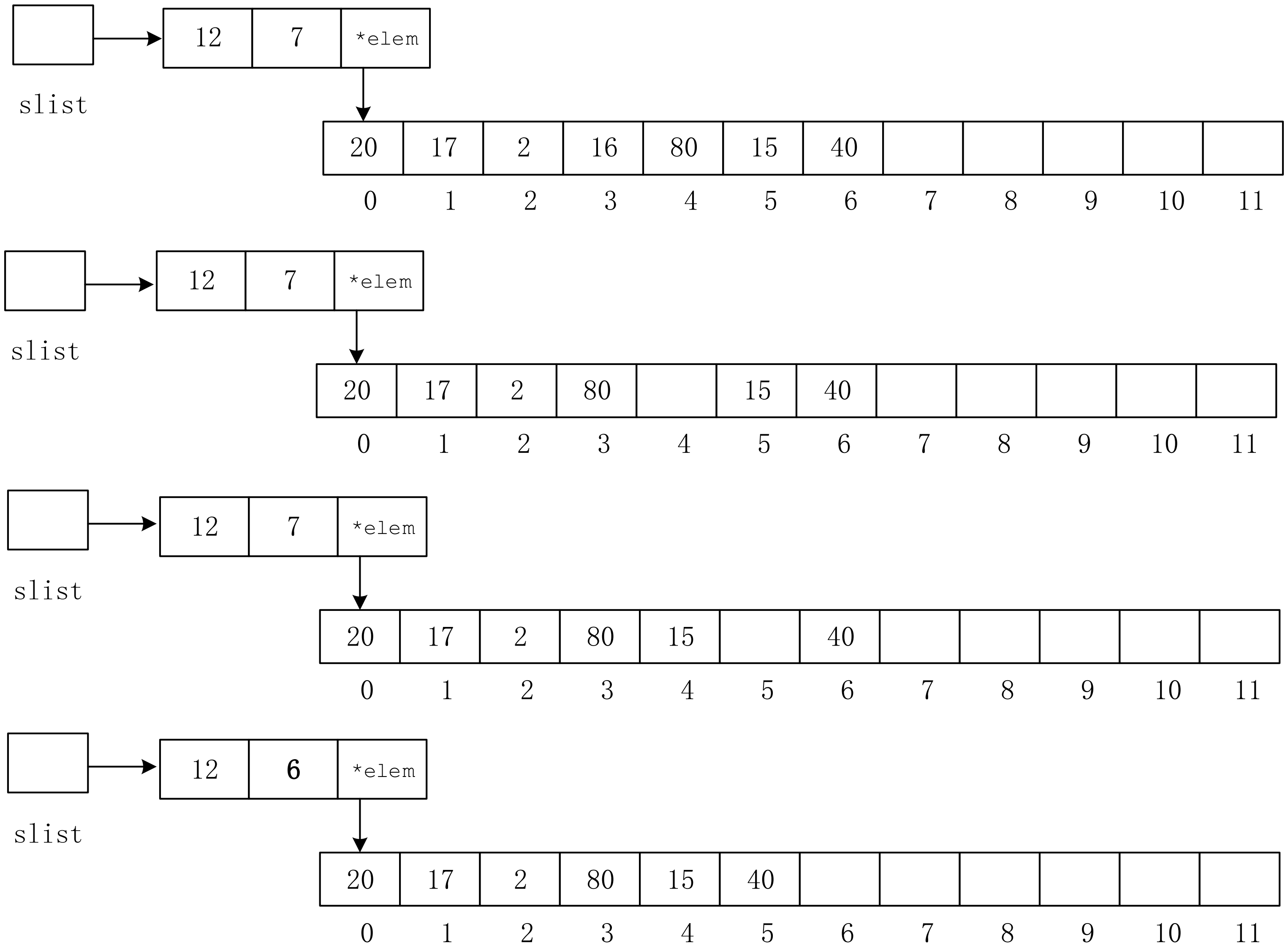
$(k_0,\ k_1,\ \dots,\ k_{p-1},\ k_{p+1},\ \dots,\ k_{n-1})$



1.移动结点　　**2.减少表长**

- **要检查删除位置的有效性0<=p<n**
- **注意数据的移动方向:从下标小的元素开始**

顺序表删除举例


12 7 *elem
slist
20 17 2 16 80 15 40
0 1 2 3 4 5 6 7 8 9 10 11
12 7 *elem
slist
20 17 2 80 15 40
0 1 2 3 4 5 6 7 8 9 10 11
12 7 *elem
slist
20 17 2 80 15 40
0 1 2 3 4 5 6 7 8 9 10 11
12 6 *elem
slist
20 17 2 80 15 40
0 1 2 3 4 5 6 7 8 9 10 11

顺序表删除算法

算法2-7

```c
int DelIndex_seq(SeqList slist ,int p) //删除下标为p的元素
{
        int q;
        if(p<0||p>=slist->n){   //不存在下标为p的元素
                printf("Not exist\n");
                return 0;
        }
        for (q = p; q<slist->n-1; q++){ //下标p之后的元素向前移动
                slist->elem[q]=slist->elem[q+1];
        }
        slist->n = slist->n - 1; //顺序表长度减1
        return 1;
}
```

删除算法时间复杂度

$(k_0, k_1, \ldots, k_{i-1}, k_i, k_{i+1} \ldots, k_{n-1})$

$(k_0, k_1, \ldots, k_{i-1}, k_{i+1} \ldots, k_{n-1})$

算法的时间主要花费在结点的移动上，在表中删除第i个位置上一个结点的移动次数为（n-i-1）

当i=n-1时，无须移动结点；

最好时间复杂度O(1)

当i=0时，须移动表中所有结点；

最坏时间复杂度O(n)

$$M_d = \sum_{i=0}^{n-1}(n-i-1)P_i \qquad P_i = \frac{1}{n}$$

$$= \frac{1}{n}\sum_{i=0}^{n}(n-i-1) = \frac{n-1}{2}$$

平均时间复杂度O(n)