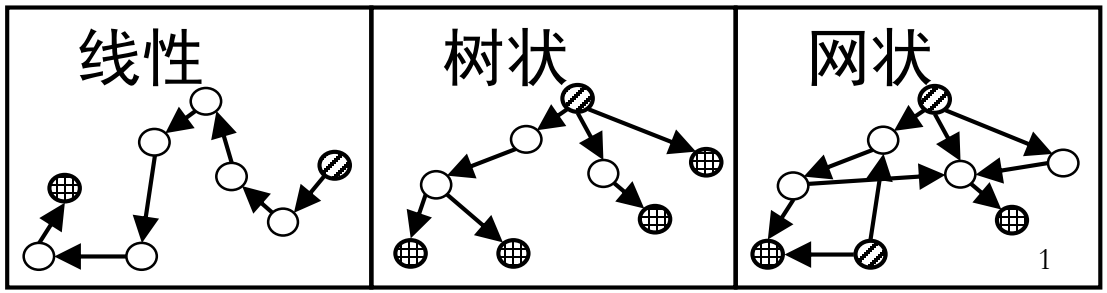
第四章 数据结构

- 4.1 基本概念
- 4.2 线性表
- 4.3 栈和队列
- 4.4 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
- 4.8 图



第四章 数据结构

- 4.1 基本概念
- 4.2 线性表
  - 4.2.1 基本概念
  - 4.2.2 查找结点
  - 4.2.3 添加结点
  - 4.2.4 删除结点
  - 4.2.5 排序
  - 4.2.6 存储方式与算法复杂度
- 4.3 栈和队列
- 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

➢4.2 线性表

线性表(list)，又称并列表，线性并列表，或者有序表。

➢4.2.1 基本概念

➢线性表的定义

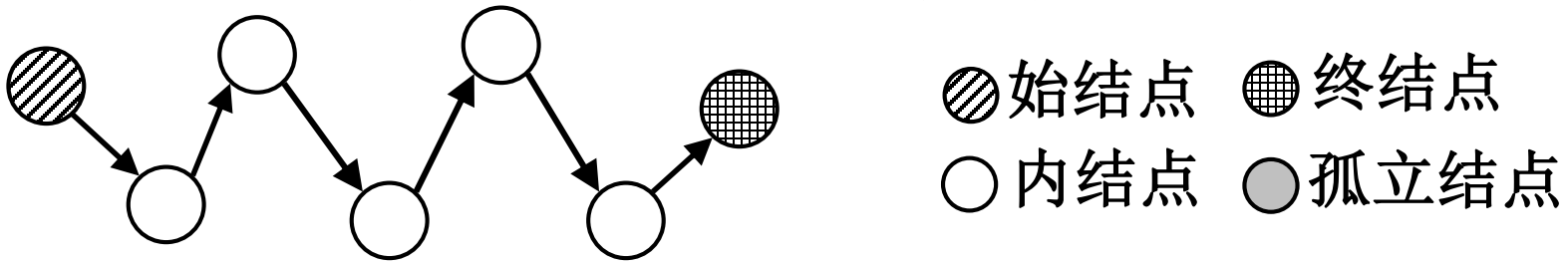
设B=(K, R), 若K中有n个结点$K = \{k_0, \ldots, k_{n-1}\}$，R中只有一种关系N，即$N=\{(k_i,k_{i+1}) \mid k_i, k_{i+1} \in K, 0 \le i \le n-1\}$，则称B为线性表。

➢推论

有且仅有一个始结点和一个终结点，其余为内结点。

除终结点外，每个结点有且仅有一个后件。

除始结点外，每个结点有且仅有一个前件。



➢课程约定

- 如果没有特别说明，数据结构中的结点k只有一个数据场分量。
- 假定可以调用一个已设计的出错函数error(), 用于出错处理。

➢4.2.1 基本概念

➢线性表的顺序存储

采用数组实现线性表的顺序存储。

- 存储单元

  $k_i = \{\alpha k_i, \delta k_i, pk_i\}$

  可省略指针场，其中，

  $\alpha k_i = i$

  $\delta k_i = a[i]$

  $pk_i = \alpha k_{i+1} = \alpha k_i + 1 = i+1$

存储单元定义

αk_0	$\delta k_0 = 0$
……	……
αk_i	$\delta k_i = i$
……	……
αk_{n-1}	$\delta k_{n-1} = n-1$

数组

a[0]
……
a[i]
……
a[n-1]
……
a[M]

- 数据定义

```
#define   M   1000
short n;          /* M>=n  */
short  a[M];    /* 在a[0]~a[n-1]中存放结点值δk0~ δkn-1  */
```

➢4.2.1 基本概念

➢线性表的顺序存储

- 生成线性表的程序

```
scanf("%hd", &n);
if(n > M  || n < 1)
    error();    /* 结点数超界    */
for(j=0; j<n; j++)
    scanf("%hd", &a[j]);
```

➢4.2.1 基本概念

➢线性表的链接存储

采用链表实现线性表的链接存储。

- 存储单元

  $k_i = \{\alpha k_i, \delta k_i, pk_i\}$

  其中，

  $pk_i = \alpha k_{i+1}$

- 数据定义

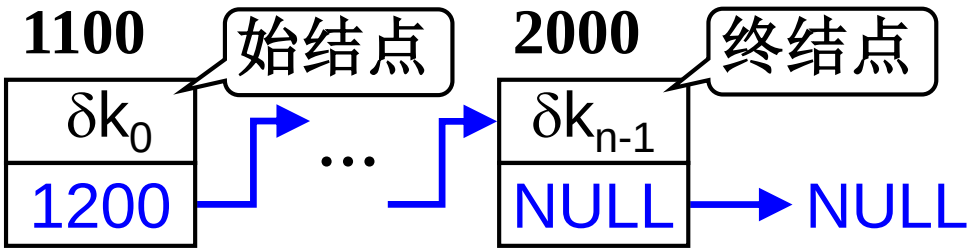
```
#define NODE struct node
NODE
{
   short    num;    /* δk_i     */
   NODE  *next;     /* αk_i+1   */
};
short      n;
```

存储单元定义

αk_0	δk_0	$pk_0 = \alpha k_1$
……	……	……
αk_i	δk_i	$pk_i = \alpha k_{i+1}$
……	……	……
αk_{n-1}	δk_{n-1}	$pk_{n-1} = \varphi$

存储单元取值

1100	*x*->num	1200
……	……	……
2000	*x*->num	NULL



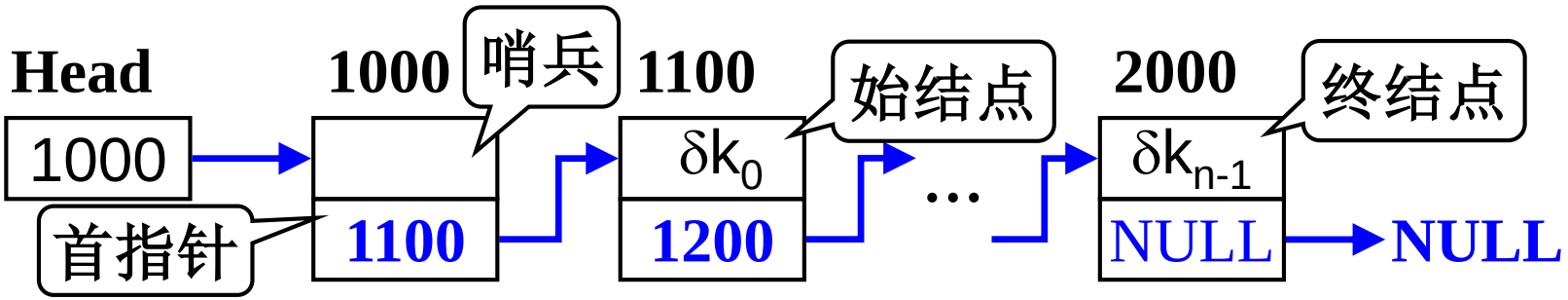
➢4.2.1 基本概念

➢线性表的链接存储

- 生成带哨兵的链表

指针Head指向一个空置的结点(哨兵)，首指针为Head->next，即Head->next为始结点k_0的地址αk_0，终结点k_{n-1}的指针next指向空(NULL)。

```
NODE   *Head;
/* 生成哨兵        */
Head = (NODE *)(malloc(sizeof(NODE)));
if(Head == NULL)
   error();  /* 内存分配失败    */
Head->next = NULL;
/* 生成链表          */
createList(Head);
```

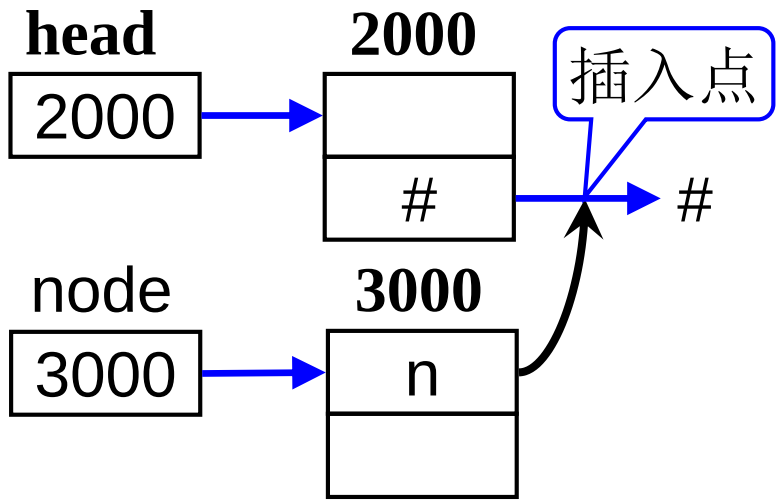


●链表生成函数

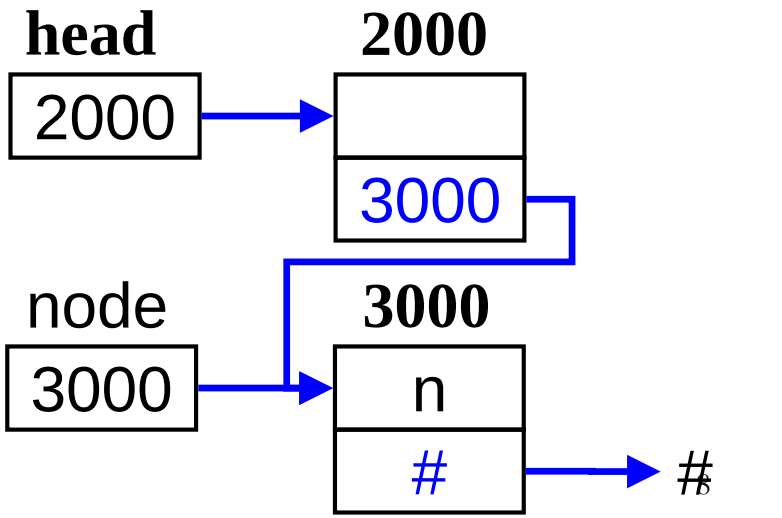
```
void createList(NODE *head)
{
   short  n, j, num;
   NODE   *node;
   scanf("%hd", &n);
   if(n < 1)
      error();  /* 结点数超界      */
   for(j=0; j<n; j++)
   {
      node = (NODE *)(malloc
         (sizeof(NODE)));
      if(node == NULL)
         error(); /* 内存分配失败   */
      scanf("%hd", &num);
      node->num = num;
      node->next = head->next;
      head->next = node;
   }
}
```

头部插入法

插入前



插入后

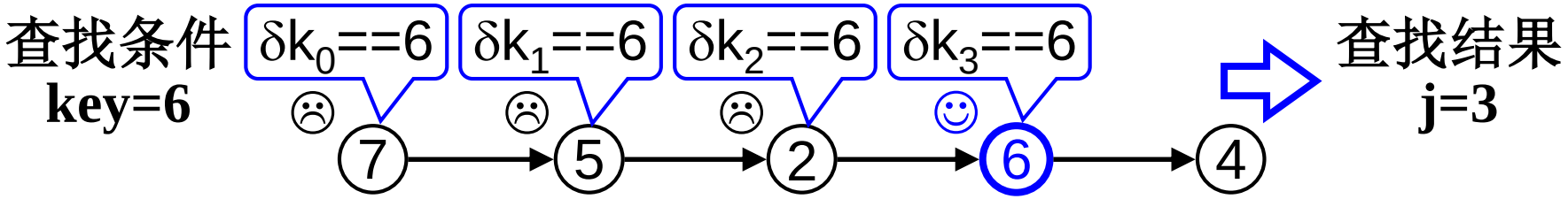


➢4.2.2 查找结点

➢查找结点的操作

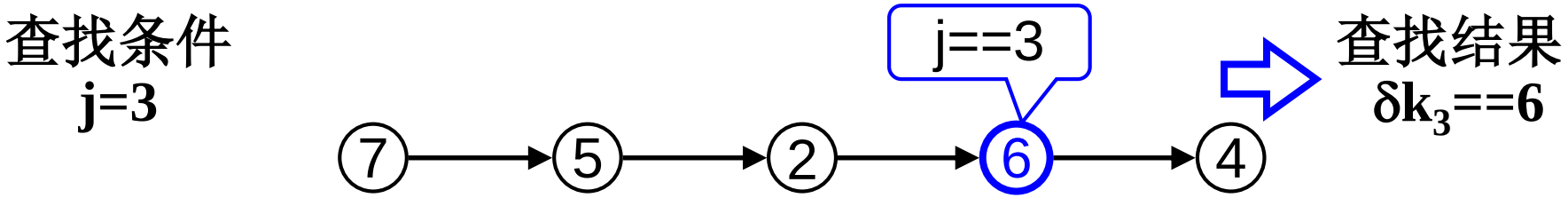
- **按结点值查找**

  根据结点k_j的值δk_j查找结点，打印序号j，使得δk_j==key。



- **按结点地址查找**

  根据结点k_j的序号j(αk)，查找结点k_j，打印结点k_j的值δk_j。



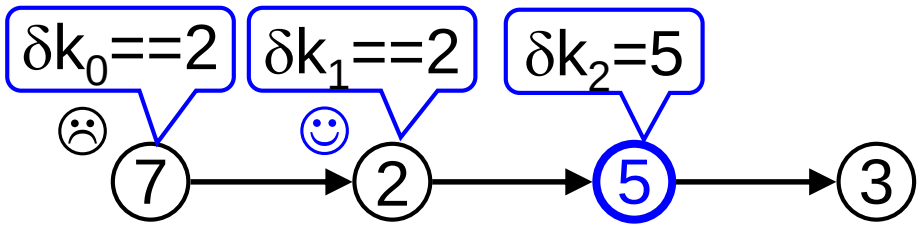
➢4.2.2 查找结点

➢查找结点的操作

- 按前件或后件查找

根据结点k_j的前件(或后件)k'的值δk'，打印序号j和结点k_j的值$δk_j$，使得δk'==key。

查找条件
前件的值
δk'=2



查找结果
$δk_2$==5

➢4.2.2 查找结点

➢顺序存储的查找结点

- 根据结点值查找结点的程序

```
#include <stdio.h>
#include <stdlib.h>
#define   M   1000
void searchNode(a, n, key)
short n, a[ ], key;
{
   short j;
   for (j=0; j<n; j++)
      if (a[j]==key)
      {
        printf("node(%hd) =", j);
         printf(" %hd\n", a[j]);
         return;
      }
   error();    /* 查找结点失败*/
}
```

```
main()
{
   short n, a[M];
   short  j, key;
   scanf("%hd", &n);
   if(n > M  || n < 1)
     error();  /* 结点数超界
   */
   for(j=0; j<n; j++)
      scanf("%hd", &a[j]);
   scanf("%hd", &key);
   searchNode(a, n, key);
}
```

- 算法复杂度

查找次数与n成正比，
最多为n次，
称该算法的时间复杂度为O(n)

➢4.2.2 查找结点

➢顺序存储的查找结点

●根据结点地址查找结点的程序

```
#include <stdio.h>
#include <stdlib.h>
#define   M   1000
void prtNode(a, n, j)
short n, a[ ], j;
{
   if(j<0 || j>n-1)
      error(); /* 结点地址超界
*/
   printf("node(%hd) =", j);
   printf(" %hd\n", a[j]);
}
```

```
main()
{
   short n, a[M];
   short  j;
   scanf("%hd", &n);
   if(n > M  || n < 1)
      error();  /* 结点数超界
   */
   for(j=0; j<n; j++)
      scanf("%hd", &a[j]);
   scanf("%hd", &j);
   prtNode(a, n, j);
}
```

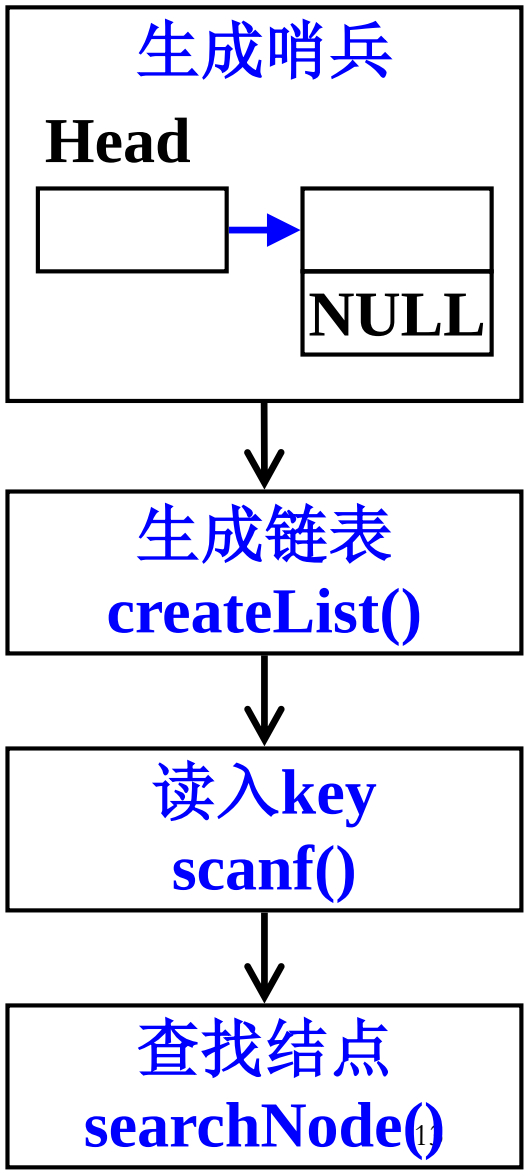
●算法复杂度

查找次数与n无关，
是一个常数C，
称该算法的时间复杂度为O(C)。

➢4.2.2 查找结点

➢链接存储的查找结点

```
main()
{
   NODE   *Head, *node;
   short key;
   /* 生成哨兵          */
   Head = (NODE *)(malloc(sizeof(NODE)));
   if(Head == NULL)
     error();  /* 内存分配失败    */
   Head->next = NULL;
   /* 生成链表          */
   createList(Head);
   /* 读入key           */
   scanf("%hd", &key);
   /* 查找结点          */
   if(node = searchNode(Head, key))
      printf("node = %hd\n", node->num);
   else
      error();    /* 查找结点失败*/
}
```

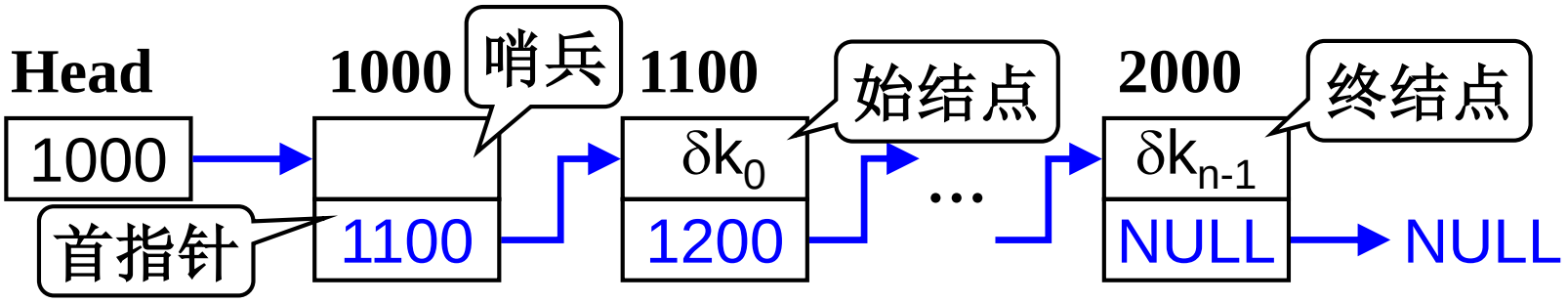


- 查找结点函数

```
NODE* searchNode(NODE *head, short key)
{
   NODE *node;
   for(node=head->next; node; node=node->next)
      if(node->num==key)
         return(node);
   return(NULL);      /* 查找结点失败      */
}
```

- 算法复杂度

  查找次数与n成正比，
  最多为n次，
  称该算法的时间复杂度为O(n)

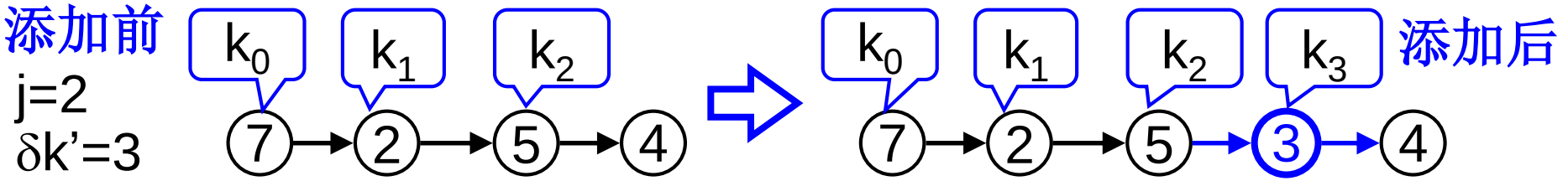


➢4.2.3 添加结点

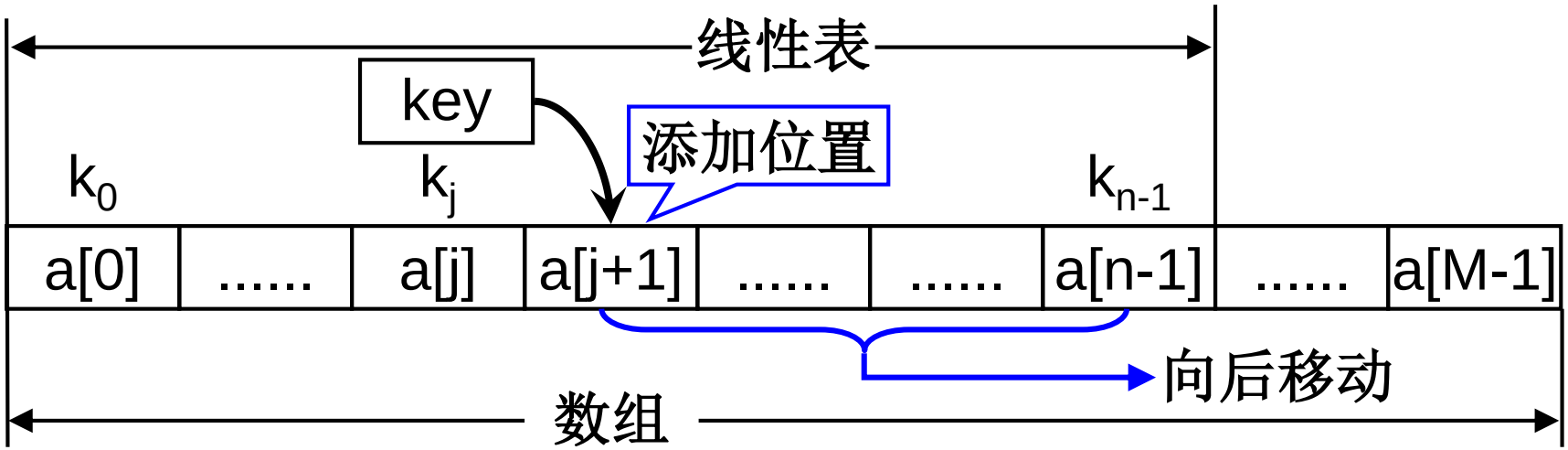
➢顺序存储的添加结点

●添加结点的操作

已知结点k_j的地址αk_j，在k_j后添加新结点k’，使得$\delta k'=key$。



将a[n-1]到a[j+1]向数组尾部移动，腾出添加新结点key的位置，再将新结点key插入。



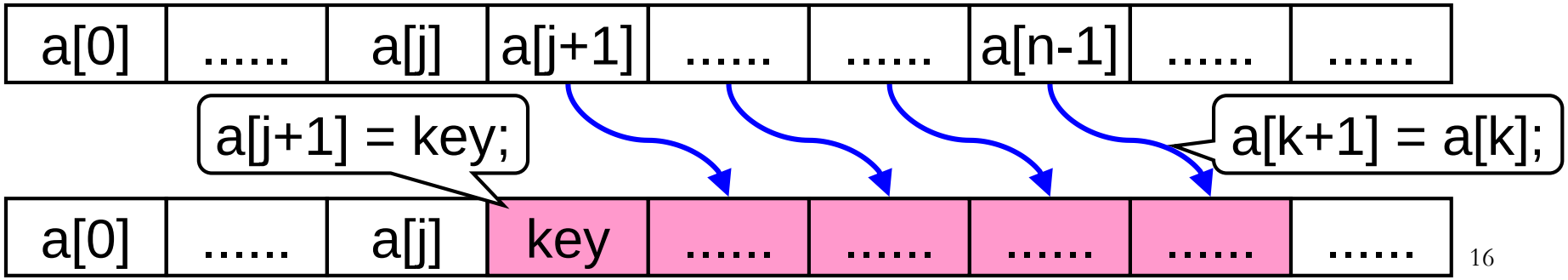
●顺序存储添加结点的程序

```
#include <stdio.h>
#include <stdlib.h>
#define   M   1000
void  addNode(a, n, j, key)
short a[], j, key, *n;
/* 添加结点后*n要加1 */
{
    short  k;
    if(j<-1 || j>(*n)-1)
        error();  /* 结点地址超界 */
    for(k=(*n)-1; k>j; k--)
        a[k+1] = a[k];   /*向后移动*/
    a[j+1] = key;
    (*n)++;
}
```

```
main()
{
    short  a[M], n, j, key;
    scanf("%hd", &n);
    if(n > M-1 || n < 1)
        error();  /* 结点数超界 */
    for(j=0; j<n; j++)
        scanf("%hd", &a[j]);
    scanf("%hd", &j);
    scanf("%hd", &key);
    addNode(a, &n, j, key);
}
```

- 添加结点的算法复杂度

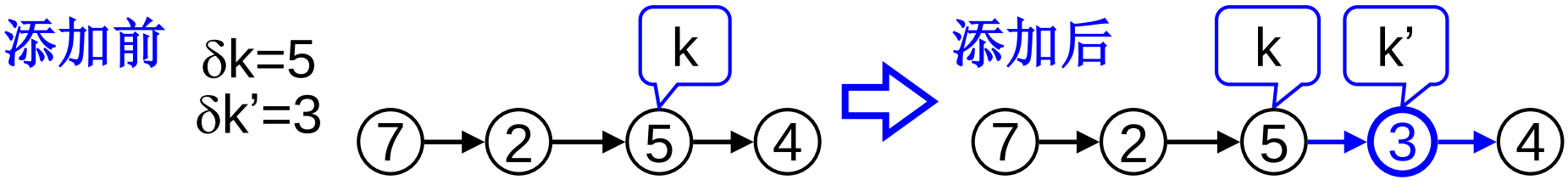
移动次数最多为n次，
算法复杂度为O(n)。



➢链接存储的添加结点

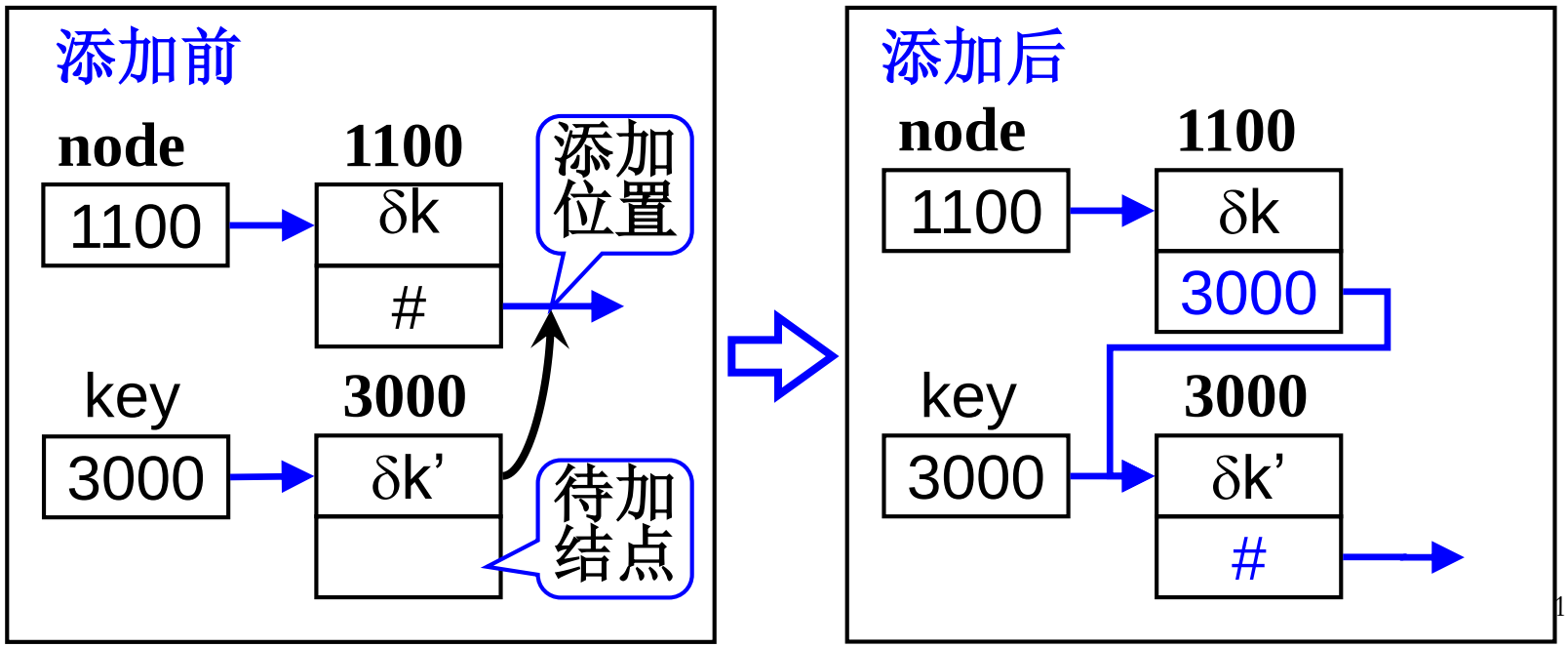
- 添加结点的操作

已知结点k的值δk，在k后添加新结点k’，使得δk’=key。



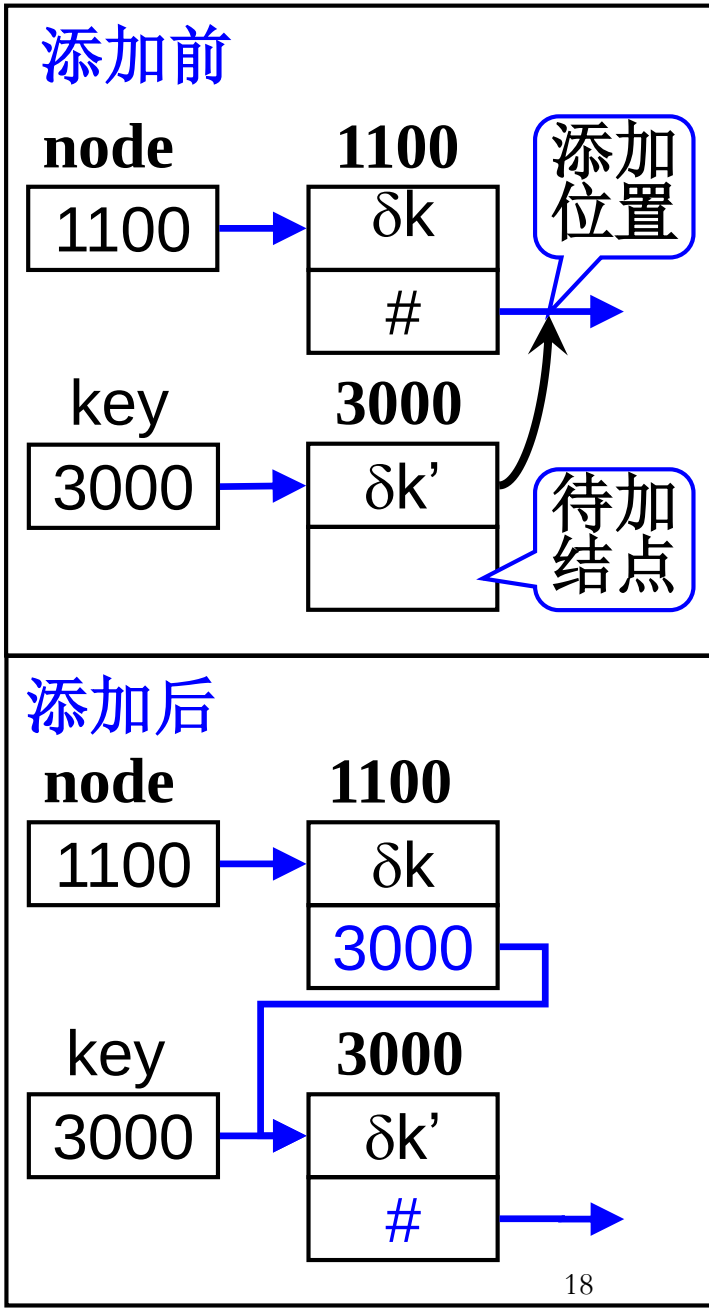
由key指向待添加结点k’。

先查找结点k，由node指向结点k，再在node后面插入结点k’。



●链接存储添加结点的函数

```
void addNode(NODE *node, short Key)
{
   NODE  *key;
   if (node==NULL)
      error();  /* 非法结点 */
   key=(NODE*)(malloc(sizeof(NODE)));
   if(key == NULL)
      error();  /* 内存分配失败  */
   key->num = Key;
   key->next = node->next;
   node->next = key;
}
```



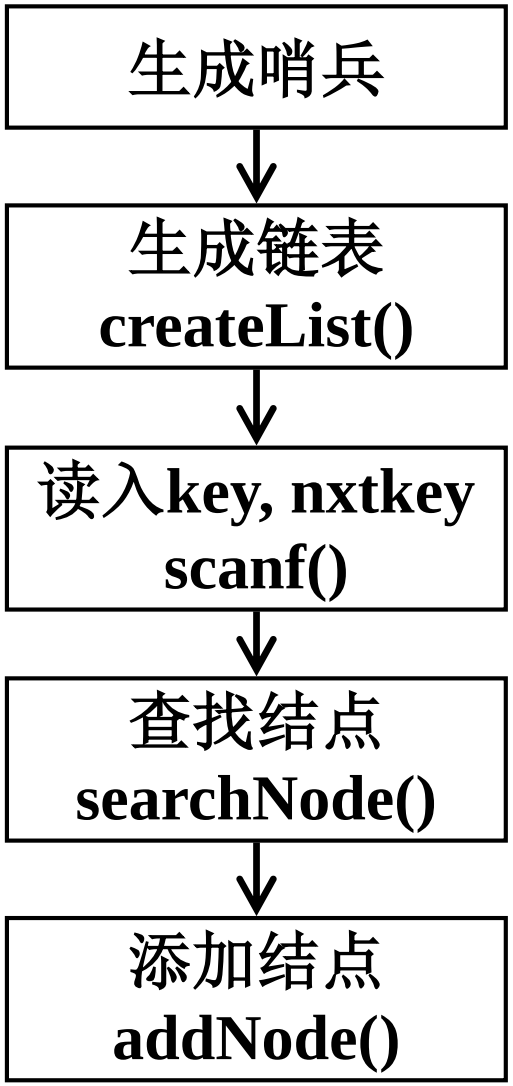
- 链接存储添加结点的算法复杂度
  添加结点的操作次数与n无关，是一个常数C，
  算法复杂度为O(C)。

● 链接存储添加结点的程序

```
#include <stdio.h>
#include <stdlib.h>
#define NODE struct node
NODE
{
   short num;
   NODE *next;
};
void createList(NODE *head)
{ ..... }  /* 略 */
NODE *searchNode(NODE *head, short key)
{ ..... }  /* 略 */
void addNode(NODE *node, short key)
{ ..... } /* 略 */
```

●链接存储添加结点的程序

```
main()
{
   short key, nxtkey;
   NODE   *Head, node;
   /* 生成哨兵        */
   Head = (NODE *)(malloc(sizeof(NODE)));
   if(Head == NULL)
      error();  /* 内存分配失败   */
   Head->next = NULL;
   /* 生成链表              */
   createList(Head);
   /* 读入key, nxtkey              */
   scanf("%hd%hd", &key, &nxtkey);
   /* 查找结点        */
   if(!(node = searchNode(Head, key))
      error();  /*查找结点失败*/
   /* 添加结点        */
   addNode(node, nxtkey);
}
```

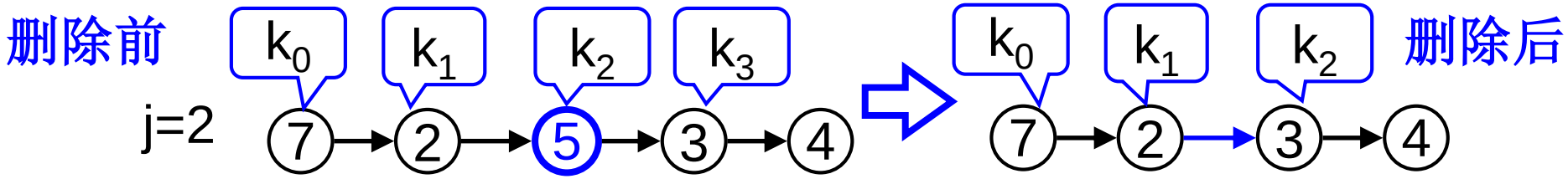


➢4.2.4 删除结点

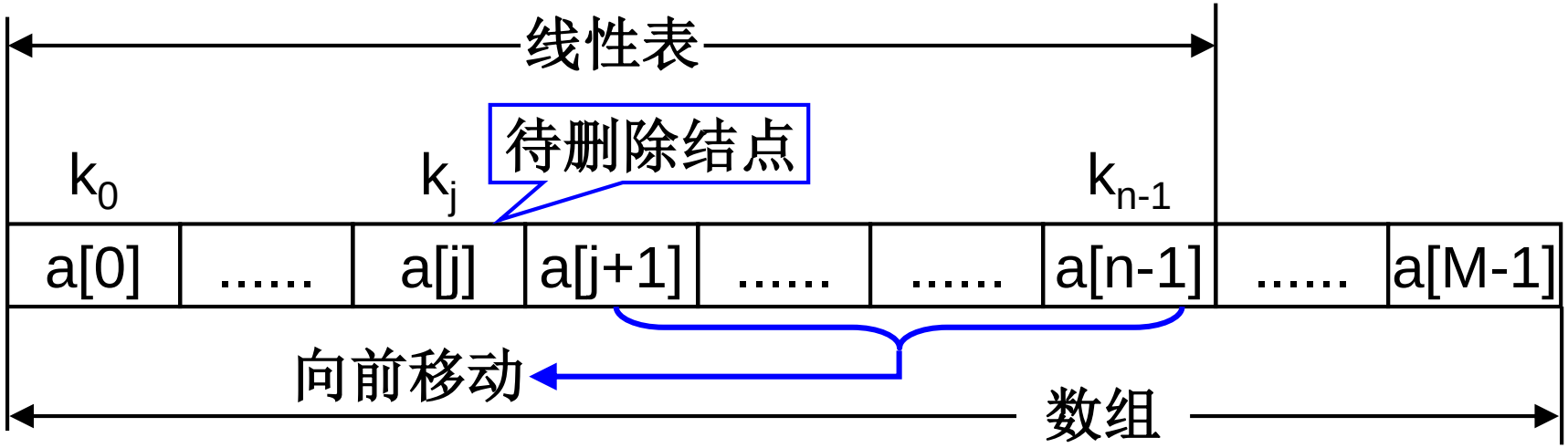
➢顺序存储的删除结点

- 删除结点的操作

已知结点k_j的地址αk_j，删除k_j。



将a[j+1]到a[n-1]向数组前部移动，待删结点a[j](k_j)被覆盖。



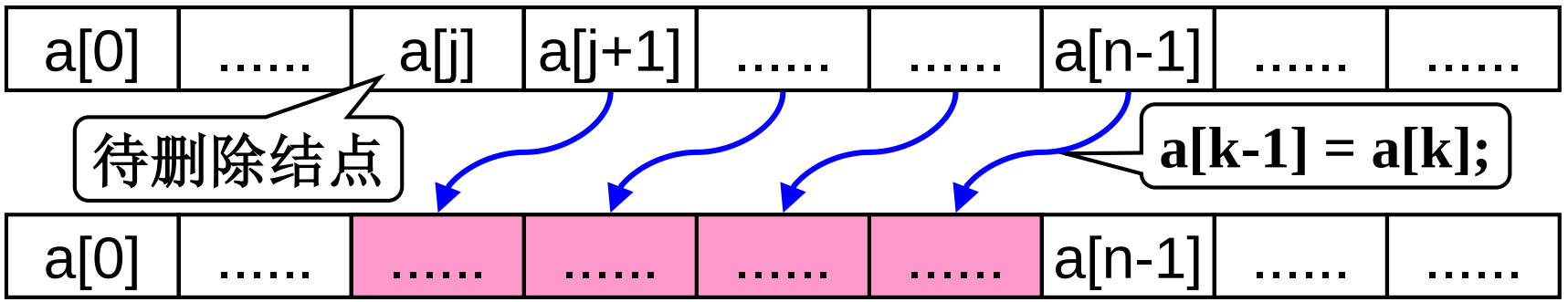
● 顺序存储删除结点的程序

```
#include <stdio.h>
#include <stdlib.h>
#define   M   1000
void  deleteNode(a, n, j)
short a[], j, *n;
/* 删除结点后*n要减1 */
{
   short  k;
   if(j<0 || j>(*n)-1)
      error();  /* 结点地址超界*/
   for(k=j+1; k<(*n); k++)
      a[k-1] = a[k]; /*向前移动*/
   (*n)--;
}
```

```
main()
{
   shaort  a[M], n, j, key;
   scanf("%hd", &n);
   if(n > M-1 || n < 1)
     error();  /* 结点数超界  */
   for(j=0; j<n; j++)
     scanf("%hd", &a[j]);
   scanf("%hd", &j);
   deleteNode(a, *n, j);
}
```

- 删除结点的算法复杂度

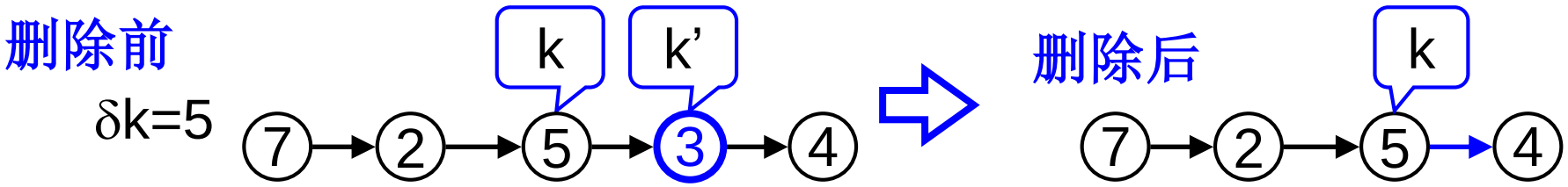
移动次数最多为n-1次，
算法复杂度为O(n)。



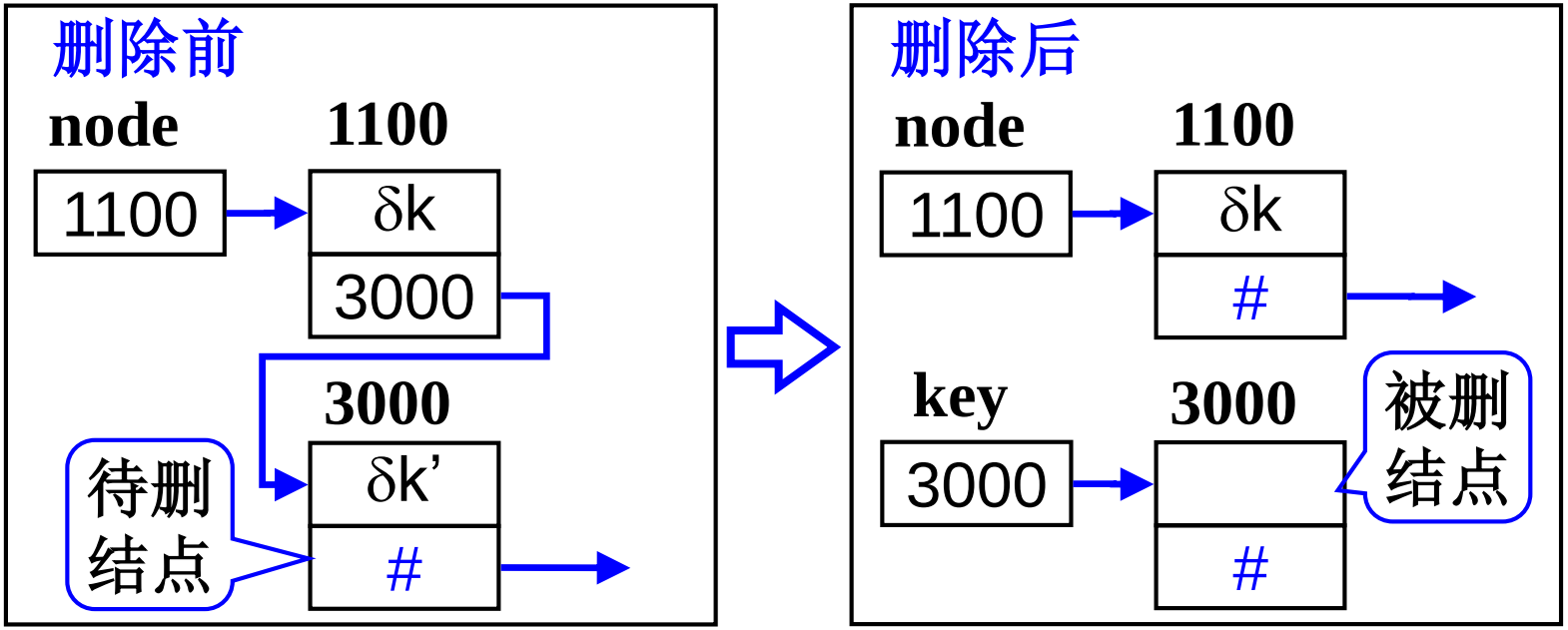
➢链接存储的删除结点

- 删除结点的操作

已知结点k的值δk，删除k的后件k’。

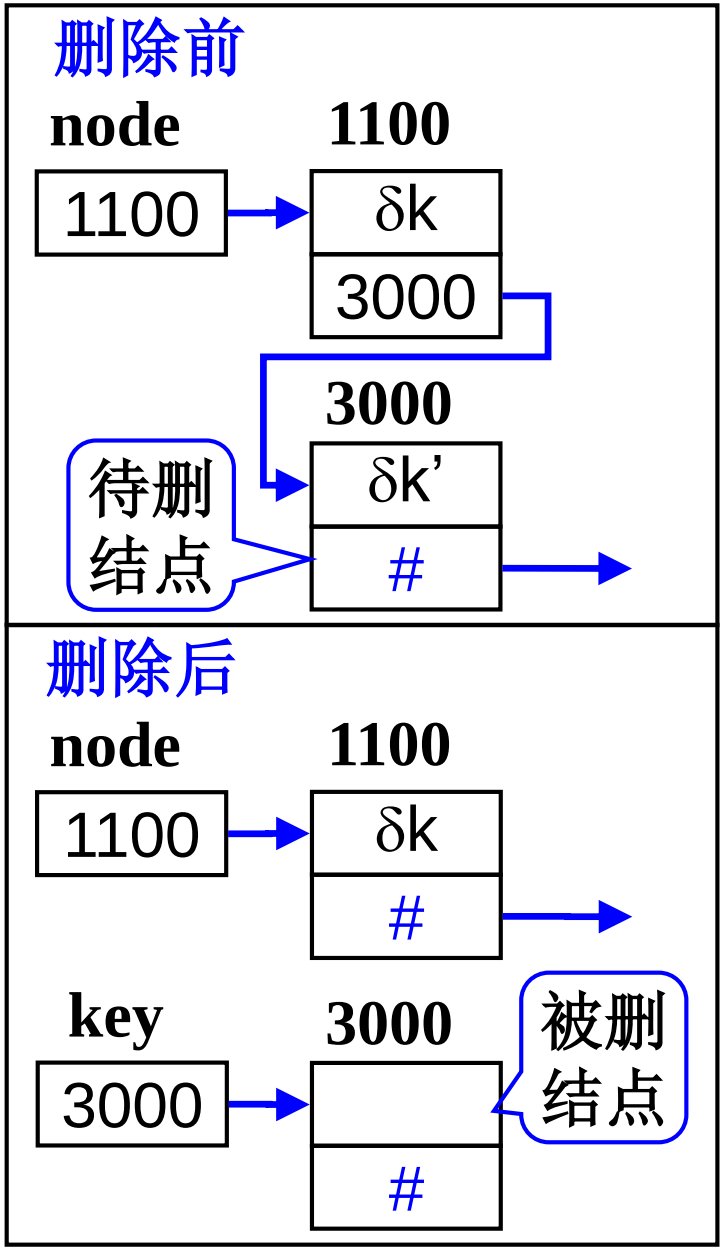


先查找结点k，由node指向结点k，由node->next指向node的后件k’(待删结点key)，再删除结点k’。



- 链接存储删除结点的函数

```
void deleteNode(NODE *node)
{
   NODE *key;
   if (node==NULL)
     error();      /* 非法结点      */
   key = node->next;
   node->next = key->next;
   /* 释放删除结点的存储单元*/
   free(key);
}
```



- 链接存储删除结点的算法复杂度

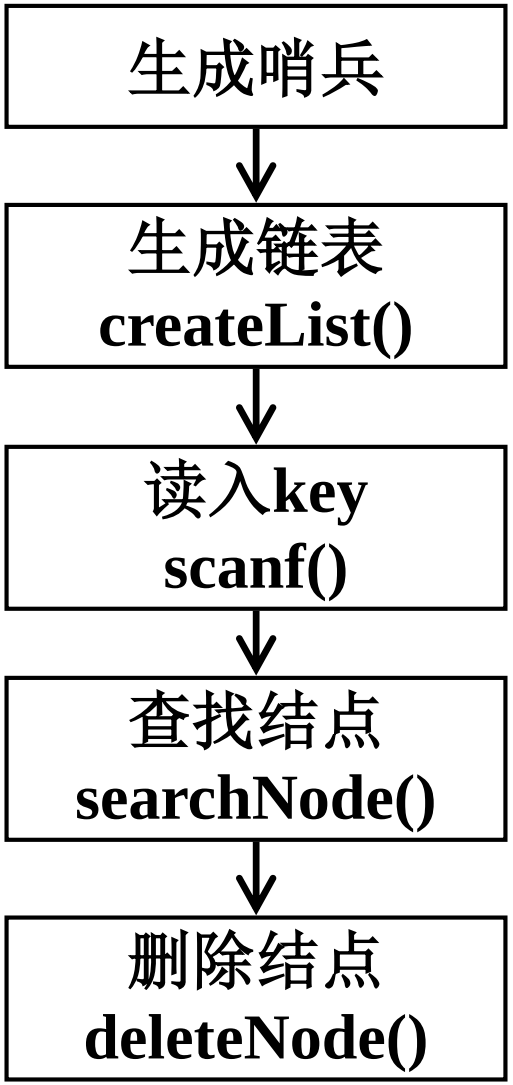
  删除结点的操作次数与n无关，
  是一个常数C，算法复杂度为O(C)。

● 链接存储删除结点的程序

```
#include <stdio.h>
#include <stdlib.h>
#define NODE struct node
NODE
{
   short num;
   NODE *next;
};
void createList(NODE *head)
{ ..... }  /* 略 */
NODE *searchNode(NODE *head, short key)
{ ..... }  /* 略 */
```

● 链接存储删除结点的程序

```
main()
{
   short key;
   NODE   *Head, node;
   /* 生成哨兵          */
   Head = (NODE *)(malloc(sizeof(NODE)));
   if(Head == NULL)
     error();  /* 内存分配失败   */
   Head->next = NULL;
   /* 生成链表          */
   createList(Head);
   /* 读入key          */
   scanf("%hd", &key);
   /* 查找结点          */
   if(!(node = searchNode(Head, key))
     error();  /* 查找结点失败   */
   /* 删除结点后件      */
   deleteNode(node);
}
```



➢4.2.5 排序

➢排序问题

以升序为例讨论排序问题。

- 定义

已知一个结点序列$K = \{k_0, k_1, k_2, \ldots, k_{n-1}\}$，
经过排序得到$K' = \{k'_0, k'_1, k'_2, \ldots, k'_{n-1}\}$，
使得K和K'中结点数值相同，而$\delta k_i' \le \delta k'_{i+1} (i=0, 1, \ldots, n-2)$，
称新的结点序列K'为已排序的结点序列。

- 需要掌握的五种排序算法

(1) 插入排序法
(2) 选择排序法
(3) 冒泡排序法
(4) 快速排序法
(5) 合并排序法

以下仅讨论插入排序算法。

➢插入排序算法

- 插入排序算法描述

设已知结点序列为$K = \{k_0, k_1, k_2, \ldots, k_{n-1}\}$，

将已排序的结点序列记为S，未排序的结点序列记为U。

初始：令$S = \{k_0\}$，$U = K - \{k_0\} = \{k_1, k_2, \ldots, k_{n-1}\}$

循环执行的插入步骤：

从i=1到n-1

令$key = k_i \mid k_i \in U$，$U = U - \{k_i\}$，

在S中插入key，使得$S = S \cup \{key\}$为已排序。

若$U=\varphi$，插入排序结束。

➢插入排序算法

- 排序过程示例

已知结点序列为{6, 5, 9, 4, 8}，

初始：令S = {k_0} = {6}，U = {k_1, …, k_4} = {5, 9, 4, 8}。

循环插入的过程示例：

Step 1: i=1，插入5， 得S={5, 6}， U={9, 4, 8}；

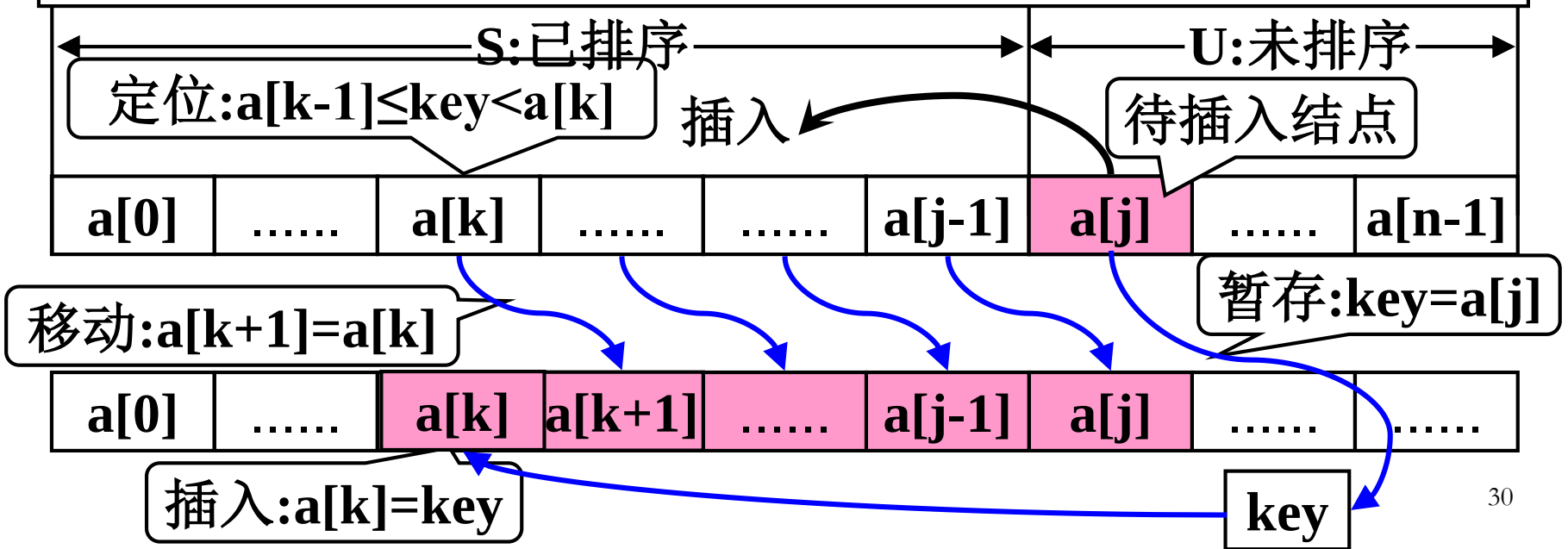
Step 2: i=2，插入9，得S={5, 6, 9}， U={4, 8}；

Step 3: i=3，插入4，得S={4, 5, 6, 9}， U={8}；

Step 4: i=4，插入8， 得S={4, 5, 6, 8, 9}， U=φ，结束。

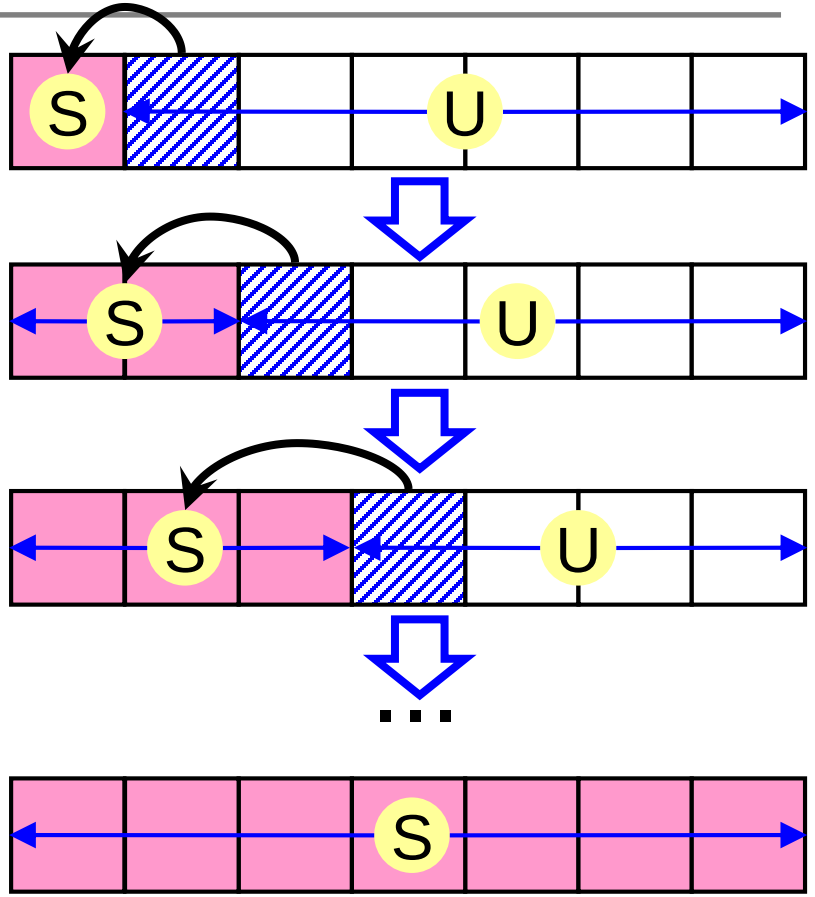
➢顺序存储的插入排序函数 掌握得还不够熟练

```
void insert(short *a, short n)
{
   short key, j, k;
   for(j=1; j<n; j++)                  /* 初始   */
   {  /* 暂存: key=a[j] ，  定位: key<a[k]     */
      for(key=a[j], k=j-1; key<a[k] && k>=0; k--)
         a[k+1] = a[k];                /* 移动   */
      a[++k] =key;                     /* 插入   */
   }
}
```



● 顺序存储的插入排序程序

```
#include <stdio.h>
#include <stdlib.h>
#define   M   1000
void insert(short *a, short n)
{ ..... }/* 略 */
main()
{
   short  a[M], n, j;
   scanf("%hd", &n);
   if(n > M-1 || n < 1)
      error();     /*结点数超界 */
   for(j=0; j<n; j++)
      scanf("%hd", &a[j]);
   insert(a, n);          /* 排序   */
   for(j=0; j<n; j++) /*打印    */
      printf("%hd\n", a[j]);
}
```



- 顺序存储插入排序的算法复杂度

考虑n很大时，最多移动次数

$= 1 + \ldots + (n-2) + (n-1)$

$= (n-1) * n / 2 \approx n^2 / 2 \approx n^2$

所以，算法复杂度为$O(n^2)$。

➢链接存储的插入排序

- 链接存储的插入排序算法

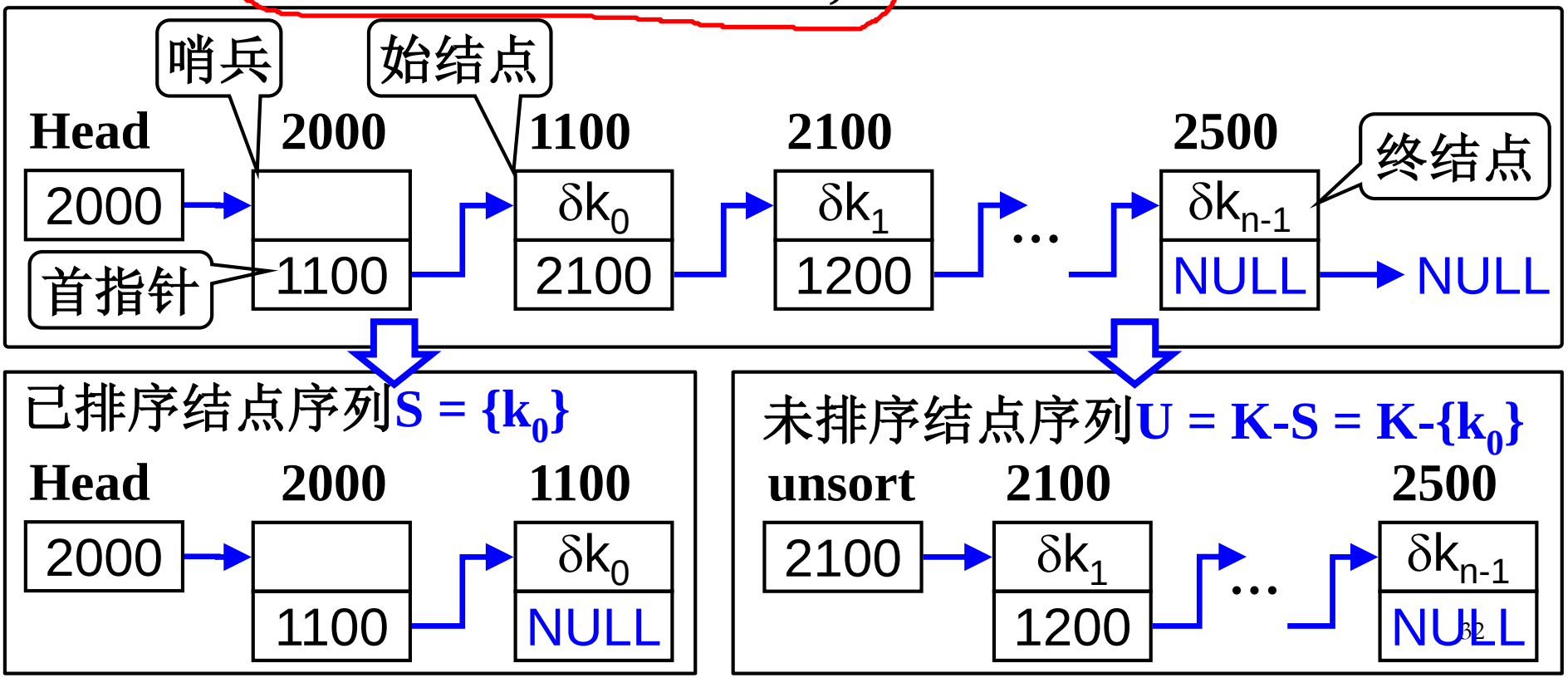
1) 初始设置

由head->next指向S = {k_0}，

由指针unsort指向未排序的结点序列U = K - S = K - {k_0}。

程序语句为：

```
unsort = head->next->next;
head->next->next = NULL;
```



2) 从U中逐个取出结点在S中排序的过程

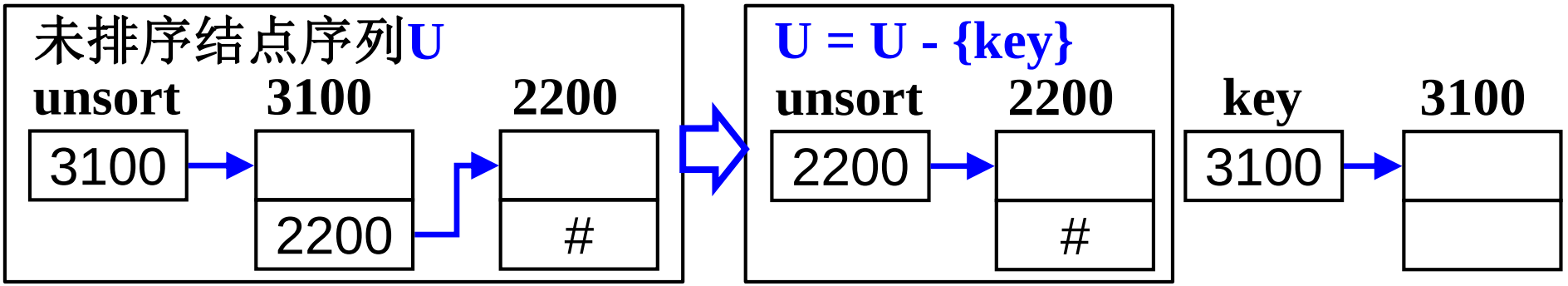
while(unsort)　　/*while(unsort!=NULL)*/

从U中取出一个结点，由指针key指向，即在U中删除结点：

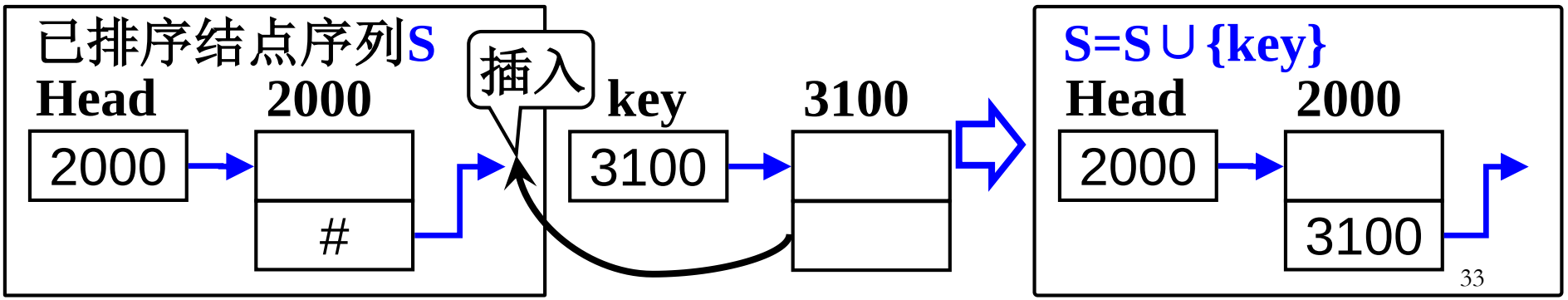
$key = k_i \mid k_i \in U$，U = U - {key}

程序语句为：

```
key = unsort;
unsort = key->next;
```



将结点key插入S：S=S∪{key}，并且保持S为已排序。



●链接存储的插入排序函数

- 链接存储插入排序算法的时间复杂度

  从U取出结点为n-1次(while循环)，定位插入点的次数最多为n-2(for循环)，

  所以算法复杂度为$O(n^2)$。

```
void insort(head)
NODE *head;
{
    NODE *unsort, *node, *key;
    unsort = head->next->next;          /* 初始：S = {k0}                */
    head->next->next = NULL;            /* 初始：U = K - {k0}            */
    while(unsort)                       /* U非空，进行循环               */
    {
        key = unsort;                   /* U = U - {key}                 */
        unsort = key->next;
        for(node=head; node->next; node=node->next)
            if(key->number <= node->next->number)
                break;                  /* 定位插入点                    */
        key->next = node->next;         /* 将key插入S，保持S有序         */
        node->next = key;
    }
}
```

●链接存储的插入排序程序

```
#include <stdio.h>
#include <stdlib.h>
#define NODE struct node
NODE
{
   short num;
   NODE *next;
};
void createList(NODE *head)
{ ..... }  /* 略 */
void insert(NODE *head)
{ ..... }  /* 略 */
```

●链接存储的插入排序程序

```
main()
{
   NODE   *Head, node;
   /* 生成哨兵    */
   Head = (NODE *)(malloc(sizeof(NODE)));
   if(Head == NULL)
     error();  /* 内存分配失败   */
   Head->next = NULL;
   /* 生成链表          */
   createList(Head);
   /* 排序      */
   insert(Head);
   /* 打印      */
   for(node=Head->next; node; node=node->next)
      printf("%hd\n", node->num);
}
```

➢ 4.2.6 存储方式与算法复杂度

基本操作	操作函数	顺序存储	链接存储
查找结点	searchNode()	O(n)	O(n)
查找结点	prtNode()	O(C)	O(n)
添加结点	addNode()	O(n)	O(C)
删除结点	deleteNode()	O(n)	O(C)
插入排序	insert()	$O(n^2)$	$O(n^2)$

编程时要根据实际情况来决定采用哪一种存储方式：

- 例如，需要频繁进行添加结点和删除结点操作的应用，采用链接存储较好。而只需要根据结点地址打印结点，则采用顺序存储较好。
- 另外，还应该综合n的可知情况决定采用数组还是链表。

作　　业

- 习题　　4-1，4-38
- 上机题　elearning上完成上机作业：
  4-39.1(顺序表操作)，
  4-40.1(链接表操作)，
  4-39.2或者4-39.3任选1题，
  4-40.2或者4-40.3任选1题

➢上机说明

在windows下完成C语言程序的编写：vc

在linux下完成C语言程序的编写

- 文本编辑器：vi
- C语言编译工具：gcc

  $ gcc -c main.c　　“编译”生成main.o

  $ gcc -o try main.o　　“链接”
- 使用make维护程序的编译及链接，例如编写Makefile内容如下：

```
TARGET = linelist
SRC = ListMain.c ListSub.c
OBJ = ListMain.o ListSub.o
$(TARGET) : $(OBJ)
        gcc -o $@ $(OBJ)
.c.o:
        gcc -c $<
clean:
        rm $(OBJ)
```