

计算机考研系列书课包

玩转数据结构



主讲人

刘财政

高阶应用

本讲内容

考点一：线性表的高阶应用 (10.1)

考点二：二叉树的应用 (10.2)

考点三：图的高阶应用 (10.3)

考点四：其他高阶应用 (10.4)

算法解法:

- 暴力解: 枚举方法
- 可行解:
- 最优解: 缘分方法

算法解题套路：

- 方法体，非完整程序，核心代码
- 结构体定义
- 算法思想和算法步骤
- 算法实现
- 算法复杂度分析：时间复杂度和空间复杂度

得分神助攻：

- 图文结合
- 代码注释
- 命名规范
- 格式严谨

在算法中一般可直接使用的库函数

1.max:求最大值,时间复杂度为 $O(n)$

2.min:求最小值,时间复杂度为 $O(n)$

3.length:求长度,时间复杂度为 $O(n)$

4.pop():出栈

5.push():入栈

刘财政

启航教育

考点一：

线性表的高阶应用

刘财政

启航教育

刘财政

启航教育

核心回顾

考点一：线性表的高阶应用

```
typedef int DataType;    /*定义线性表的数据类型，假设为int型*/

typedef struct Node {
    DataType data;        /*数据域，保存结点的值 */
    struct Node *next;    /*指针域*/
} LNode, *LinkList;      /*结点的类型 */
```

data	next
------	------

```
typedef struct Node LNode
```

```
typedef struct Node *LinkList
```



考点一：线性表的高阶应用



如何向内存申请一个结点？

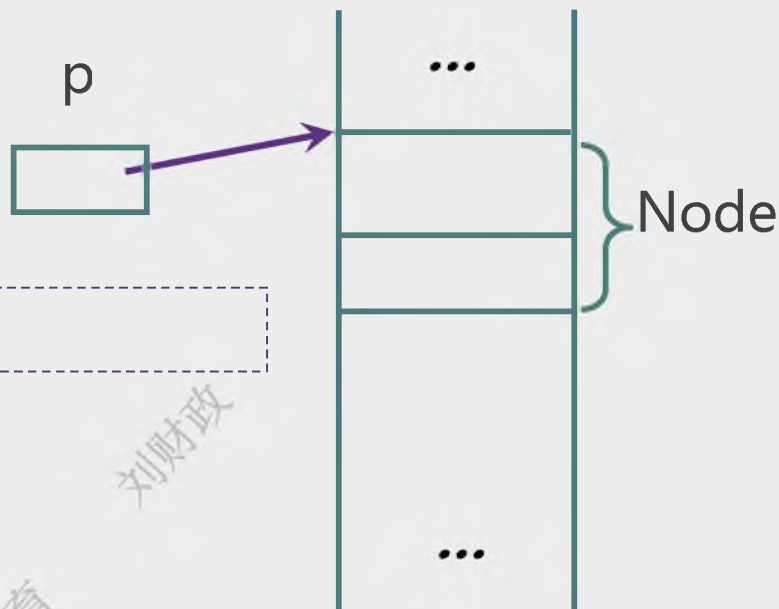
```
LNode *p = NULL;
```

```
p = (LNode *)malloc(sizeof(LNode));
```



如何将一个结点释放给内存？

```
free(p);
```



考点一：线性表的高阶应用



设指针 p 指向某个Node类型的结点

该结点用 $*p$ 来表示, $*p$ 为结点变量。将“指针 p 所指结点”简称为“结点 p ”



如何引用结点 p 的数据域（指针域）？

$*p.data$

$*p.next$



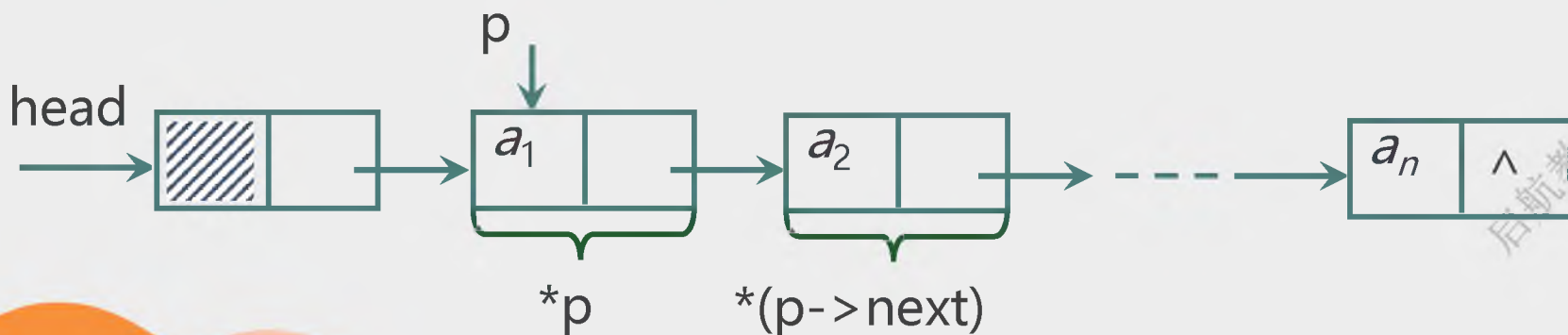
$p \rightarrow data$



$p \rightarrow next$



如何表示结点 p 的下一个结点？



考点一：线性表的高阶应用




初始化一个单链表

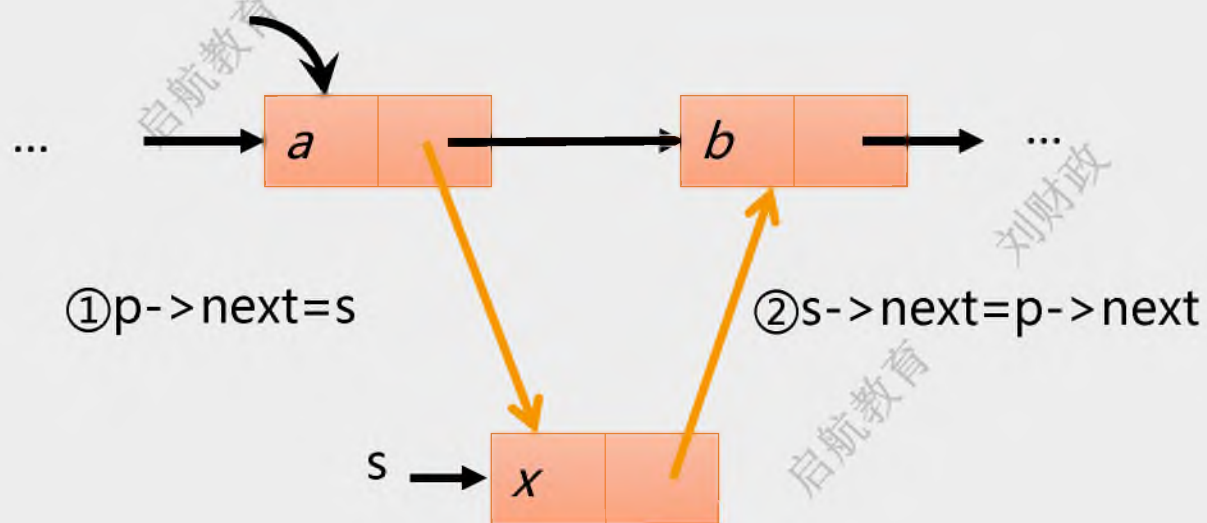


```
LNode *InitList()  
{  
    LNode *head = (LNode *)malloc(sizeof(LNode));  
    head->next = NULL;  
    return head;  
}
```

建立一个空的单链表，即创建一个头结点。

考点一：线性表的高阶应用


 插入一个节点：实现结点 a_{i-1} 、 x 和 a_i 之间逻辑关系的变化

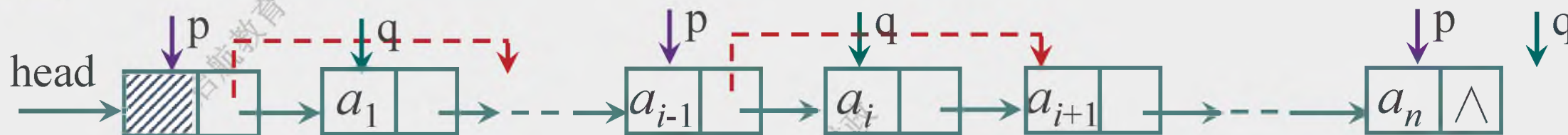


 操作描述：

```
s = (LNode *)malloc(sizeof(LNode));
s->data = x;
s->next = p->next;
p->next = s;
```

考点一：线性表的高阶应用

 删除一个结点：实现结点 a_{i-1} 、 a_i 和 a_{i+1} 之间逻辑关系的变化



操作描述：

```
q=p->next; x=q->data;
p->next=q->next; free(q);
```

表尾的特殊情况：

虽然被删结点不存在，但其前驱结点却存在！

考点一：线性表的高阶应用



后移操作:指针向后移动

$p = p \rightarrow \text{next}$, 将指针P向后移动一个位置, 指向直接后继结点

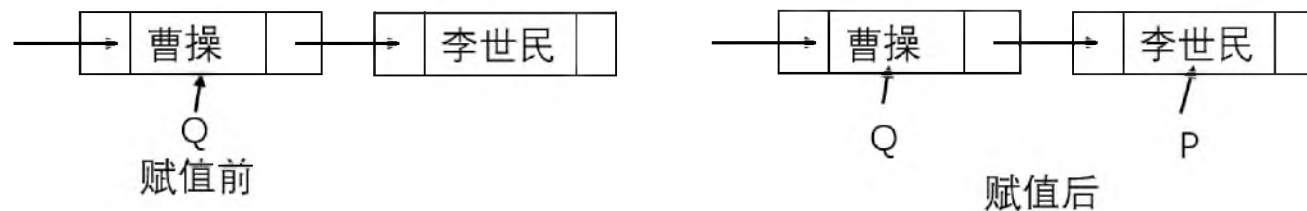


考点一：线性表的高阶应用



赋值操作:赋值为直接后继结点

$P = Q \rightarrow \text{next}$, 将P赋值为Q的直接后继结点, 此时P是Q的直接后继,
Q是P的直接前驱



考点一：线性表的高阶应用

操作接口：LNode * LinkList(DataType a[], int n)

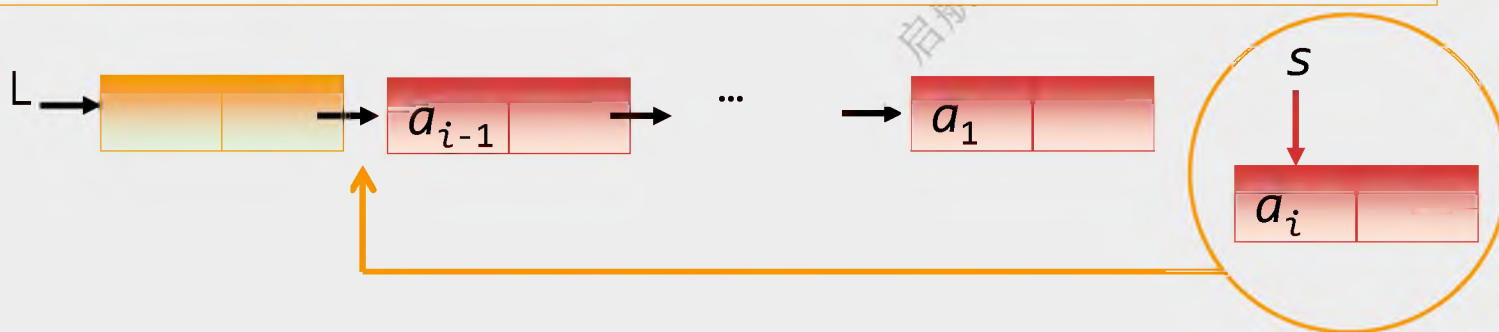
数组 a

35	12	24	33	42
----	----	----	----	----



头插法：插在头结点的后面

- 从一个空表开始，创建一个头结点。
- 依次读取字符数组 a 中的元素，生成新结点
- 将新结点插入到当前链表的表头上，直到结束为止。



注意：链表的结点顺序与逻辑次序相反。

考点一：线性表的高阶应用

```
LNode *CreatList(DataType a[ ], int n)
{
    LNode *s = NULL;
    LNode *head = (LNode *)malloc(sizeof(LNode));
    head->next = NULL;
    for (int i = 0; i < n; i++)
    {
        s = (LNode *)malloc(sizeof(LNode));
        s->data = a[i];
        s->next = head->next; head->next = s;
    }
    return head;
}
```

考点一：线性表的高阶应用

操作接口： `LNode * LinkList(DataType a[], int n)`

数组 a

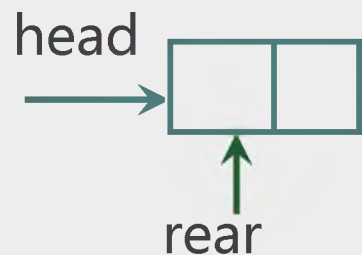
35	12	24	33	42
----	----	----	----	----



尾插法：插在尾结点的后面

📍 为方便在尾结点后面插入结点，设指针 `rear` 指向尾结点

初始化



```
head = (LNode *)malloc(sizeof(LNode));
rear = head;
```

刘财政
启航教育

考点一：线性表的高阶应用

```
LNode *CreatList(DataType a[ ], int n)
{
    LNode *s = NULL, *rear = NULL;
    LNode *head = (LNode *)malloc(sizeof(LNode));
    rear = head;
    for (int i = 0; i < n; i++)
    {
        s = (LNode *)malloc(sizeof(LNode));
        s->data = a[i];
        rear->next = s; rear = s;
    }
    rear->next = NULL; /*单链表建立完毕，将终端结点的指针域置空*/
    return head;
}
```

高阶应用

考点一：线性表的高阶应用

线性表的高阶应用

- 0.1 哈希策略
- 0.2 单链表或者数组逆置策略
- 0.3 单链表或者数组旋转策略
- 0.4 双指针策略

考点一：线性表的高阶应用

0.1 哈希套路

考点一：线性表的高阶应用

将记录的存储位置与它的关键字之间建立一个对应关系

常用的方法有：

直接定址法： $\text{hash}(\text{key}) = \text{keyhash}(\text{key}) = a * \text{key} + b$

取余法： $\text{hash}(\text{key}) = \text{key} \% p$

空间换时间的一种思路

考点一：线性表的高阶应用

1、使用算法复杂度为 $O(N)$ 实现排序，其中 N 是元素个数，且元素大小不超过 n 。

例如：给定1,4,5,6,4,3,4,2,1,5,6。

0	1	2	3	4	5	6
0	2	1	1	3	2	2

考点一：线性表的高阶应用

```
void SortByN(int a[], int length){
```

```
    if (a == NULL || length < 0) return;
```

```
    //重新申请一个数组
```

```
    const int N = 200; int b[N]; int i, j; int key ;
```

```
    //把新申请的数组全部初始化为0
```

```
    for (i = 0; i < N; ++i) {b[i] = 0;}
```

```
    //遍历a数组
```

```
    for (i = 0; i < length; ++i) {
```

```
        key = a[i];
```

```
        ++b[key];
```

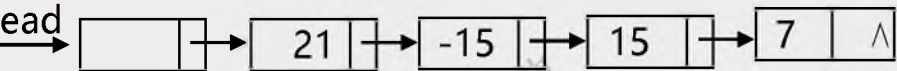
```
    }
```

```
}
```


考点一：线性表的高阶应用

2、用单链表保存 m 个整数，结点的结构为：[data][link]，且 $|data| \leq n$ (n 为正整数)。现要求设计一个时间复杂度尽可能高效的算法，对于链表中 data 的绝对值相等的结点，仅保留第一次出现的结点而删除其余绝对值相等的结点。例如

，若给定的单链表 head 如下：



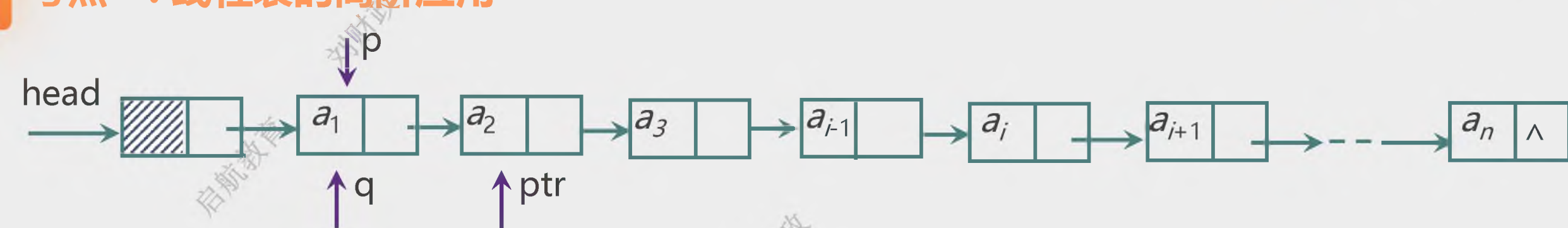
则删除结点后的 head 为：



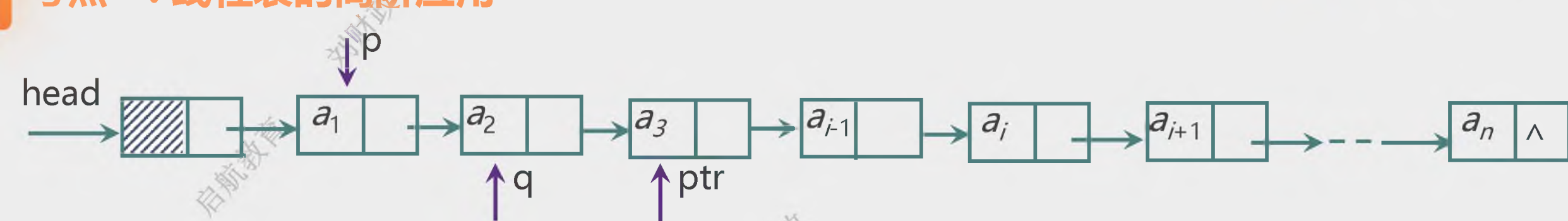
要求：

- 1) 给出算法的基本设计思想。
- 2) 使用 C 或 C++ 语言，给出单链表结点的数据类型定义。
- 3) 根据设计思想，采用 C 或 C++ 语言描述算法，关键之处给出注释。
- 4) 说明你所设计算法的时间复杂度和空间复杂度。

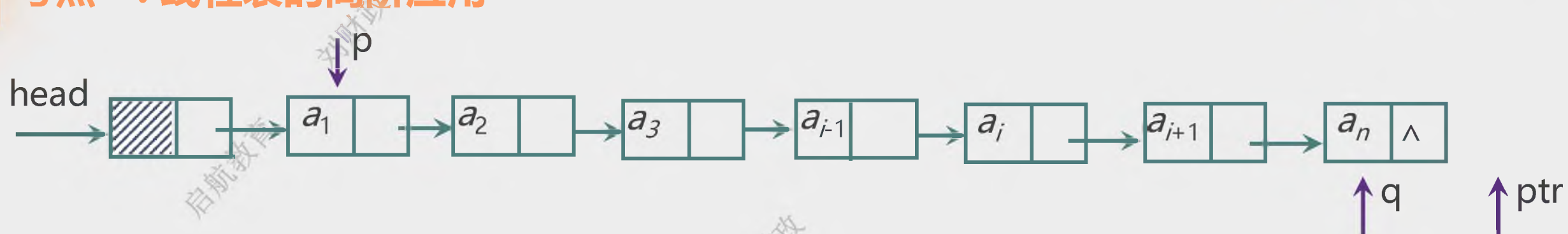
考点一：线性表的高阶应用



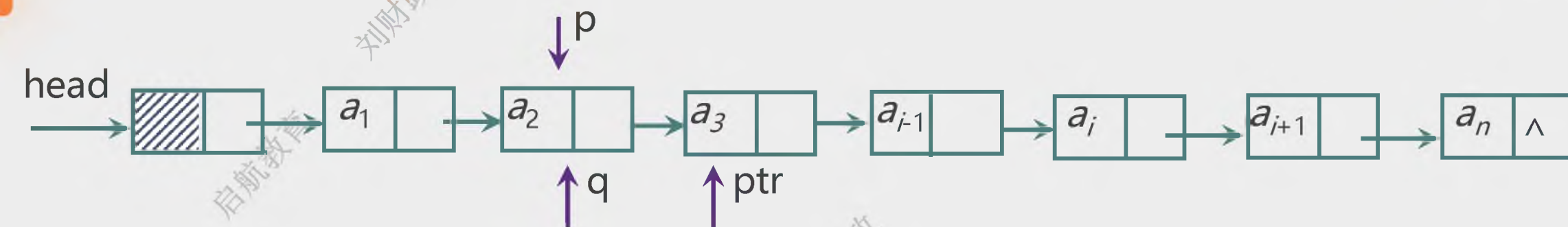
考点一：线性表的高阶应用



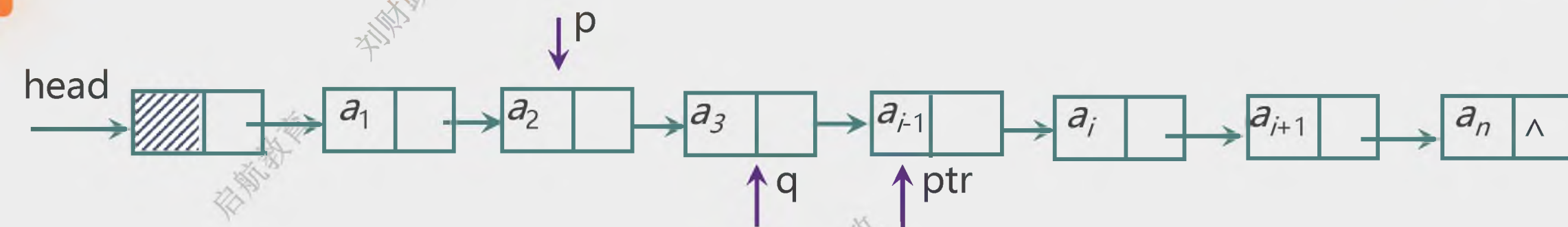
考点一：线性表的高阶应用



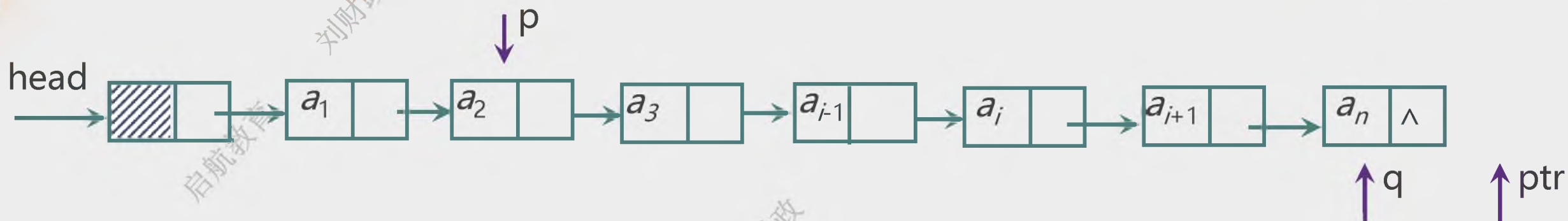
考点一：线性表的高阶应用



考点一：线性表的高阶应用



考点一：线性表的高阶应用



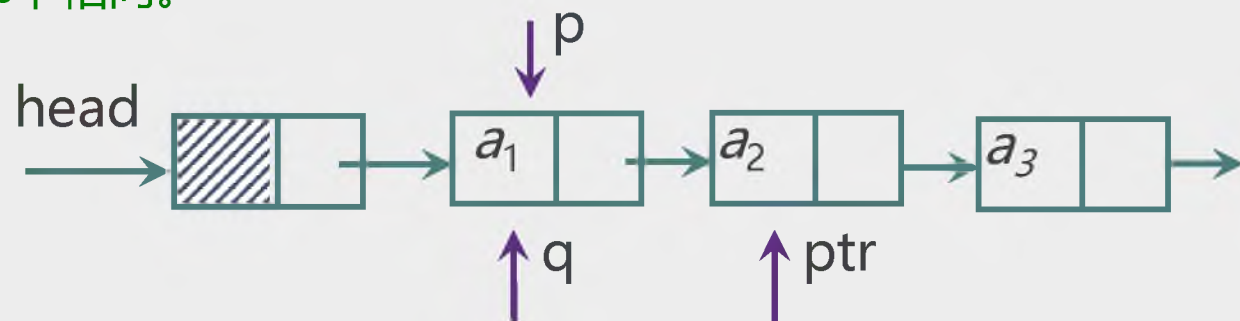
考点一：线性表的高阶应用

结构体定义

```
typedef struct LNode{  
    ElemType data;  
    struct LNode *link;  
}LNode, *LinkedList;
```

// 删除单链表中所有值重复的结点，使得所有结点的值都不相同。

```
void Delete_Node_Value(LNode *L) {
    // 删除以L为结点的单链表中所有值相同的结点
    LNode *p = L->link, *q, *ptr;
    while (p != NULL) { // 检查链表中的所有结点
        q = p; ptr = p->link;
        // 检查结点p的所有后继结点ptr
        while (ptr != NULL) {
            if (|ptr->data| == |p->data|)
                {q->link = ptr->link; free(ptr); ptr = q->link;}
            else {q = ptr; ptr = ptr->link;}
        }
        p = p->link;
    }
}
```



// 删除单链表中所有值重复的结点，使得所有结点的值都不相同。

```
void Delete_Node_Value(LNode *L) {
```

// 删除以L为结点的单链表中所有值相同的结点

```
LNode *p = L->link, *q, *ptr;
```

```
while (p != NULL) { // 检查链表中的所有结点
```

```
    q = p; ptr = p->link;
```

//检查结点p的所有后继结点ptr

```
    while (ptr != NULL) {
```

```
        if (|ptr->data| == |p->data|)
```

```
            {q->link = ptr->link; free(ptr); ptr = q->link;}
```

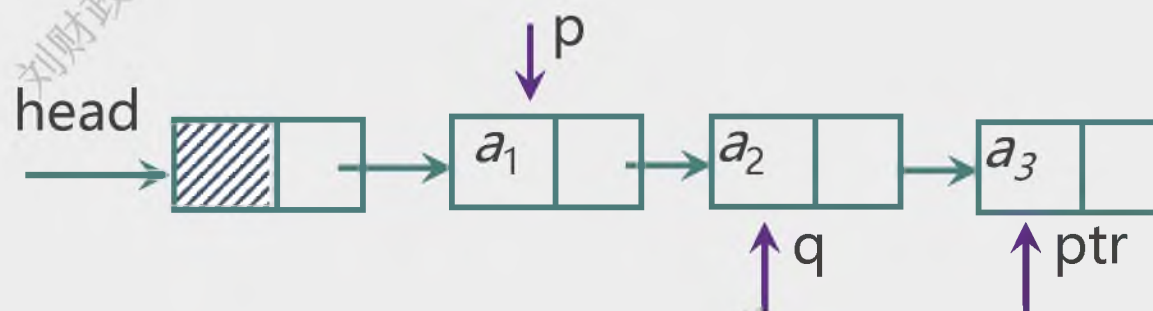
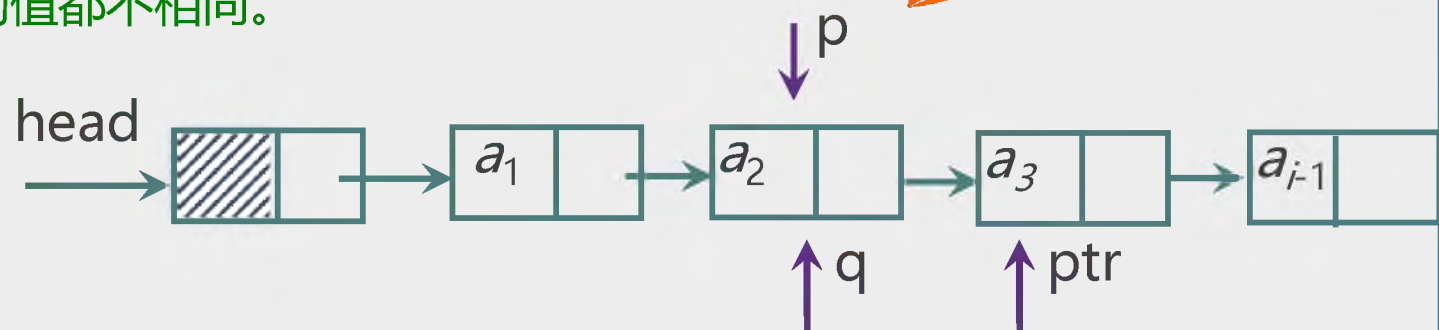
```
        else {q = ptr ; ptr = ptr->link;}
```

```
    }
```

```
    p = p->link;
```

```
}
```

```
}
```



// 删除单链表中所有值重复的结点，使得所有结点的值都不相同。

```
void Delete_Node_Value(LNode *L) {
```

// 删除以L为结点的单链表中所有值相同的结点

```
LNode *p = L->link, *q, *ptr;
```

```
while (p != NULL) { // 检查链表中的所有结点
```

```
    q = p; ptr = p->link;
```

//检查结点p的所有后继结点ptr

```
    while (ptr != NULL) {
```

```
        if (|ptr->data| == |p->data|)
```

```
            {q->link = ptr->link; free(ptr); ptr = q->link;}
```

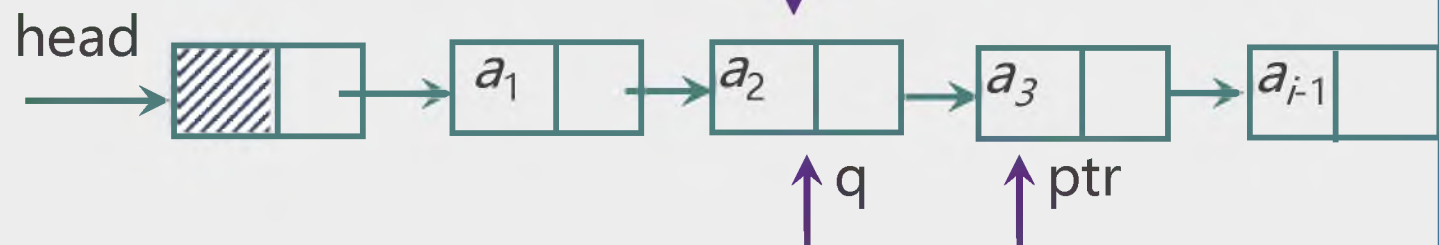
```
        else {q = ptr ; ptr = ptr->link;}
```

```
    }
```

```
    p = p->link;
```

```
}
```

```
}
```



$O(n^2)$

考点一：线性表的高阶应用

(1) 算法的基本设计思想 算法的核心思想是用空间换时间。使用辅助数组记录链表中已出现的数值，从而只需对链表进行一趟扫描。因为 $|data| \leq n$ ，故辅助数组 q 的大小为 $n+1$ ，各元素的初值均为 0。依次扫描链表中的各结点，同时检查 $q[|data|]$ 的值，如果为 0，则保留该结点，并令 $q[|data|]=1$ ；否则，将该结点从链表中删除。

(2) 使用 C 语言描述的单链表结点的数据类型定义

```
typedef struct node {  
    int data;  
    struct node *link;  
}NODE;  
typedef NODE *PNODE;
```


考点一：线性表的高阶应用

```
void func(LNode *h, int n) {
    LNode *p = h, r;
    int *q, m;
    q = (int *)malloc(sizeof(int) * (n + 1)); // 申请 n+1 个位置的辅助空间
    int i;
    for (i = 0; i < n + 1; i++) *(q + i) = 0; // 数组元素初值置 0
    while (p->link != NULL) {
        m = p->link > data > 0 ? p->link > data : -p->link > data;
        if (*(q + m) == 0) { // 判断该结点的 data 是否已出现过
            *(q + m) = 1; // 首次出现
            p = p->link; // 保留
        } else { // 重复出现
            r = p->link; // 删除
            p->link = r->link;
            free(r);
        }
    }
    free(q);
}
```

考点一：线性表的高阶应用

```
void func(LNode *h, int n) {  
    LNode *p = h, r;  
    int *q, m;  
    q = (int *)malloc(sizeof(int) * (n + 1)); // 申请 n+1 个位置的辅助空间  
    int i;  
    for (i = 0; i < n + 1; i++) *(q + i) = 0; // 数组元素初值置 0  
    while (p->next != NULL) {  
        m = p->next->data > 0 ? p->next->data : -p->next->data;  
        if (*(q + m) == 0) { // 判断该结点的 data 是否已出现过  
            *(q + m) = 1; // 首次出现  
            p = p->next; // 保留  
        } else { // 重复出现  
            r = p->next; // 删除  
            p->next = r->next;  
            free(r);  
        }  
    }  
    free(q);  
}
```

考点一：线性表的高阶应用

```
void func(LNode *h, int n) {  
    LNode *p = h, r;  
    int *q, m;  
    q = (int *)malloc(sizeof(int) * (n + 1)); //申请 n+1 个位置的辅助空间  
    int i;  
    for (i = 0; i < n + 1; i++) *(q + i) = 0; //数组元素初值置 0  
    while (p->next != NULL) {  
        m = p->next->data > 0 ? p->next->data : -p->next->data;  
        if (*(q + m) == 0) { //判断该结点的 data 是否已出现过  
            *(q + m) = 1; //首次出现  
            p = p->next; //保留  
        } else { //重复出现  
            r = p->next; //删除  
            p->next = r->next;  
            free(r);  
        }  
    }  
    free(q);  
}
```

考点一：线性表的高阶应用

3、已知一个整数序列 $A = (a_0, a_1, \dots, a_{n-1})$ ，其中 $0 \leq a_i \leq n$ ($0 \leq i < n$)。若存在 $a_{p_1} = a_{p_2} = \dots = a_{p_m} = x$ 且 $m > n/2$ ($0 \leq p_k < n, 1 \leq k \leq m$)，则称 x 为 A 的**主元素**。例如 $A = (0, 5, 5, 3, 5, 7, 5, 5)$ ，则 5 为主元素；又如 $A = (0, 5, 5, 3, 5, 1, 5, 7)$ ，则 A 中没有主元素。假设 A 中的 n 个元素保存在一个一维数组中，请设计一个**尽可能高效的算法**，找出 A 的主元素。若存在主元素，则输出该元素；否则输出 -1。要求：

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想，采用 C 或 C++ 语言描述算法，关键之处给出注释。
- (3) 说明你所设计算法的时间复杂度和空间复杂度。

考点一：线性表的高阶应用

(1) 算法思想：算法的基本设计思想 算法的核心思想是用空间换时间。使用辅助数组记录数组中已出现的数值，从而只需对数组进行一趟扫描。因为 $|data| \leq n$ ，故辅助数组 q 的大小为 $n+1$ ，各元素的初值均为 0。依次扫描数组中的各结点，同时检查 $q[|data|]$ 的值，如果为 0，并令 $q[|data|]=1$ ；否则， $q[|data|] + 1$ 。

考点一：线性表的高阶应用

例： $A=(0, 5, 5, 3, 5, 7, 5, 5)$ ，则 5 为主元素；又如 $A=(0, 5, 5, 3, 5, 1, 5, 7)$ ，则 A 中没有主元素。

0	1	2	3	4	5	6	7
1	0	0	1	0	5	0	1

0	1	2	3	4	5	6	7
1	1	0	1	0	4	0	1

考点一：线性表的高阶应用

(3) 算法实现：

```
int findMajorData ( int a[], int n ) {  
    int i;    int *q,m;  
    q=(int *)malloc(sizeof(int)*(n+1)); //申请 n+1 个位置的辅助空间  
    for(i=0;i<n+1;i++) *(q+i)=0; //数组元素初值置 0  
    for(i = 0; i < n; i++){  
        m = a[i];  
        if(*(q+m)==0) { *(q+m) = 1; } //首次出现  
        else{ *(q+m) += 1 } //重复出现  
    }  
}
```

考点一：线性表的高阶应用

```
for(i=0;i<n+1;i++){  
    if(*(q+i)>n/2) return i; //确认候选主元素  
    return -1; //不存在主元素  
}
```

(3) 时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$

考点一：线性表的高阶应用

4、给定一个含 n ($n \geq 1$) 个整数的数组，请设计一个在时间上尽可能高效的算法，找出数组中未出现的最小正整数。例如，数组 $\{-5, 3, 2, 3\}$ 中未出现的最小正整数是1,数组 $\{1, 2, 3\}$ 中未出现的最小正整数是4.要求：

- (1) 给出算法的基本设计思想
- (2) 根据设计思想，采用C或C++语言描述算法，关键之处给出注释。
- (3) 说明你所设计的算法的时间复杂度和空间复杂度

考点一：线性表的高阶应用

(1) 给出算法的基本设计思想：设要查找的数组中未出现的最小正整数为 K 。 K 取值范围只能是 $[1, n+1]$ 。采用类似基数排序的思想，分配一个数组 $B[n]$ ，用来标记 A 中是否出现了 $1 \sim n$ 之间的正整数。从左至右依次扫描数组元素 $A[i]$ 并标记数组 B 。若 $A[i]$ 是负数、零或是大于 n ，则忽略该值；否则，根据计数排序的思想将 $B[A[i]-1]$ 置为1。标记完毕，遍历数组 B ，查找第一个值为0的元素，其下标+1即为目标元素 K ；找不到0时， $K = n+1$ 。

考点一：线性表的高阶应用

```
int findMissMin(int A[],int n){  
    int i,*B;//标记数组  
    B = (int *)malloc(sizeof(int)*n); //分配空间  
    memset(B,0,sizeof(int)*n); //赋初值为0  
    for(i = 0;i < n;i++){  
        if(A[i] > 0 && A[i] <= n) //若A[i]的值介于1~n，则标记数组B  
            B[A[i]-1] = 1;  
    }  
    for(i = 0;i<n;i++) { //扫描计数数组，找到目标值K  
        if(B[i] == 0) break;  
    }  
    return i + 1; //返回结果  
}
```

考点一：线性表的高阶应用

哈希策略解题总结：

- (1) 找题眼： $0 \leq a_i \leq n$
- (2) 开数组，并置0
- (3) 定下标：计数或者判有无
- (4) 后处理

考点一：线性表的高阶应用

0.2 单链表或者数组逆置

考点一：线性表的高阶应用

1、给定一个单链表，求出一个快速算法，实现单链表逆序

```
LinkedList reverseList(LinkedList head) {
```

```
    LinkedList tmp = NULL;
```

```
    LinkedList cur = NULL;
```

```
    LinkedList node = NULL;
```

```
    if (head == NULL) return NULL;
```

```
    node = head;
```

```
    cur = head->next;
```

```
    while (cur != NULL){
```

```
        tmp = cur->next;
```

```
        cur->next = node;
```

```
        node = cur;
```

```
        cur = tmp;
```

```
    }
```

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

```
    return node;
```

```
}
```

考点一：线性表的高阶应用

2、给定一个数组，求出一个算法，实现数组逆置。

启航教育

刘财政

启航教育



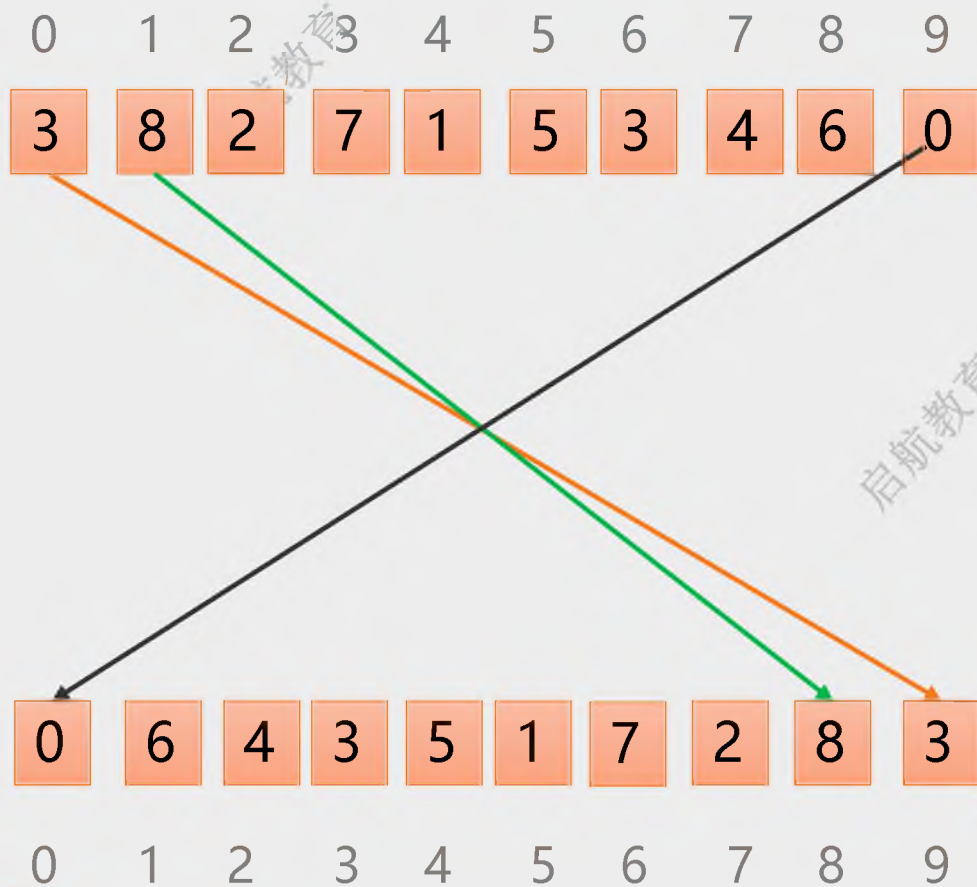
启航教育

刘财政

启航教育

考点一：线性表的高阶应用

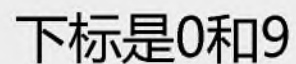
方法一：开辟数组法



时间复杂度是 $O(n)$

空间复杂度是 $O(n)$

方法二：对位交换法



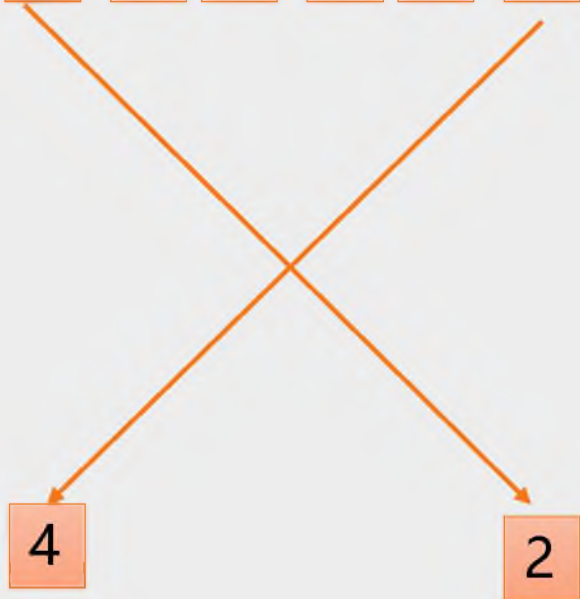
方法二：对位交换法



考点一：线性表的高阶应用

方法二：对位交换法

0	1	2	3	4	5	6	7	8	9
3	8	2	7	1	5	3	4	6	0

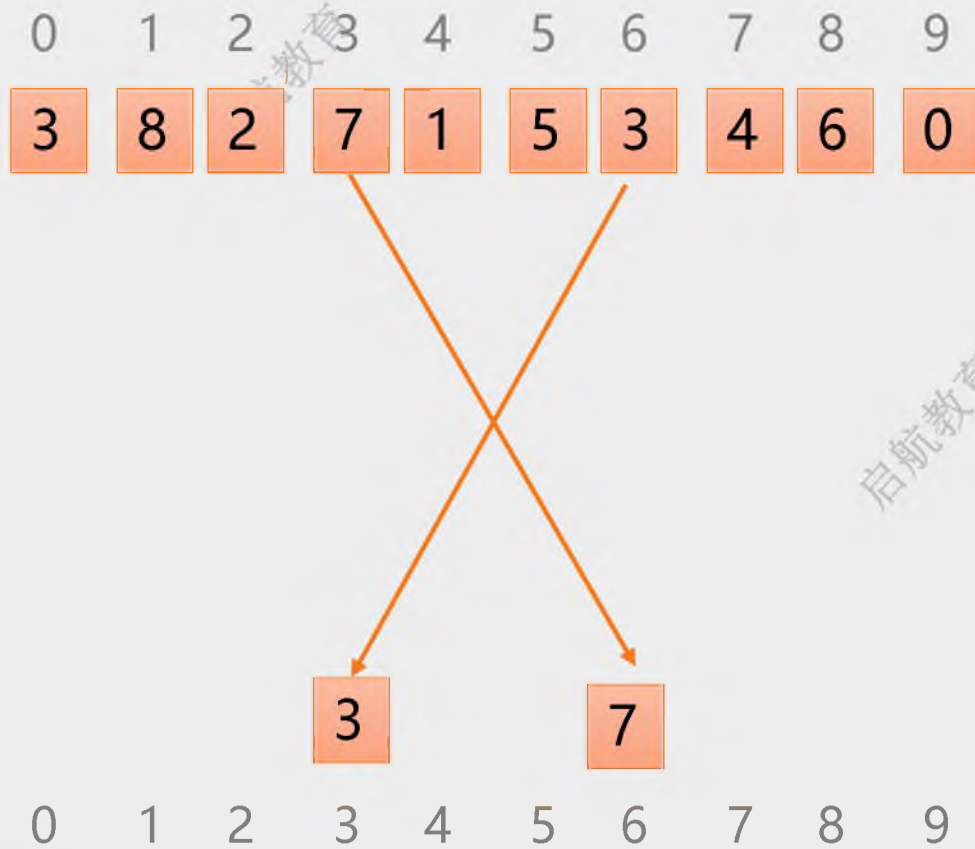


下标是2和7

0	1	2	3	4	5	6	7	8	9
		4					2		

考点一：线性表的高阶应用

方法二：对位交换法



下标是3和6

考点一：线性表的高阶应用

方法二：对位交换法

0	1	2	3	4	5	6	7	8	9
3	8	2	7	1	5	3	4	6	0

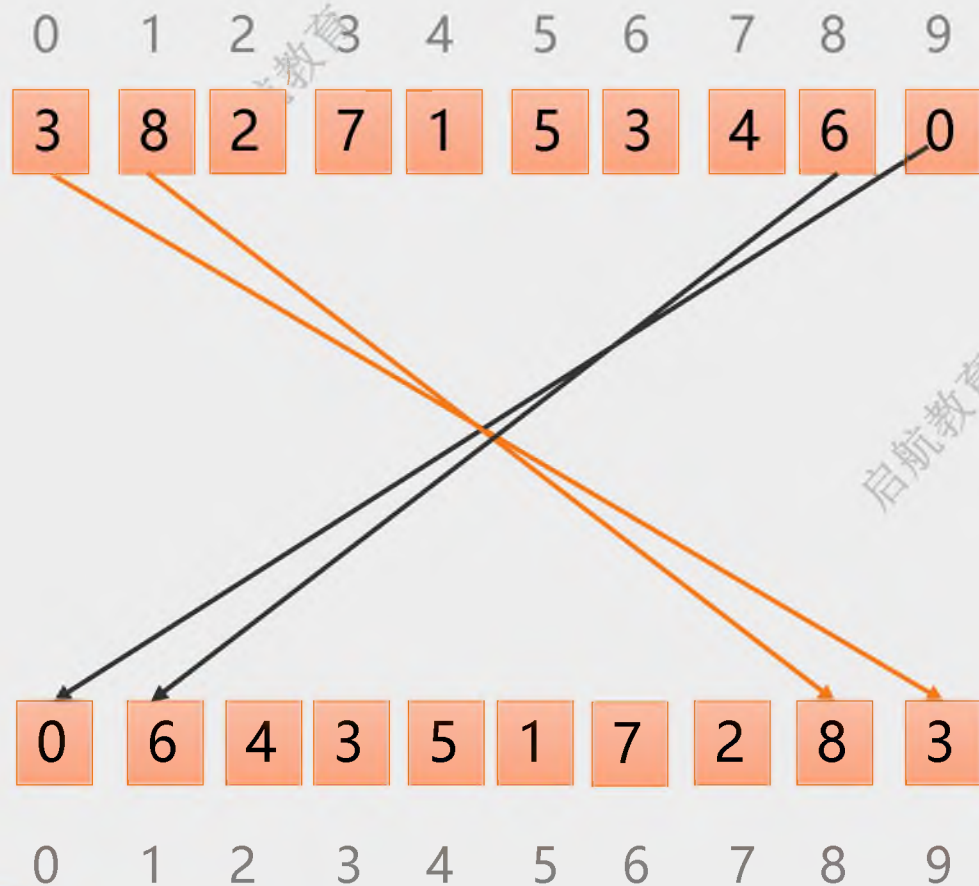


下标是4和5

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

考点一：线性表的高阶应用

方法二：对位交换法



下标是0和9

下标是1和8

下标是2和7

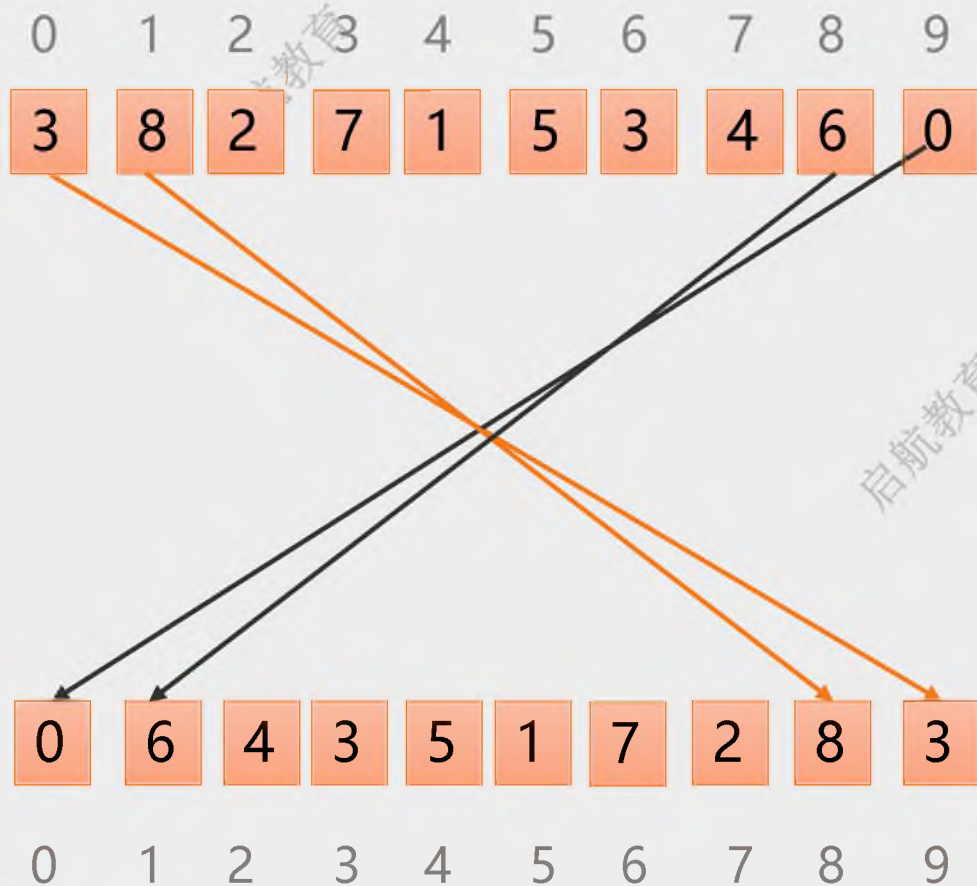
下标是3和6

下标是4和5

下标和是9 (length-1)

考点一：线性表的高阶应用

方法二：对位交换法



对位交换：

下标是0和9

下标是1和8

下标是2和7

下标是3和6

下标是4和5

时间复杂度是 $O(n)$

空间复杂度是 $O(1)$

考点一：线性表的高阶应用

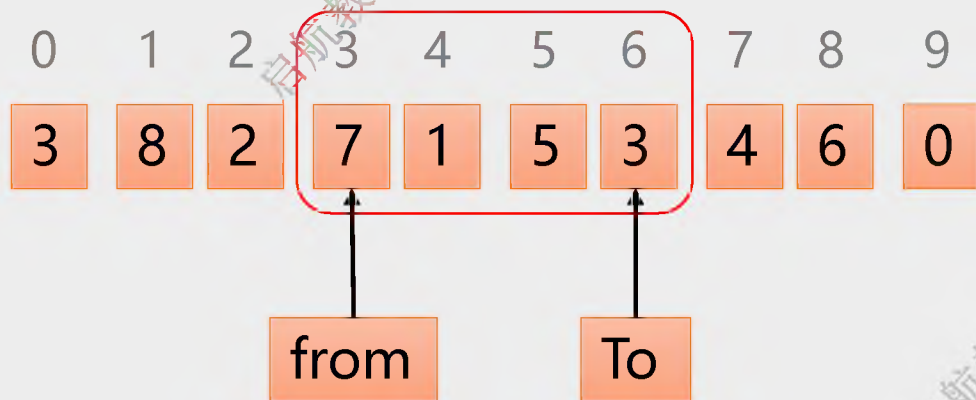
// 试写一算法，实现顺序表的就地逆置，

// 即利用原表的存储空间将线性表 $(a_1, a_2, a_3, \dots, a_n)$ 逆置为 $(a_n, a_{n-1}, \dots, a_2, a_1)$ 。

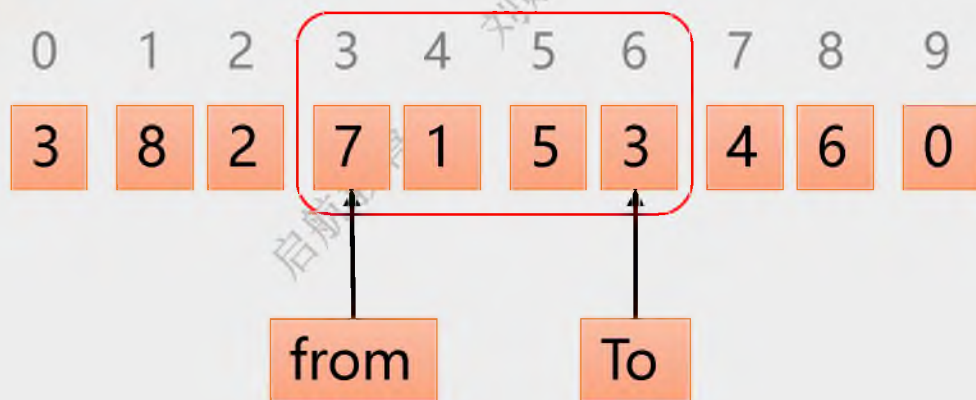
```
int ListOppose(SeqList* L){  
    int i;   ElemType x;  
    // 只需要遍历原表的一半就可以实现数据元素位置的交换  
    for (i = 0; i < L->length / 2; i ++ ) {  
        x = L->elem[i];  
        L->elem[i] = L->elem[L->length - i - 1];  
        L->elem[L->length - i - 1] = x;  
    }  
    return 1;  
}
```

考点一：线性表的高阶应用

【改进，部分逆置】试写一算法，实现顺序表的就地逆置，但是要逆置这个数组那一段的开始下标from，要逆置那一段的结束下标to。



考点一：线性表的高阶应用



需要交换值的元素下表是：

from + i VS to - i

$i = (\text{to} - \text{from} + 1) / 2$

考点一：线性表的高阶应用

/* 这个函数传入的参数为要逆置的数组 要逆置这个数组那一段的
开始下标from 要逆置那一段的结束下标to */

```
void Reverse(int R[], int from, int to){  
    int i, temp;//temp为临时变量  
    for(i=0; i < (to - from + 1)/2; i++){  
        temp = R[from + i];  
        R[from + i] = R[to - i];  
        R[to - i] = temp;  
    }  
}
```

考点一：线性表的高阶应用

3、设线性表 $L=(a_1, a_2, a_3, \dots, a_{n-2}, a_{n-1}, a_n)$ 采用带头结点的单链表保存，链表中结点定义如下：

```
typedef struct node{  
    int data;  
    struct node * next;  
} NODE;
```

请设计一个空间复杂度为 $O(1)$ 且时间上尽可能高效的算法，重新排列 L 中的各结点，得到线性表 $L'=(a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}, \dots)$ 。

要求：

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想，采用 C 或 C++ 语言描述算法，关键之处给出注释。
- (3) 说明你所设计的算法的时间复杂度。

考点一：线性表的高阶应用

(1)算法的基本设计思想：算法分 3 步完成。

第 1 步，采用两个指针交替前行，找到单链表的中间结点；

第 2 步，将单链表的后半段结点原地逆置；

第 3 步，从单链表前后两段中依次各取一个结点，按要求重排。

考点一：线性表的高阶应用

(2)算法实现：

```
void change_list(NODE * h){  
    NODE *p, * q, * r, * s;  
    p=q=h;  
    while(q->next != NULL){ //寻找中间结点  
        p=p ->next; //p 走一步  
        q=q->next;  
        if(q->next!=NULL) q=q->next; //q 走两步  
    }  
}
```

考点一：线性表的高阶应用

(2)算法实现：

```
void change_list(NODE * h){  
    q=p->next; //p 所指结点为中间结点, q 为后半段链表的首结点  
    p->next=NULL;  
    while(q!=NULL) { //将链表后半段逆置  
        r=q ->next;  
        q->next= p->next;  
        p->next=q  
        q=r;  
    }  
}
```


考点一：线性表的高阶应用

```
void change_list(NODE * h){  
    s=h->next;  //s 指向前半段的第一个数据结点，即插入点  
    q=p->next;  //q 指向后半段的第一个数据结点  
    p->next=NULL;  
    while(q!=NULL) {    //将链表后半段的结点插入到指定位置  
        r=q->next;  //r 指向后半段的下一个结点  
        q->next=s->next;  //将 q 所指结点插入到 s 所指结点之后  
        s->next=q;  
        s=q->next;  //s 指向前半段的下一个插入点  
        q=r;  
    }  
}
```

(3)算法的时间复杂度：时间复杂度为 $O(n)$ 。

考点一：线性表的高阶应用

单链表或者数组逆置解题总结：

- (1) 数组逆置：对位交换法
- (2) 单链表逆置：头逆尾顺

考点一：线性表的高阶应用

0.3 单链表或者数组旋转

0.【回顾】设有一个线性表存放在一维数组 $a[0, \dots, n-1]$ 中，编程将数组中每个元素循环右移 k 位，要求只用一个辅助单元，时间复杂度为 $O(n)$ 。

给定 $A(a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_n)$



给定 $A(a, b)$



给定 $A(b^{-1}, a^{-1})$



给定 $A(b, a^{-1})$



给定 $A(b, a) = A(b, a)$

逆置 $\text{reverse}(0, n+m-1)$

逆置 $\text{reverse}(0, n-1)$

逆置 $\text{reverse}(n, m+n-1)$

给定A(1, 2, 3, 4, a, b, c, d, e)



给定A(a, b)



给定A(b^{-1} , a^{-1})



给定A(b, a^{-1})



给定A(b, a)=A(b, a)

逆置reverse(0, 8)

给定A(e, d, c, b, a, 4, 3, 2, 1)

逆置reverse(0, n-1)

给定A(a, b, c, d, e, 4, 3, 2, 1)

逆置reverse(n, m+n-1)

给定A(a, b, c, d, e, 1, 2, 3, 4)

```
void Reverse(int* array, int p, int q) { // 这一步是实现数组逆置
```

```
    for (; p < q; p ++, q -- ){
```

```
        int temp = array[p];
```

```
        array[p] = array[q];
```

```
        array[q] = temp;
```

```
    }
```

```
}
```

```
void RightShift(int* array, int n, int k) {
```

```
    k %= n;
```

```
    Reverse(array, 0, n - 1);
```

```
    Reverse(array, 0, k - 1);
```

```
    Reverse(array, k, n - 1);
```

```
}
```

考点一：线性表的高阶应用

- 1、设将 n ($n > 1$) 个整数存放于一维数组 R 中。试设计一个在时间和空间两方面都尽可能高效的算法。将 R 中保存的序列循环左移 p ($0 < p < n$) 个位置，即将 R 中的数据由 $(X_0, X_1 \dots X_{n-1})$ 变换为 $(X_p, X_{p+1} \dots X_{n-1}, X_0, X_1 \dots X_{p-1})$ 。要求：
- (1) 给出算法的基本设计思想。
 - (2) 根据设计思想，采用 C 或 C++ 语言描述算法，关键之处给出注释。
 - (3) 说明你所设计算法的时间复杂度和空间复杂度。

考点一：线性表的高阶应用

(1) 算法的基本设计思想： 可以将这个问题看做是把数组 ab 转换成数组 ba (a 代表数组的前 p 个元素, b 代表数组中余下的 $n-p$ 个元素), 先将 a 逆置得到 $a^{-1}b$, 再将 b 逆置得到 $a^{-1}b^{-1}$, 最后将整个 $a^{-1}b^{-1}$ 逆置得到 $(a^{-1}b^{-1})^{-1}=ba$ 。设 Reverse 函数执行将数组元素逆置的操作, 对 abcdefgh 向左循环移动 3 ($p=3$) 个位置的过程如下:

Reverse(0, $p-1$)得到cbadefgh;

Reverse(p , $n-1$)得到 cbahgfed;

Reverse(0, $n-1$)得到 defghabc;

考点一：线性表的高阶应用

```
void Reverse(int R[],int from,int to) {  
    int i,temp;  
    for(i = 0; i < (to-from+1)/2; i++){  
        temp = R[from+i];  
        R[from+i] = R[to-i];  
        R[to-i] = temp;  
    }  
} // Reverse  
  
void Converse(int R[],int n,int p){  
    Reverse(R,0,p-1);  
    Reverse(R,p,n-1);  
    Reverse(R,0,n-1);  
}
```

考点一：线性表的高阶应用

2、搜索旋转排序数组

假设按照升序排序的数组在预先未知的某个点上进行了旋转。（如，数组 $[0,1,2,4,5,6,7]$ 可能变为 $[4,5,6,7,0,1,2]$ ）。搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回-1。假设数组中不存在重复的元素。

示例 1:

输入: $\text{nums} = [4,5,6,7,0,1,2]$, $\text{target} = 0$

输出: 4

示例 2:

输入: $\text{nums} = [4,5,6,7,0,1,2]$, $\text{target} = 3$

输出: -1

考点一：线性表的高阶应用

```
int binarySearch(int nums[], int left, int right, int target)
{
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

考点一：线性表的高阶应用

```
int search(int nums[], int n, int target) {  
    //使用二分法  
    if (n == 0 || nums == NULL) return -1;  
    //先用二分法找到分界点  
    int left = 0, right = n - 1;  
    while (left < right) {  
        int mid = left + (right - left) / 2;  
        if (nums[mid] > nums[right]) left = mid + 1;  
        else right = mid;  
    }  
}
```

考点一：线性表的高阶应用

```
int search(int nums[], int n, int target) {  
    //分界点为Left  
    int split = left;  
    //如果没有分界点，采用普通的二分法  
    if (split == 0)  
        return binarySearch(nums, 0, n - 1, target);  
    //判断target在哪个范围  
    left = 0;  
    right = n - 1;  
    if (target >= nums[split] && target <= nums[right]) {  
        left = split;  
    } else {  
        left = 0;  
        right = split;  
    }  
    //二分法查找  
    return binarySearch(nums, left, right, target);  
}
```

考点一：线性表的高阶应用

3、给定一个链表和一个特定值 x ，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。应当保留两个分区中每个节点的初始相对位置。示例：

输入: head = 1->4->3->2->5->2, $x = 3$

输出: 1->2->2->4->3->5

考点一：线性表的高阶应用

```
LinkedList partition(LinkedList head, int x) {  
    LinkedList dummy = (LinkedList)malloc(sizeof(LNode));  
    dummy->next = head;  
    LinkedList pre = dummy, cur = head;;  
    while (pre->next && pre->next->data < x)    pre = pre->next;  
    cur = pre;  
    while (cur->next) {  
        if (cur->next->data < x) {  
            LinkedList tmp = cur->next;  
            cur->next = tmp->next;  
            tmp->next = pre->next;  
            pre->next = tmp;  
            pre = pre->next;  
        } else {  
            cur = cur->next;  
        }  
    }  
    return dummy->next;  
}
```


考点一：线性表的高阶应用

4、链表旋转

给定一个链表，进行旋转链表的操作，讲链表每个结点向右移动 k 个位置，其中 k 是非负数。

示例1：

输入：1-> 2-> 3-> 4-> 5-> null, $k = 2$

输出：4-> 5-> 1-> 2-> 3-> null,

实例2：

输入：0-> 1-> 2-> null, $k = 4$

输出：2-> 0-> 1-> null, $k = 4$

考点一：线性表的高阶应用

```
LinkedList rotateRight(LinkedList head, int k) {  
    if (!head) return NULL;  
    int n = 1;  
    LinkedList cur = head;  
    while (cur->next) {  
        ++n;  
        cur = cur->next;  
    }  
    cur->next = head;  
    int m = n - k % n;  
    for (int i = 0; i < m; ++i) {  
        cur = cur->next;  
    }  
    LinkedList newhead = cur->next;  
    cur->next = NULL;  
    return newhead;  
}
```

考点一：线性表的高阶应用

单链表或者数组旋转解题总结：

- (1) 数组旋转：分段逆置
- (2) 单链表旋转：改为循环单链表，再断链

考点一：线性表的高阶应用

0.4 双指针策略

考点一：线性表的高阶应用

◆ 双指针策略：

- 一快，一慢
- 一早，一晚
- 一左，一右

当遇到两个单链表或者数组，实现两个单链表或者数组的比较；

当遇到一个单链表，要对单链表的不同位置进行操作或者比较；

考点一：线性表的高阶应用

1、已知一个带有表头结点的单链表，结点结构为：

data	link
------	------

假设该链表只给出了头指针 list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第 k 个位置上的结点(k 为正整数)。若查找成功，算法输出该结点的 data 域的值，并返回 1；否则，只返回 0。要求：

- (1) 描述算法的基本设计思想；
- (2) 描述算法的详细实现步骤；
- (3) 根据设计思想和实现步骤，采用程序设计语言描述算法（使用 C、C++ 语言实现），关键之处请给出简要注释。

考点一：线性表的高阶应用

第一种解法：大力出奇迹

考点一：线性表的高阶应用

(1) 算法的基本设计思想

问题的关键是设计一个尽可能高效的算法，通过链表的一趟遍历，找到倒数第 k 个结点的位置。算法的基本设计思想是：

- 定义两个指针变量 p 和 q ，初始时均指向头结点的下一个结点（链表的第一个结点）。 p 指针沿链表移动；
- 当 p 指针移动到第 k 个结点时， q 指针开始与 p 指针同步移动；
- 当 p 指针移动到最后一个结点时， q 指针所指示结点为倒数第 k 个结点。
- 以上过程对链表仅进行一遍扫描。

考点一：线性表的高阶应用

(2) 算法的详细实现步骤：

① $\text{count}=0$ ， p 和 q 指向链表表头结点的下一个结点；

② 若 p 为空，转⑤；

③ 若 count 等于 k ，则 q 指向下一个结点；否则，

$\text{count}=\text{count}+1$ ；

④ p 指向下一个结点，转②；

⑤ 若 count 等于 k ，则查找成功，输出该结点的 data 域的值，返回1；否则，说明 k 值超过了线性表的长度，查找失败，返回 0；

⑥ 算法结束。

考点一：线性表的高阶应用

(3) 算法实现：

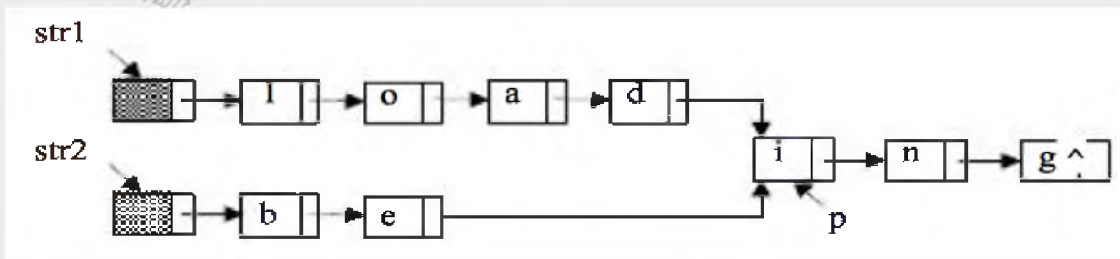
```
typedef int ElemType;           // 链表数据的类型定义
typedef struct LNode{           // 链表结点的结构定义
    ElemType data;              // 结点数据
    struct LNode *link;         // 结点链接指针
} *LinkList;
```

考点一：线性表的高阶应用

```
int Search_K(LinkedList list, int k) {  
    LinkedList p = list->link, q = list->link; // 指针p、q指示第一个结点  
    int count = 0;  
    while (p != NULL) { // 遍历链表直到最后一个结点  
        if (count < k){  
            count ++ ;  
            p = p->link;  
        }else {  
            q = q->link;  
            p = p->link;  
        }  
    }  
    if (count < k) return 0;  
    else {  
        printf("%d", q->data);  
        return 1;  
    }  
}
```

考点一：线性表的高阶应用

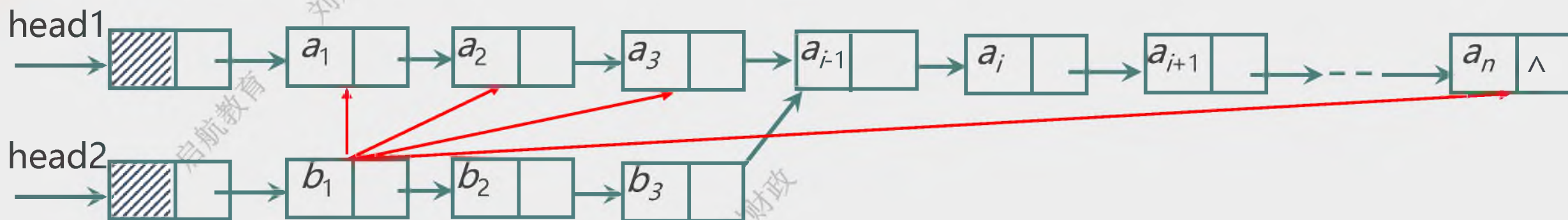
2、假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，则可共享相同的后缀存储空间，例如，“loading”和“being”的存储映像如下图所示。



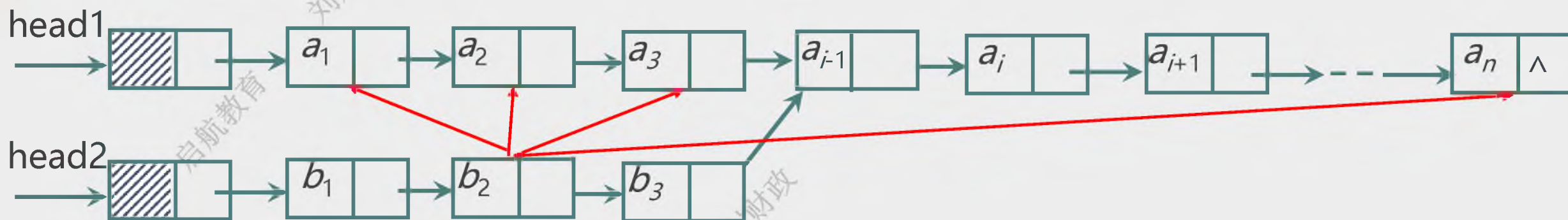
设 str1 和 str2 分别指向两个单词所在单链表的头结点，链表结点结构 `[data,next]`，请设计一个时间上尽可能高效的算法，找出由 str1 和 str2 所指向两个链表共同后缀的起始位置（如图中字符 i 所在结点的位置 p）。要求：

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想，采用 C 或 C++ 语言描述算法，关键之处给出注释。
- (3) 说明你所设计算法的时间复杂度。

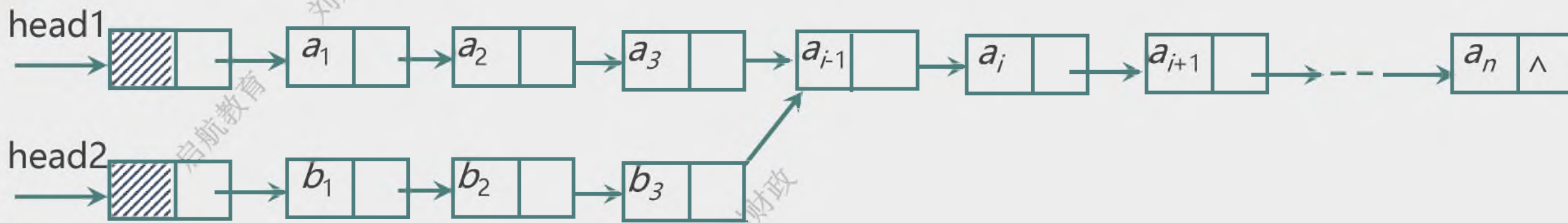
【方法一：】回溯法



【方法一：】回溯法

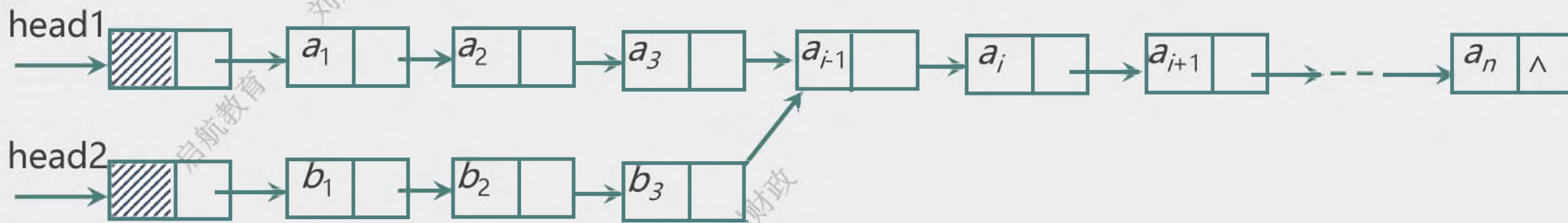


【方法二：】逆置法



破坏了原始结构

【方法三：】双指针法

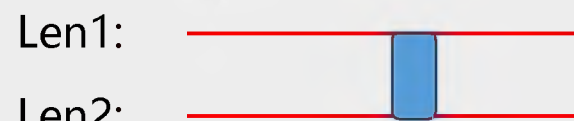


公共后缀，存在后端对齐的情况

【方法三：】双指针法

公共后缀，存在后端对齐的情况：（不是一般性）

两个单链表的长度有如下关系：



$\text{Len1} = \text{Len2}$

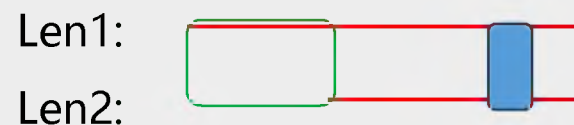
谁长谁先走



$\text{Len1} < \text{Len2}$

谁长谁削头

相等一起走



$\text{Len1} > \text{Len2}$

```
LinkedList SearchFirst(LinkedList L1, LinkedList L2) {  
    // 获取两个链表的长度  
    int len1 = getLength(L1); int len2 = getLength(L2);  
    // 用于指向较长较短链表的第一个结点  
    LinkedList longList, shortList; int dist;  
    if (len1 > len2) {  
        longList = L1->next;  
        shortList = L2->next;  
        dist = len1 - len2;  
    }else {  
        longList = L2->next;  
        shortList = L1->next;  
        dist = len2 - len1;  
    }  
}
```

```
LinkedList SearchFirst(LinkedList L1, LinkedList L2) {  
    while (dist -- ) longList = longList->next;  
    while (longList != NULL) {  
        if (longList == shortList)  
            return longList;  
        else {  
            longList = longList->next;  
            shortList = shortList->next;  
        }  
    }  
    //没有公共结点  
    return NULL;  
}
```

考点一：线性表的高阶应用

3、对于一个数组，将数组中负数的放左边，正数放右边。

```
int arra[] = { -1, 3, 9, 0, -5, -20, -3, 4, 0, 8, -12, 7 };
```



无序整数序列



负数



整数

- base (基準)



考点一: 线性表的高阶应用

// 一个线性表中的元素为全部为正或者负整数,
// 将正、负整数分开, 使线性表中所有负整数在正整数前面。

```
void Rearrange(SeqList* a, int n) {  
    int low = 0, high = n - 1;  
    // 枢纽元素, 只是暂存, 不作为比较对象  
    ElemType t = a->elem[low];  
    while (low < high) {  
        // 寻找小于0的元素  
        while (low < high && a->elem[high] >= 0) high -- ;  
        a->elem[low] = a->elem[high];  
        // 寻找大于0的元素  
        while (low < high && a->elem[low] <= 0) low ++ ;  
        a->elem[high] = a->elem[low];  
    }  
    a->elem[low] = t; // 将枢纽元素放到最终位置  
}
```

考点一：线性表的高阶应用

4、已知由 $n(n \geq 2)$ 个正整数构成的集合 $A = \{a_k | 0 \leq k < n\}$, 将其划分为两个不相交的子集 A_1 和 A_2 , 元素个数分别是 n_1 和 n_2 , A_1 和 A_2 中元素之和分别为 S_1 和 S_2 , 设计一个尽可能高效的划分算法, 满足 $|n_1 - n_2|$ 最小且 $|S_1 - S_2|$ 最大。

考点一：线性表的高阶应用

第一种解法（暴力法）： 排序

考点一：线性表的高阶应用

(1)算法的基本设计思想

由题意知,将最小的 $(n/2)_{\text{向下取整}}$ 元素放在A1中,其余的元素放在A2中,分组结果即可满足题目要求。仿照快速排序的思想,基于枢轴将n个整数划分为两个子集。根据划分后枢轴所处的位置i分别处理

①若 $i = (n/2)_{\text{向下取整}}$,则分组完成,算法结束;

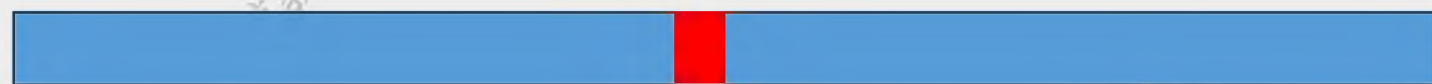
②若 $i < (n/2)_{\text{向下取整}}$,则枢轴及之前的所有元素均属于A1,继续对i之后的元素进行划分;

③若 $i > (n/2)_{\text{向下取整}}$,则枢轴及之后的所有元素均属于A2,继续对i之前的元素进行划分;

基于该设计思想实现的算法,无须对全部元素进行全排序,其平均时间复杂度是 $O(n)$,空间复杂度是 $O(1)$ 。

【方法三：】双指针法

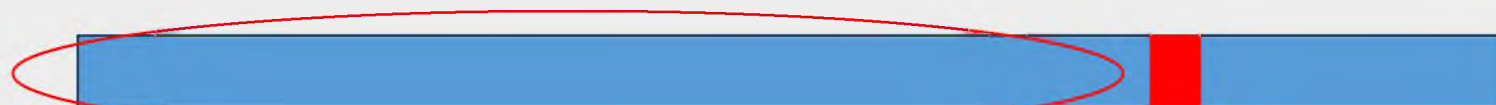
仿照快速排序的思想，基于枢轴将 n 个整数划分为两个子集。根据划分后枢轴所处的位置 i 分别处理



$i = n/2$ 结束



A1 $i < n/2$



$i > n/2$ A2

考点一：线性表的高阶应用

```
int setPartition(int a[], int n) {  
    int pivotkey, low = 0, low0 = 0;  
    int high = n - 1, high0 = n - 1;  
    int flag = 1, k = n / 2;  
    int s1 = 0, s2 = 0;  
    while (flag) {  
        pivotkey = a[low];  
        while (low < high) {  
            while (low < high && a[high] >= pivotkey) -- high;  
            if (low != high) a[low] = a[high];  
            while (low < high && a[low] <= pivotkey) ++ low;  
            if (low != high) a[high] = a[low];  
        }  
        a[low] = pivotkey;  
    }  
}
```

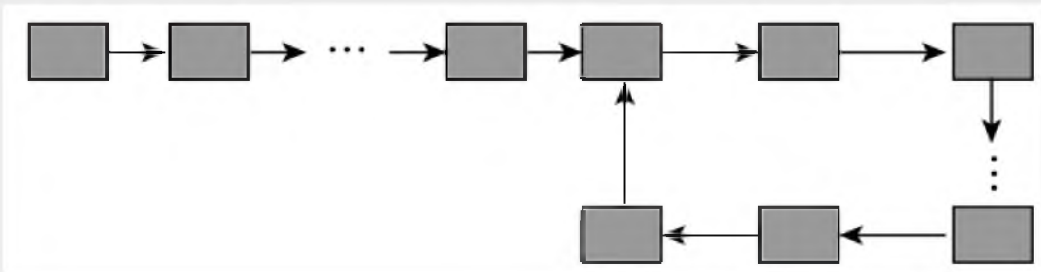
考点一：线性表的高阶应用

```
if (low == k - 1) flag = 0;
else {
    if (low < k - 1) {
        low0 = ++ low;
        high = high0;
    } else {
        high0 = -- high;
        low = low0;
    }
}
}
int i;
for (i = 0; i < k; i ++ ) s1 += a[i];
for (i = k; i < n; i ++ ) s2 += a[i];

return s2 - s1;
}
```

考点一：线性表的高阶应用

5、给定一个单链表，判断其中是否有环？



考点一：线性表的高阶应用

判断链表是否存在环，办法为：设置两个指针(fast, slow)，初始值都指向头，slow每次前进一步，fast每次前进二步，如果链表存在环，则fast必定先进入环，而slow后进入环，两个指针必定相遇。 算法实现：

```
typedef struct node{  
    char data ;  
    node * next ;  
}Node;
```

考点一：线性表的高阶应用

```
int exitLoop(LinkedList head) {  
    LinkedList fast, slow;  
    slow = head;  
    fast = head;  
    while (slow != NULL && fast->next != NULL) {  
        slow = slow->next;  
        fast = fast->next->next;  
        if (slow == fast) return 1;  
    }  
    return 0;  
}
```

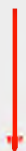
考点一：线性表的高阶应用

题目：

① 哈希： $0 \leq a_i \leq n \Rightarrow$ { ①计数
②判有无

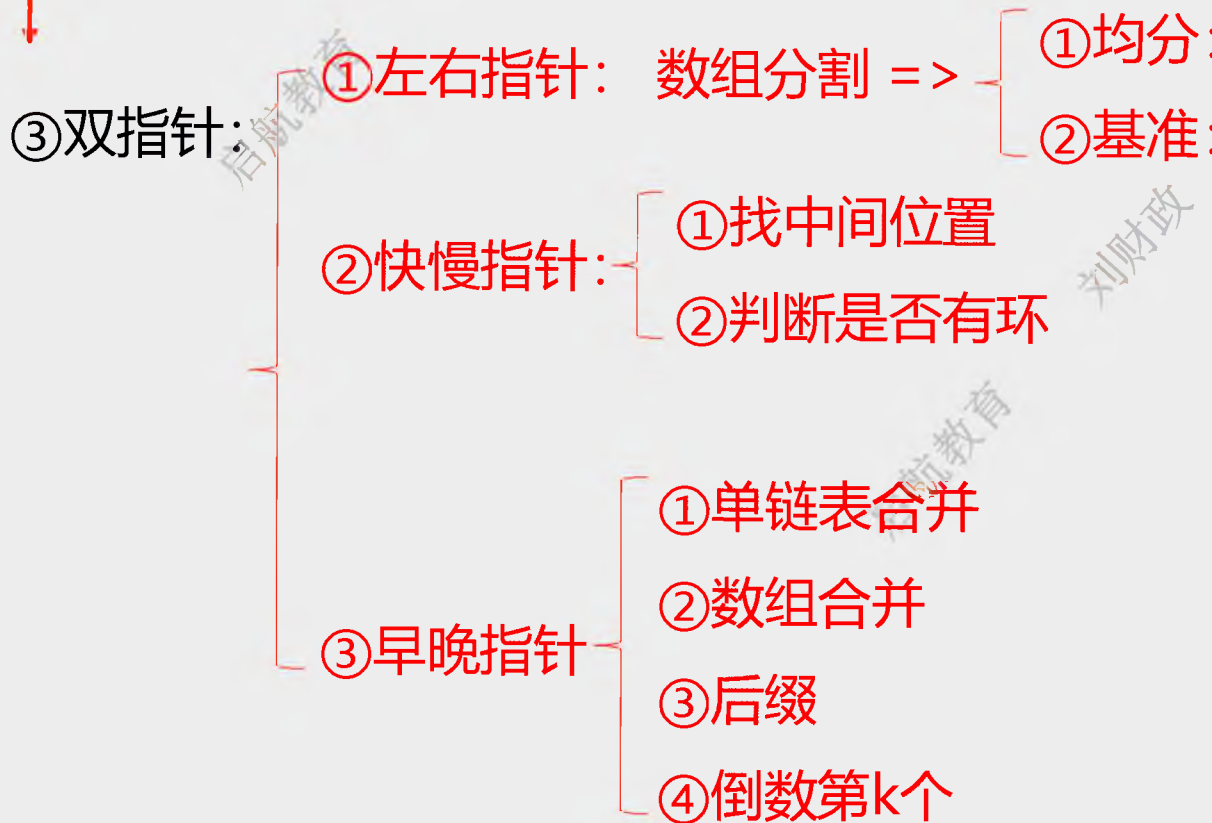


② 逆置：{ ①数组逆置：对位交换法
②单链表逆置：头逆尾顺



③ 旋转：{ ①数组旋转：分段逆置
②单链表旋转：改为循环单链表，再断链

考点一：线性表的高阶应用



- 二分法（有序）
- 快速排序（无序）

刘财政

启航教育

考点二：

二叉树的高阶应用

刘财政

启航教育

刘财政

启航教育

核心回顾

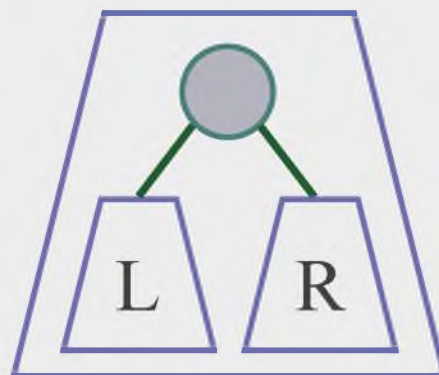
考点二: 二叉树的高阶应用

//递归形式

```
void PreorderTraverse(BTNode* T) {  
    if (T == NULL) return;  
    printf("%d ", T->data);  
    PreorderTraverse(T->Lchild);  
    PreorderTraverse(T->Rchild);  
}
```

按照**先左后右**的方式扫描二叉树,
区别仅在于访问结点的时机

```
#define DataType int  
typedef struct BTNode {  
    DataType data;  
    struct BTNode *Lchild, *Rchild;  
}BTNode;
```



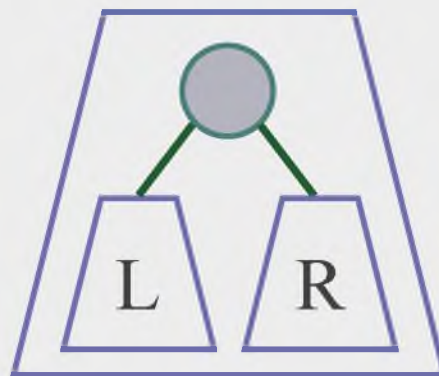
考点二: 二叉树的高阶应用

//递归形式

```
void InorderTraverse(BTNode* T) {  
    if (T == NULL) return;  
    InorderTraverse(T->Lchild);  
    printf("%d ", T->data);  
    InorderTraverse(T->Rchild);  
}
```

按照**先左后右**的方式扫描二叉树,
区别仅在于访问结点的时机

```
#define DataType int  
typedef struct BTNode {  
    DataType data;  
    struct BTNode *Lchild, *Rchild;  
}BTNode;
```



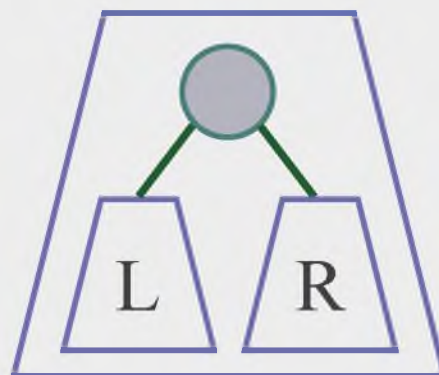
考点二: 二叉树的高阶应用

//递归形式

```
void PostorderTraverse(BTNode* T) {  
    if (T == NULL) return;  
    PostorderTraverse(T->Lchild);  
    PostorderTraverse(T->Rchild);  
    printf("%d ", T->data);  
}
```

按照**先左后右**的方式扫描二叉树,
区别仅在于访问结点的时机

```
#define DataType int  
typedef struct BTNode {  
    DataType data;  
    struct BTNode *Lchild, *Rchild;  
}BTNode;
```



考点二: 二叉树的高阶应用

```
1 #define MAX_QUEUE_SIZE 50
2 void LevelOrderTraversal(BTNode *root) {
3     if (root == NULL) {return;}
4     BTNode* Queue[MAX_QUEUE_SIZE];
5     int front = 0, rear = 0;
6     BTNode *node = root;
7     if (node != NULL) {
8         Queue[rear++] = node; //结点入队
9         while(front < rear) {
10             node = Queue[front++];
11             printf("%d ", node->data);
12             if (node->Lchild != NULL) {
13                 Queue[rear++] = node->Lchild;
14             }
15             if (node->Rchild != NULL) {
16                 Queue[rear++] = node->Rchild;
17             }
18         }
19     }
20 }
```

```
#define DataType int
typedef struct BTNode {
    DataType data;
    struct BTNode *Lchild, *Rchild;
}BTNode;
```

高阶应用

考点二: 二叉树的高阶应用

1、【南京理工大学 2019,江南大学 2020】求二叉树中叶子的个数。

【解析】求二叉树的叶子结点数: 直接利用先序遍历二叉树算法求二叉树的叶子结点数。只要将先序遍历二叉树算法中的 `visit()` 函数简单地进行修改就可以。

考点二: 二叉树的高阶应用

算法实现如下:

```
void CountLeaf(BTNode* T, int* count) {  
    if(T != NULL && T->Lchild == NULL && T->Rchild == NULL){  
        *count = *count + 1;  
    }  
    if (T) {  
        CountLeaf(T->Lchild, count);  
        CountLeaf(T->Rchild, count);  
    }  
}
```

考点二: 二叉树的高阶应用

2、【北京理工大学 2016】求二叉树的高度。

二叉树的高度是指二叉树中结点层次的最大值, 也就是该结点的左右子树中的最大高度加1。当二叉树为空时, 其高度为0。否则, 高度为其左右子树中的最大高度加1

```
int treeDepth(BTNode* T) {  
    if (T == NULL) return 0;  
  
    int hl = treeDepth(T->Lchild);  
    int hr = treeDepth(T->Rchild);  
  
    return (hl > hr) ? (hl + 1) : (hr + 1);  
}
```

考点二: 二叉树的高阶应用

3、【东北大学 2015, 吉林大学 2017, 江南大学 2018, 河海大学 2020】求二叉树的宽度。

【解析】所谓二叉树的宽度,是指二叉树各层结点个数的最大值。算法思想:要求的是特定某一层的结点个数,因此我们需要从头结点开始,记录每层的结点个数,对于当前层的每个结点,在弹出自身之后把其左、右子树压入 deque,当把当前层全部弹出队列之后,在队列中剩下的就是下一层的结点。然后比较队列的 size 和之前得到的 maxWidth,取最大值即为队列的宽度。最终队列为空,得到的 maxWidth 就是二叉树的宽度。

考点二: 二叉树的高阶应用

```
int treeWidth(BTNode* root) {  
    if(root == NULL) return 0;  
    BTNode* p = NULL;  
    BTNode* queue[MAX_QUEUE_SIZE];  
    int rear = 0;    int front = 0;  
    int width = 0;    int maxWidth = 0;  
    queue[rear++] = root;  
    while (front < rear) {  
        width = (rear - front);  
        if (maxWidth < width) maxWidth = width;  
        int i;  
        for (i = 0; i < width; i++){  
            p = queue[front++];  
            if (p->Lchild) queue[rear++] = p->Lchild;  
            if (p->Rchild) queue[rear++] = p->Rchild;  
        }  
    }  
    return maxWidth;  
}
```

考点二: 二叉树的高阶应用

4、【北京理工大学 2017】 交换二叉树的左、右子树

【解析】把一颗二叉树抽象成一个根结点和左右子结点，类似于先序遍历，交换根结点的左右子树，再去先交换左孩子的左右子树，最后交换右孩子的左右子树。

```
void Swap1(BTNode* root) {  
    if (root == NULL) return;  
    BTNode* temp = root->Lchild;  
    root->Lchild = root->Rchild;  
    root->Rchild = temp;  
    Swap1(root->Lchild);  
    Swap1(root->Rchild);  
}
```


考点二: 二叉树的高阶应用

5、二叉树的最小深度:给定一个二叉树,求解二叉树的最小深度。最小深度是从根结点到最近的叶子结点的最短路径上的结点数。

二叉树的高度是指二叉树中结点层次的最小的值,也就是该结点的左右子树中的最小高度加1。当二叉树为空时,其高度为0。否则,高度为其左右子树中的最小高度加1

```
int minDepth(BTNode* root) {  
    if(root == NULL) return 0;  
    int heightOfLeft = minDepth (root->Lchild);  
    int heightOfRight = minDepth (root->Rchild);  
    return  
        heightOfLeft > heightOfRight ? heightOfRight + 1 : heightOfLeft + 1;  
}
```

考点二: 二叉树的高阶应用

6、【中南大学2017, 东北大学 2018】对称树。

(1) 首先判断当前结点是否为NULL, 如果都为NULL, 显然是相等的,

(2) 如果不是两棵树的当前结点都为NULL, 其中有一个为NULL, 那么两棵树必不相等,

(3) 如果两棵树的两个结点的值不相等, 那么两棵树必不相等。条件判断完后, 说明当前结点相等且不为NULL。接下来就再判断当前结点的左右子树, 在递归方法中, 用递归手段去判断;

考点二: 二叉树的高阶应用

```
int isMirror(BTNode* t1, BTNode* t2) {  
    if (t1 == NULL && t2 == NULL) return 1;  
    if (t1 == NULL || t2 == NULL) return 0;  
    if (t1->data != t2->data) return 0;  
    else  
        return t1->data == t2->data  
            && isMirror(t1->Rchild, t1->Lchild)  
            && isMirror(t2->Lchild, t2->Rchild);  
}  
  
int isSymmetric(BTNode* root) {  
    return isMirror(root, root);  
}
```

考点二: 二叉树的高阶应用

7、【中国传媒大学 2016】一棵二叉树的繁茂度定义为 R 层结点数的最大值与树的高度的乘积。编写一个算法求二叉树的繁茂度。要求:

- (1) 给出算法的基本设计思想;
- (2) 根据设计思想, 采用 C 或 C++ 语言描述算法, 关键之处给出注释

【解析】 二叉树的繁茂度 = 最大宽度 \times 树的高度

思路: 先分别求树的最大宽度和高度, 再求乘积

考点二: 二叉树的高阶应用

```
int maxWidth(BTNode* root) {  
    if(root == NULL) return 0;  
    BTNode* p = NULL;  
    BTNode* queue[MAX_QUEUE_SIZE];  
    int rear = 0;    int front = 0;  
    int width = 0;    int maxWidth = 0;  
    queue[rear++] = root;  
    while (front < rear) {  
        width = (rear - front);  
        if (maxWidth < width) maxWidth = width;  
        int i;  
        for (i = 0; i < width; i++){  
            p = queue[front++];  
            if (p->Lchild) queue[rear++] = p->Lchild;  
            if (p->Rchild) queue[rear++] = p->Rchild;  
        }  
    }  
    return maxWidth;  
}
```

考点二: 二叉树的高阶应用

```
int BiTreeDepth(BTNode* T) {  
    if (T == NULL) return 0;  
  
    int hl = treeDepth(T->Lchild);  
    int hr = treeDepth(T->Rchild);  
  
    return (hl > hr) ? (hl + 1) : (hr + 1);  
}  
  
int maxPros(BiTree T){  
    return maxWidth(T) * BiTreeDepth(T);  
}
```

考点二: 二叉树的高阶应用

8、【长沙理工大学 2020】设计一个求结点 x 在二叉树中的双亲结点算法

要设计一个求二叉树 T 中指定节点 x 的双亲节点的算法, 可以按照以下步骤进行

:1) 创建一个递归函数 $\text{parent}(T, x)$, 其中 root 是当前子树的根节点 T , x 是要查找其双亲节点的节点.

2) 首先检查根节点是否为空或者根节点是否就是要查找的节点 x , 若是, 则说明没有双亲节点, 返回空(或者其他适合的标识)

3) 如果不是根节点, 检查根节点的左子树和右子树是否存在 x 节点。若左子树中找到了 x 节点, 则返回根节点作为 的双亲节点

4) 否则, 在右子树中找到了 节点, 则同样返回根节点作为 x 的双亲节点

```
BiTNode *parent(BiTree T, ElemType x){
    BiTNode *ans;
    if(T==NULL)
        return NULL;
    if(T->lchild==NULL && T->rchild==NULL)
        return NULL;
    else{
        if(T->lchild->data==x || T->rchild->data==x)
            return T;
        else{
            ans=parent(T->lchild,x);
            if(ans)
                return ans;
            ans=parent(T->rchild,x);
            if(ans)
                return ans;
            return NULL;
        }
    }
}
```


考点二：二叉树的高阶应用

9、【吉林大学 2018】假定用两个一维数组 $L[1:n]$ 和 $R[1:n]$ 作为有 n 个节点二叉树的存储结构， $L[i]$ 和 $R[i]$ 分别指示节点 i 的左儿子和右儿子，0表示空。试写一个算法判断节点 u 是否为节点 v 的子孙。

考点二: 二叉树的高阶应用

```
bool Dencendant(int[] L, int[] R, int n, int u, int v){  
    if(L[v]==0 && R[v]==0)  
        return FALSE;  
    for(int i=0; i<n; i++){  
        if(L[v]==u || R[v]==u)  
            return TRUE;  
        else{  
            if(Dencendant(L,R,n,u,L[v]) || Dencendant(L,R,n,u,R[v]))  
                return TRUE;  
            else  
                return FALSE;  
        }  
    }  
}
```

考点二：二叉树的高阶应用

10、【吉林大学 2016，2012，南京航空航天大学 2016】请设计一个算法判断二叉树T是否为一棵完全二叉树，若是，返回1；否则返回0。

完全二叉树主要有两点：

- 当一个结点有右孩子，但是没有左孩子，直接返回false
- 当一个节点有左孩子无右孩子，那么接下来要遍历的节点必须是叶子结点。
(叶子结点左右孩子为空)

考点二: 二叉树的高阶应用

```
bool check_complete_tree(BTNode * t){
    bool res = true;
    queue< BTNode *> que;
    que.push(t);
    while(!que.empty()){
        for(int i=0; i<que.size(); ++i){
            BTNode * tmp = que.front();
            que.pop();
            if(tmp->left == NULL && tmp->right != NULL){
                res = false;
                break;
            }
            if(tmp->left != NULL)
                que.push(tmp->left);
            if(tmp->right != NULL)
                que.push(tmp->right);
        }
    }
    return res;
}
```

考点二: 二叉树的高阶应用

12、【山东大学 2017】设二叉树采用链式存储结构, 定义结点结构为(leftchild, data, rightchild), 其中data为元素的值, leftchild和rightchild分别表示指向左子结点的指针和指向右子结点的指针, root为指向根的指针, p所指的节点为任一给定的节点, 编写算法, 求从根节点到p所指节点之间路径。

考点二: 二叉树的高阶应用

算法思想:

- 1.使用先序遍历T的非递归方法遍历二叉树
- 2.当访问到结点p时, 说明从根节点到p所指结点之间的路径已经找到。
- 3.在遍历过程中, 使用一个辅助栈来保存遍历过程中的结点
- 4.每次遍历到一个结点时, 将其入栈。
- 5.若当前结点是叶子结点或者其右孩子已经被访问过, 则该结点可以出栈
- 6.若出栈的结点是目标结点p, 则辅助栈中保存的结点即为从根节点到p所指结点之间的路径

考点二: 二叉树的高阶应用

```
bool path(BiTNode* root, BiTNode* node, Stack* s) {  
    BiTNode *p, *q; // ElemType p;  
    int i = 0; p = root; q = NULL;  
    init_stack(s);  
    if (p == NULL || node == NULL) return false;  
    if (p == node) {  
        push(s, p);  
        return true;  
    }  
}
```

考点二: 二叉树的高阶应用

```
while (p != NULL || !is_empty(s)) {  
    while (p) {  
        push(s, p); //非空就先压进去  
        if (p == node) //node已经压进去了  
            return true;  
        p = p->left; //先根遍历  
    }  
    top(s, &p); //回到分支的根  
    if (p->right == q || p->right == NULL) {  
        q = p; //第一个判断条件很关键, 判是否已经遍历过  
        pop(s, &p);  
        p = NULL;  
    } else {  
        p = p->right;  
    }  
}  
return false;
```

```
}
```


谢谢大家

考点三： 图的高阶应用

刘财政

启航教育

核心回顾

考点三: 图的遍历

```
void DFS(AdjGraph *G, int v)
{ ArcNode *p; int w;
  visited[v]=1;           //置已访问标记
  printf("%d ", v);       //输出被访问顶点的编号
  p=G->adjlist[v].firstarc; //p指向顶点v的第一条边的边头结点

  while (p!=NULL)
  { w=p->adjvex;
    if (visited[w]==0)
      DFS(G, w);           //若w顶点未访问, 递归访问它
    p=p->nextarc;          //p指向顶点v的下一条边的边头结点
  }
}
```

考点三: 图的高阶应用

```
void DFS_traverse_Graph(AGraph* G) {  
    int v;  
    for (v = 0; v < G->n; v ++ )  
        visited[v] = 0;  
    ArcNode* p = G->adjList[v].firstArc;  
    for (v = 0; v < G->n; v ++ )  
        if (visited[v] == 0) DFS(G, v);  
}
```

考点三: 图的高阶应用

```
void BFS(int pos, AGraph* G) {
```

```
    int queue[G->n]; int head = 0, tail = 0;
```

```
    ArcNode* p;
```

```
    queue[tail++] = pos; visited[pos] = 1;
```

```
    while (head < tail) {
```

```
        pos = queue[head++];
```

```
        printf("%d ", pos);
```

```
        p = G->adjList[pos].firstArc;
```

```
        while (p != NULL) {
```

```
            if (visited[p->adjvex] == 0) {
```

```
                queue[tail++] = p->adjvex;
```

```
                visited[p->adjvex] = 1;
```

```
            }
```

```
            p = p->nextarc;
```

```
        }
```

```
    }
```

```
}
```

考点三: 图的高阶应用

```
void BFS_traverse_Graph(AGraph* G) {  
    int i, k;  
    for (k = 0; k < G->n; k++)  
        visited[k] = 0;  
    for (i = 0; i < G->n; i++) {  
        if (visited[i] == 0)  
            BFS(i, G);  
    }  
}
```


考点三: 图的高阶应用

1、基于邻接矩阵的DFS算法

考点三: 图的高阶应用

```
int visited[MAX]; //访问标志数组
typedef struct {
    VertexType vexs[MAX]; //顶点表
    EdgeType arc[MAX][MAX]; //邻接矩阵 可看作边表
    int numVertexes, numEdges;
    int GraphType; //图的类型 无向0, 有向1
} MGraph;
//深度优先递归算法
void DFS(MGraph G, int i) {
    int j;
    visited[i] = 1; //被访问的标记
    printf("%d ", G.vexs[i]);
    for (j = 0; j < G.numVertexes; j++) {
        if (G.arc[i][j] == 1 && !visited[j]) //边(i,j)存在且j顶点未被访问, 递归
            DFS(G, j);
    }
}
```

考点三: 图的高阶应用

```
int visited[MAX]; //访问标志数组
typedef struct {
    VertexType vexs[MAX]; //顶点表
    EdgeType arc[MAX][MAX]; //邻接矩阵 可看作边表
    int numVertexes, numEdges;
    int GraphType; //图的类型 无向0, 有向1
}MGraph;
//深度优先遍历
void DFS_traverse_Graph(MGraph G){
    int i;
    for(i=0; i<G.numVertexes; i++)
        visited[i]=0;
    for(i=0; i<G.numVertexes; i++)
        if(visited[i] == 0)
            DFS(G, i);
}
```

考点三: 图的高阶应用

2、基于邻接矩阵的BFS算法

```
typedef struct {  
    int data[MAX];  
    int front;  
    int rear;  
}Queue;  
int InitQueue(Queue* Q) {  
    Q->front = 0;  
    Q->rear = 0;  
    return 1;  
}  
int QueueEmpty(Queue Q) {  
    if (Q.front == Q.rear) {  
        return 1;  
    }  
    return 0;  
}
```

考点三: 图的高阶应用

2、基于邻接矩阵的BFS算法

```
int EnQueue(Queue* Q, int e) {  
    if ((Q->rear + 1)%MAX == Q->front) { return 1;}  
    Q->data[Q->rear] = e;  
    Q->rear = (Q->rear + 1) % MAX;  
    return 0;  
}  
int DeQueue(Queue* Q, int* e) {  
    if (Q->front == Q->rear) {return 0;}  
    *e = Q->data[Q->front];  
    Q->front = (Q->front + 1) % MAX;  
    return 1;  
}  
int getQueueTopData(Queue* Q) {  
    return Q->data[Q->front];  
}
```

考点三: 图的高阶应用

```
void BFS(MGraph G, int j) {
    Queue Q;
    int e;
    InitQueue(&Q);
    visited[j] = 1;
    printf("%d ", G.vexs[j]);
    EnQueue(&Q, j);
    while (!QueueEmpty(Q)){
        DeQueue(&Q, &e);
        for (int k = 0; k < G.numVertexes; k++) {
            if (G.arc[e][k] == 1 && visited[k] == 0) {
                visited[k] = 1;
                printf("%d ", G.vexs[k]);
                EnQueue(&Q, k);
            }
        }
    }
}
```

考点三: 图的高阶应用

```
void BFS_traverse_Graph(MGraph G) {  
    for (int i = 0; i < G.numVertexes; i++) {  
        visited[i] = 0;  
    }  
    for (int j = 0; j < G.numVertexes; j++) {  
        if (!visited[j]) {  
            BFS(G, j);  
        }  
    }  
}
```

高阶应用

考点三: 图的高阶应用

1、假设图采用邻接表表示, 采用深度优先遍历和广度优先遍历, 判断结点*i*和结点*j*之间是否存在路径。

深度优先方法: 设置一个全局的数组visited[] 用于存储定点的访问情况, 初始化全是0, 并且用0表示没有访问过, 用1表示访问过。当采用深度优先遍历方式时, 如果从*i*出发, 遍历结束时, 如果visited[j] = 1, 表示结点*i*和结点*j*之间有路径, 否则没有路径。

```
int visited[MaxVetex];
```

```
int DFSMethod01(AGraph* G, int i, int j) {
```

```
    memset(visited, 0, sizeof(visited));
```

```
    DFS(i, G);
```

```
    if (visited[j] == 0) return 0;
```

```
    else return 1;
```

```
}
```

考点三: 图的高阶应用

1、假设图采用邻接表表示, 采用深度优先遍历和广度优先遍历, 判断结点*i*和结点*j*之间是否存在路径。

广度优先方法: 设置一个全局的数组visited[] 用于存储定点的访问情况, 初始化全是0, 并且用0表示没有访问过, 用1表示访问过。当采用广度优先遍历方式时, 如果从*i*出发, 遍历结束时, 如果visited[j] = 1, 表示结点*i*和结点*j*之间有路径, 否则没有路径。

```
int visited[MaxVetex];  
int BFSMethod(AGraph* G, int i, int j) {  
    memset(visited, 0, sizeof(visited));  
    BFS(i, G);  
    if (visited[j] == 0) return 0;  
    else return 1;  
}
```

考点三：图的高阶应用

2、假设图采用邻接表表示，设计一个算法，判断无向图是否是连通图

深度优先方法：设置一个全局的数组visited[] 用于存储定点的访问情况，初始化全是0，并且用0表示没有访问过，用1表示访问过。当采用深度优先遍历方式时，如果从i出发，遍历结束时，遍历结束时，只有存在结点j， $visited[j] = 0$ ，就表示不是连通图。

考点三: 图的高阶应用

```
int DFSMethod(AGraph *G, int i) {  
    int flag = 1;  
    memset(visited, 0, sizeof(visited));  
    DFS(i, G);  
    int j;  
    for (j = 1; j <= G->n; j ++ ) {  
        if (visited[j] == 0) {  
            flag = 0;  
            break;  
        }  
    }  
    return flag;  
}
```

考点三：图的高阶应用

2、假设图采用邻接表表示，设计一个算法，判断无向图是否是连通图

广度优先方法：设置一个全局的数组visited[] 用于存储定点的访问情况，初始化全是0，并且用0表示没有访问过，用1表示访问过。当采用广度优先遍历方式时，如果从i出发，遍历结束时，只有存在结点j， $visited[j] = 0$ ，就表示不是连通图。

考点三: 图的高阶应用

```
int BFSMethod(AGraph *G, int i) {  
    int flag = 1;  
    memset(visited, 0, sizeof(visited));  
    BFS(i, G);  
    int j;  
    for (j = 1; j <= G->n; j ++ ) {  
        if (visited[j] == 0) {  
            flag = 0;  
            break;  
        }  
    }  
    return flag;  
}
```

考点三: 图的高阶应用

3、假设图采用邻接表表示, 设计一个算法, 求解距离结点U最远的结点V;

图采用邻接表表示, 采用广度优先搜索, 从结点U出发, 最后一层的顶点距离U比较远, 特别的, 最后一层的最后一个结点一定是最远的.

考点三: 图的高阶应用

```
int maxDist(AGraph* G, int U){
    ArcNode *p;
    int queue[MAX_SIZE], front = 0, rear = 0, j, k;
    rear ++; queue[rear] = U; visited[U] = 1; int V;
    while(rear != front) {
        front = (front + 1) % MAX_SIZE;
        V = queue[front];
        p = G -> adjList[V].firstArc;
        while(p != NULL) {
            j = p -> adjvex;
            if( visited[j] == 0) {
                visited[j] = 1;
                rear = (rear + 1) % MAX_SIZE;
                queue[rear] = j;
            }
            p = p->nextarc;
        }
    }
    return V;
}
```


考点三: 图的高阶应用

5、已知某图的邻接矩阵为A, 若从顶点i到顶点j有边, 则 $A[i, j]=1$, 否则 $A[i, j]=0$ 。

试编写一算法求矩阵A的传递包C: 使得若从顶点i到顶点j有一条或多条路径, 则

$C[i, j]=1$, 否则 $C[i, j]=0$ 。

```
typedef int adjmatrix [maxvtxnum][maxvtxnum];
```

```
void Change(adjmatrix A, adjmatrix C, int n)
```

考点三：图的高阶应用

借鉴多源最短路径的Floyd算法。可将Floyd算法简化，仅判断顶点间是否联通即可。首先，如果顶点i与顶点j存在直接路径，则顶点i、j联通；若顶点i存在到顶点k的路径，而顶点k存在到顶点j的路径，则顶点i、j同样联通。表达式为：

$C[i][j] = C[i][j] \text{ or } (C[i][k] \text{ and } C[k][j])$ //C为传递包

用三层循环遍历即可求出传递包C。描述如下：

for 间接通过每个节点k

for 每个节点i

for 每个节点j

$C[i][j] = C[i][j] \text{ or } (C[i][k] \text{ and } C[k][j])$;

考点三: 图的高阶应用

```
void Change (adjmatrix A, adjmatrix C, int n) {  
    for (int i = 0; i != n; ++i)  
        for (int j = 0; j != n; ++j)  
            C[i][j] = A[i][j]; //将C初始化为邻接矩阵  
    for (int k = 0; k != n; ++k) //间接通过节点k  
        for (int i = 0; i != n; ++i)  
            for (j = 0; j != n; ++j)  
                // 节点i间接通过节点k是否和节点j联通  
                return C[i][j] = C[i][j] or (C[i][k] and C[k][j]);  
}
```

谢谢大家

考点四：其他高阶应用

1、除留余数法和链地址法解决冲突

若查找表用哈希表存储，哈希函数为 $H(\text{key}) = \text{key} \bmod n$ ， $70 < n < 100$ ，用链地址法处理冲突，设计哈希表的初始化、插入元素和删除元素的算法。

考点四：其他高阶应用

```
#define keytype int
typedef struct{ // 元素类型
    keytype key;
    char info[20];
}elemtype;

typedef struct hnode{ // 哈希表链结点类型
    elemtype data;
    struct hnode *next;
}hnode, *head;

typedef head HT[100]; // HT为哈希表类型

void InitHT(HT ht, int n);
void InsertHT(HT ht, int n, elemtype x);
int DeleteHT(HT ht, int n, keytype K); // 删除成功返回1，否则返回0
```

考点四：其他高阶应用

(1) 初始化:

```
void InitHT(HT ht, int n) {  
    int i;  
    for(i = 0; i < n; i++) {  
        ht[i]->next = NULL;  
    }  
}
```

考点四：其他高阶应用

(2) 插入：首先找到需要插入的链，如果为空则直接插入，如果不为空，遍历到链表的最后插入。如果在遍历过程中遇到相同的元素，则插入失败，退出。

```
void Insert(HT ht, int n, elemtype x) { //向哈希表中插入关键字key
```

```
    head newnode=(head)malloc(sizeof(hnode));
```

```
    head p, pre;
```

```
    if(!newnode) {
```

```
        printf("Insufficient memory !\n"); return;
```

```
    }
```

```
    newnode->data=x;
```

```
    if(ht[x.key%n]){ //该条链表不为空
```

```
        for(p=ht[x.key%n]; p!=NULL; pre=p, p=p->next){
```

```
            if(p->data.key == x.key)
```

```
                break;
```

```
        }
```

```
        newnode->next=pre->next;
```

```
        pre->next=newnode;
```

```
    } else{
```

```
        ht[x.key%n] = newnode;
```

```
        newnode->next=NULL;
```

```
    }
```

```
}
```


考点四：其他高阶应用

(3) 查找：

```
int Search(HT ht, int n, elemtype x) {  
    head p, pre;  
    if(ht[x.key%n]){ //该条链表不为空  
        for(p=ht[x.key%n]; p!=NULL; pre=p, p=p->next){  
            if(p->data.key == x.key)  
                return 1;  
        }  
        return 0;  
    } else { //该条链表为空  
        return 0;  
    }  
}
```

考点四：其他高阶应用

(4) 删除：

```
int Delete(HT ht, int n, elemtype x) { //删除哈希表中特定的关键字
    head p, pre;
    if(ht[x.key%n]){ //该条链表不为空
        for(p=ht[x.key%n]; p!=NULL; pre=p, p=p->next){
            if(p->data.key == x.key){
                head tmp = p;
                pre->next = p->next;
                free(tmp);
                return 1;
            }
        }
    }
    return 0;
}
```

考点四：其他高阶应用

2、单链表实现快排

将第一个链表第一个结点的值作为左轴，然后向右进行遍历，设置一个small指针指向左轴的下一个元素，然后比较如果比左轴小的话，使small指针指向的数据与遍历到的数据进行交换。最后将左轴元素与small指针指向的元素交换即可。之后就是递归。

```
LinkedList quicksort(LinkedList head, LinkedList end){  
    //如果头指针为空或者链表为空, 直接返回  
    if(head == NULL || head == end) return NULL;  
    int t;  
    LinkedList p = head -> next;           //用来遍历的指针  
    LinkedList small = head;  
    while( p != end){  
        if( p -> data < head -> data){      //对于小于轴的元素放在左边  
            small = small -> next;  
            t = small -> data;  
            small -> data = p -> data;  
            p -> data = t;  
        }  
        p = p -> next;  
    }  
    t = head -> data;    //遍历完后, 对左轴元素与small指向的元素交换  
    head -> data = small -> data;  
    small -> data = t;  
    quicksort(head, small);                //对左右进行递归  
    quicksort(small -> next, end);  
    return head;  
}
```

考点四：其他高阶应用

3、设计算法 Search Insert: 在一棵非空二叉排序树(按各元素的key值建立)上查找元素值为e的结点, 若该结点存在, 返回其指针; 若该结点不存在, 则插入元素值为e的新结点, 并返回新结点的指针。

```
typedef struct{
    int key;
    char info[10];
}elemtype;

typedef struct node{
    elemtype data;
    node *lchild, *rchild;
}node, *bitptr;

bitptr Search_Insert( bitptr T, elemtype e)
```

考点四：其他高阶应用

```
bitptr Search_Insert(bitptr root, Elemtyp e){
    node *p, *f, *new;
    p=root, f=root;
    while(p!=NULL){
        f=p;
        if(p->data == e)
            break;
        else if(p->data > e)
            p = p->lchild;
        else
            p=p->rchild;
    }
}
```

考点四：其他高阶应用

```
bitptr Search_Insert(bitptr root, Elemtype e){
    if(p==NULL){
        new = (node*)malloc(sizeof(node));
        new->lchild = NULL,
        new->rchild = NULL,
        new->data = e;
        if(f->data > e )
            f->lchild = new;
        else
            f->rchild = new;
        p = new;
    }
    return root;
}
```

考点四：其他高阶应用

4、给定一个二叉树，判断其是否是一个有效的二叉排序树。

假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉排序树。

考点四：其他高阶应用

```
int IsSearchTree(bitptr t) { //递归遍历二叉树是否为二叉排序树
    if(!t) //空二叉树情况
        return 1;
    else if(!(t->lchild) && !(t->rchild)) //左右子树都无情况
        return 1;
    else if((t->lchild) && !(t->rchild)) //只有左子树情况
    {
        if(t->lchild->data>t->data)
            return 0;
        else
            return IsSearchTree(t->lchild);
    }
}
```

考点四：其他高阶应用

```
int IsSearchTree(bitptr t) { //递归遍历二叉树是否为二叉排序树
    else if((t->rchild) && !(t->lchild)) //只有右子树情况
    {
        if(t->rchild->data<t->data)
            return 0;
        else
            return IsSearchTree(t->rchild);
    }
    else //左右子树全有情况
    {
        if((t->lchild->data>t->data) || (t->rchild->data<t->data))
            return 0;
        else
            return (IsSearchTree(t->lchild) && IsSearchTree(t->rchild) );
    }
}
```

考点四：其他高阶应用

5、【北方工业大学 2018】二叉排序树以二叉链表存储，设计一个高效算法，找出二叉树中所有结点数据域的最大值，并返回。

二叉树以二叉链表存储，其存储结构为：

```
typedef struct BiTNode{  
    TElemType data;  
    struct BiTNode *lchild, *rchild;  
} BiTNode, BiTree;
```

考点四：其他高阶应用

```
int getMaxValue(BiTree T){ //求最大(递归算法)
    if(T==NULL)
        return 0;
    if(T!=NULL)
    {
        if(T->data>tmax)
            tmax=T->data;
        getMaxValue(T->lchild);
        getMaxValue(T->rchild);
    }
    return tmax;
}
```

谢谢大家

考点四：其他高阶应用

(4) 删除：

```
int Delete(HT ht, int n, elemtype x) { //删除哈希表中特定的关键字
    head p, pre;
    if(ht[x.key%n]){ //该条链表不为空
        for(p=ht[x.key%n]; p!=NULL; pre=p, p=p->next){
            if(p->data.key == x.key){
                head tmp = p;
                pre->next = p->next;
                free(tmp);
                return 1;
            }
        }
    }
    return 0;
}
```

考点四：其他高阶应用

2、单链表实现快排

将第一个链表第一个结点的值作为左轴，然后向右进行遍历，设置一个small指针指向左轴的下一个元素，然后比较如果比左轴小的话，使small指针指向的数据与遍历到的数据进行交换。最后将左轴元素与small指针指向的元素交换即可。之后就是递归。