

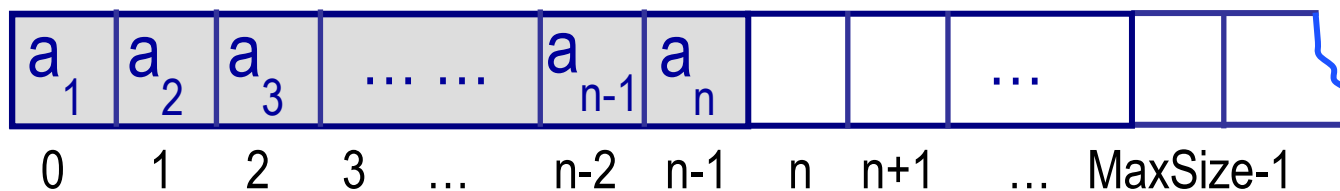


2.4 线性表的链式存储结构

将要讨论的内容

1. 线性链表的构造原理
2. 几个常用符号的说明
3. 线性链表的有关(操作)算法

顺序表



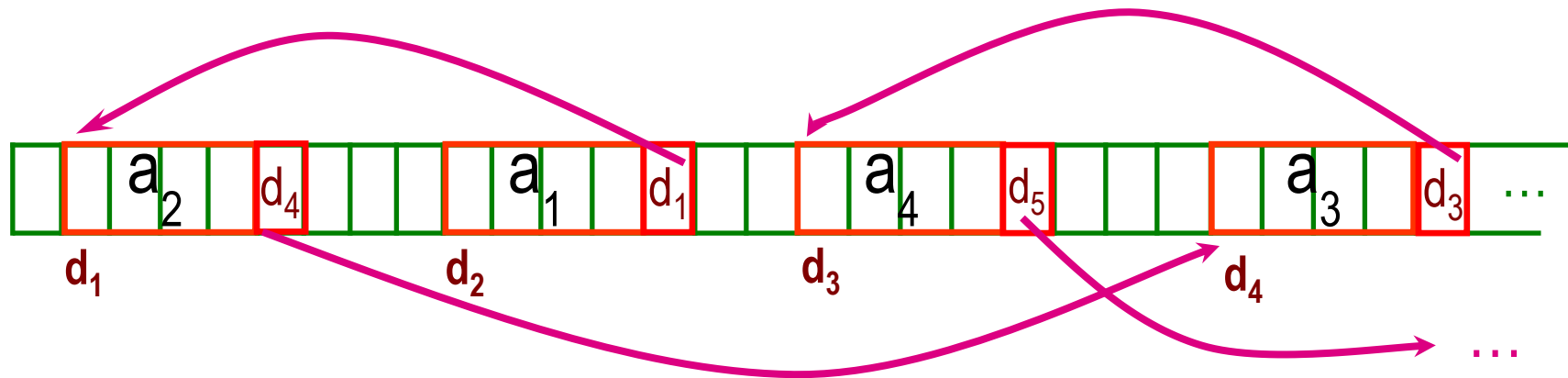
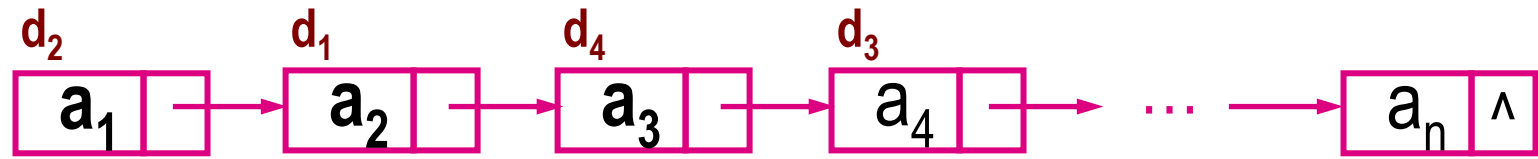
- (1) 一次分配内存
- (2) 需要连续空间
- (3) 插入和删除操作时效较低



一. 线性链表的构造原理

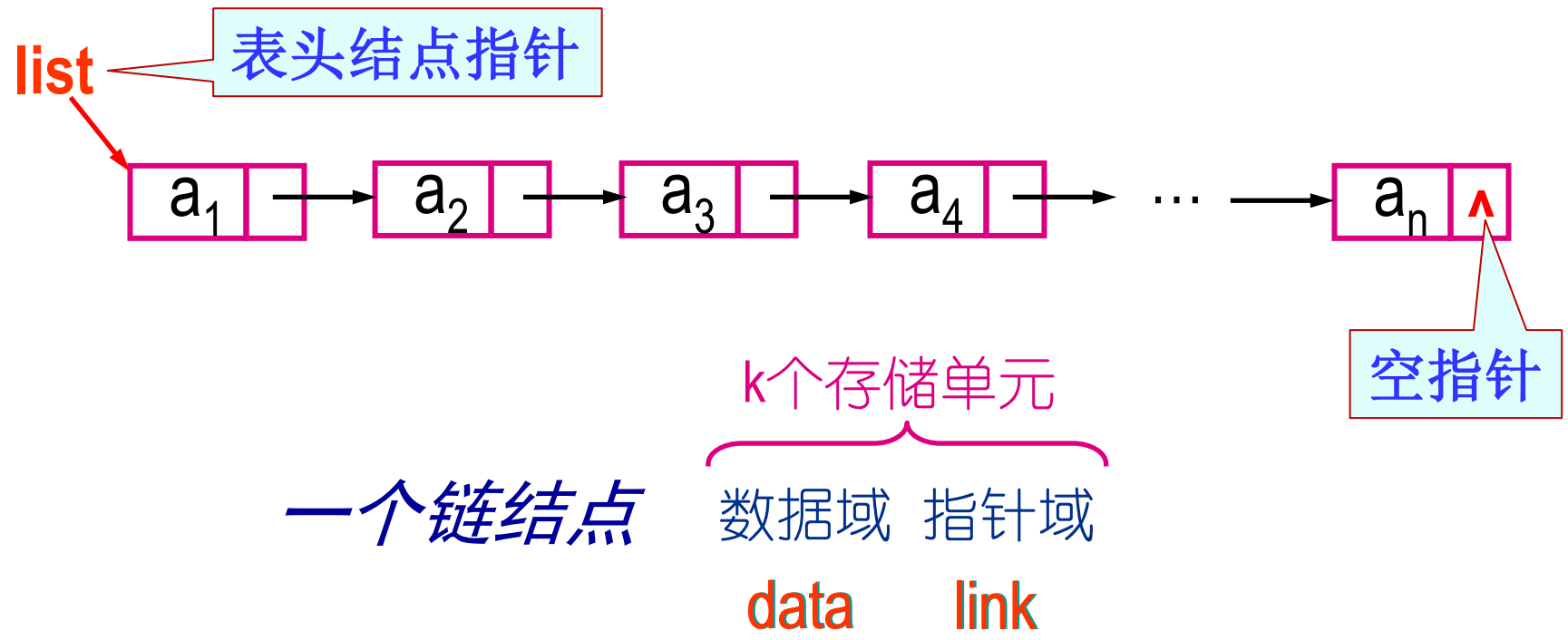
用一组地址任意的存储单元(连续的或不连续的)依次存储表中各个数据元素, 数据元素之间的逻辑关系通过 **指针** 间接地反映出来。

$$(a_1, a_2, a_3, a_4, \dots, a_{n-1}, a_n)$$






线性表的这种存储结构称为 **线性链表**，或者 **单(向)链表**，其一般形式为：





线性链表的定义

链结点: 
data link

	Num	Name	Age
a ₁	60101	张三	17
a ₂	60102	李四	16
a ₃	60103	王五	20
a ₃₀			

```
typedef struct {
    int Num;
    char Name[10];
    int Age;
} ElemType;
```

```
struct node {
    ElemType data;
    struct node *link;
};
struct node *list, *p;
```

```
struct node {
    ElemType data;
    struct node *link;
};
typedef struct node *Nodeptr;
typedef struct node Node;
Nodeptr list, p;
```

```
struct node {
    int Num;
    char Name[10];
    int Age;
    struct node *link;
};
struct node *list, *p;
```

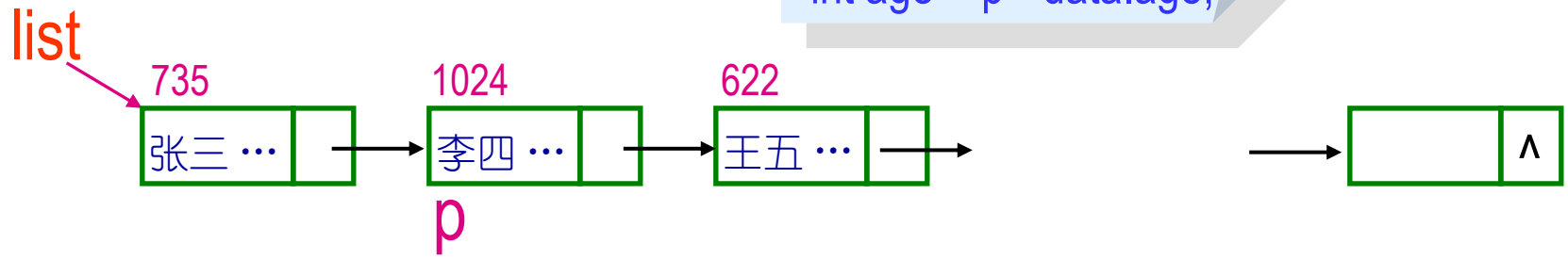


若指针变量p为指向链表中某结点的指针
(即变量p的内容为某链结点的地址), 则

```
struct node {
    int Num;
    char Name[10];
    int Age;
    struct node *link;
};
struct node *list, *p;
...
int age=p->age;
```

p->data 表示p指向的链
结点的**数据域**

```
struct node {
    ElemType data;
    struct node *link;
};
struct node *list, *p;
...
int age = p->data.age;
```



p->link 表示由p指向的链结点的指针域, 即p所指的
链结点的下一个链结点的指针 (地址)。



```
struct node{  
    ElemType  data;  
    struct node  *link;  
};
```

```
typedef struct node Node;  
typedef struct node *Nodeptr;  
Nodeptr list, p;
```

申请一个链结点的空间

```
p=(Nodeptr)malloc(sizeof(Node));
```

释放一个链结点的空间

```
free(p);
```

C语言

```
头文件: #include <stdlib.h>
```



链表使用的注意事项：

- ① 应确保链结点指针指向一个合法空间（通常由malloc函数申请而得），否则结点操作时会出现内存错误（**memory access violation**），如：
 struct Node *p;
 p->link = q;
- ② 单向链表的最后一个结点的**p->link**指针一定要为NULL，通常申请一个结点p时及时执行**p->link = NULL;**
- ③ 当链表结点删除后应及时用free(p)释放。否则会造成内存泄漏（**memory leak**），这是工程应用中常见问题；
- ④ 不要随意移动链表的**头结点指针**。



链表的基本操作

- 求线性链表的长度。
- **建立**一个线性链表。
- 在非空线性链表的第一个结点前**插入**一个数据信息为item的新结点。
- 在线性链表中由指针q 指出的结点之后**插入**一个数据信息为item的链结点。
- 在线性链表中满足某条件的结点后面**插入**一个数据信息为item的链结点。
- 从非空线性链表中**删除**链结点q(q为指向被删除链结点的指针)。



- **删除**线性链表中满足某个条件的链结点。
- 线性链表的**逆转**。
- 将两个线性链表**合并**为一个线性链表。
- **检索**线性链表中的第 i 个链结点。

.....

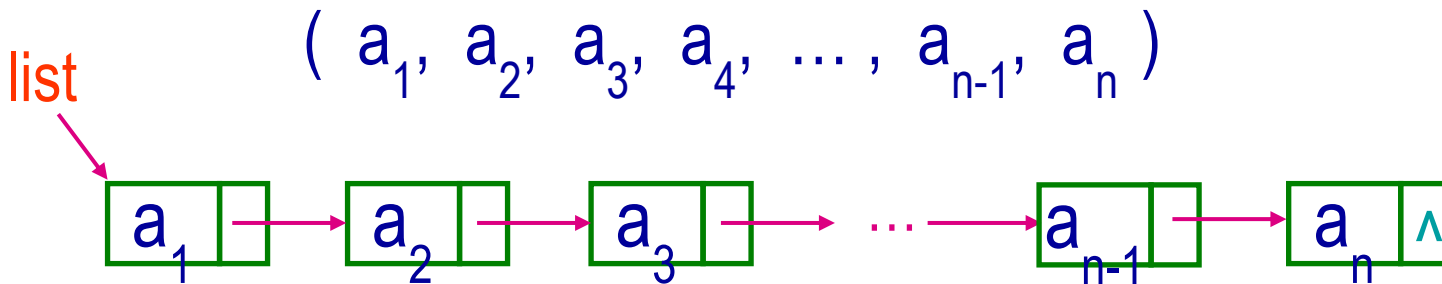


链表的基本操作

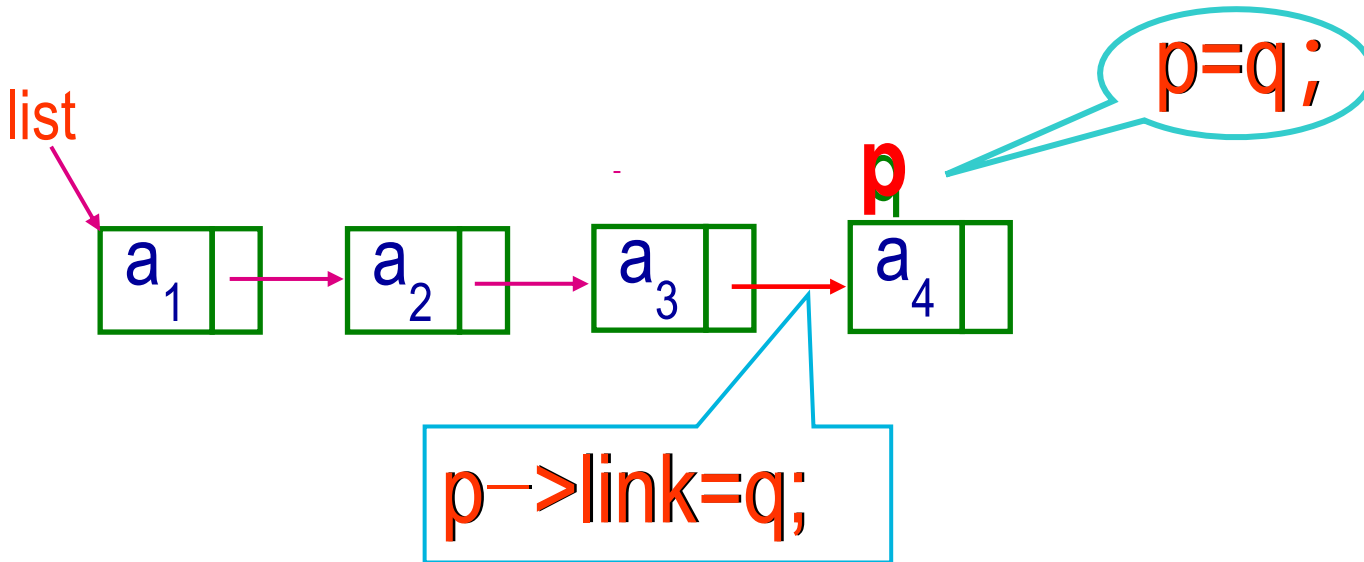
- `createList(int n);` //创建一个具有n个结点的链表
- `getLength(Nodeptr list);` //获得链表的长度
- `destroyList (Nodeprt list);` //销毁一个表
- `printList(Nodeptr list);` //输出一个表
- `insertFirst (Nodeptr list, ElemType elem);` //在链表头插入一个元素
- `insertLast(Nodeptr list , ElemType elem);` //在链表尾插入一个元素
- `insertNode(Nodeptr list , Nodeptr p, ElemType elem);` //在链表某一结点后插入包含某一个元素的结点
- `searchNode(Nodeptr list , ElemType elem);` //在链表中查找某一元素
- `deleteNode(Nodeptr list , ElemType elem);` //在链表中删除包含某一元素结点



1. 建立一个线性链表



一个结点插入链表尾部的过程:





*/*创建一个具有n个结点的链表*/*

```
Nodeptr createList( int n ) {
```

*/*list是链表头指针，q指向新申请的结点，p指向链表最后一个结点*/*

```
Nodeptr p, q, list=NULL;
```

```
int i;
```

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

```
    q=(Nodeptr)malloc(sizeof(Node));
```

```
    q->data=read();    /* 取一个数据元素 */
```

```
    q->link=NULL;
```

```
    if (list==NULL)    /*链表为空*/
```

```
        list=p=q;
```

```
    else
```

```
        p->link=q;    /* 将新结点链接在链表尾部 */
```

```
        p=q;
```

```
    }  
    return list;
```

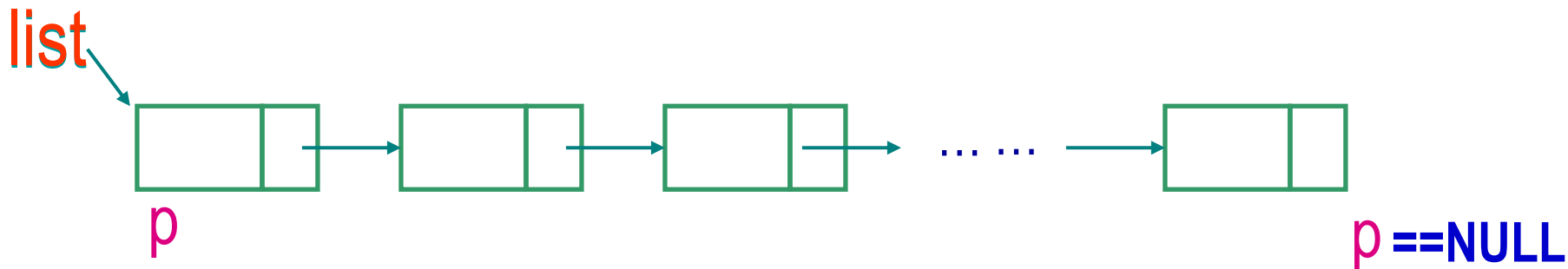
```
}
```

申请一个新的链结点

时间复杂度 $O(n)$



2. 求线性链表的长度



初始: `n=0; p=list;`

`p=p->link; n++;`

链表长度

结束: `p==NULL`

注意: list是一个指向链表头节点的指针。在实际使用时千万不要移动链表头结点指针!



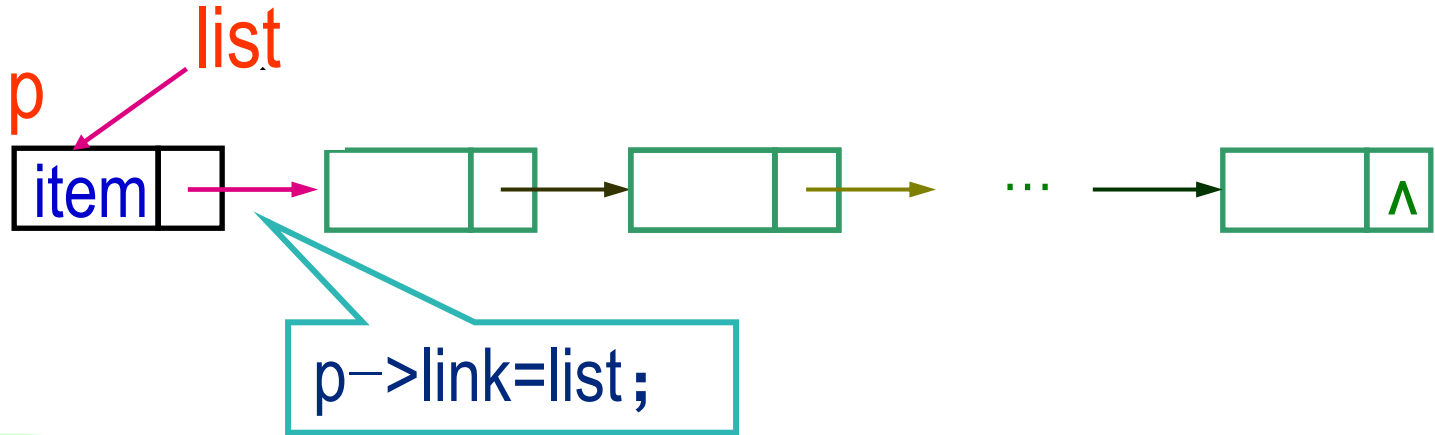
```
int getLength( Nodeptr list ){
    Nodeptr p;          /* p为遍历链表结点的指针 */
    int n=0;           /* 链表的长度置初值0 */
    for(p=list; p!=NULL; p=p->link)
        n++;          /* p依次指向链表的下一结点 */
                    /* 对链表结点累计计数 */

    return n;          /* 返回链表的长度n */
}
```

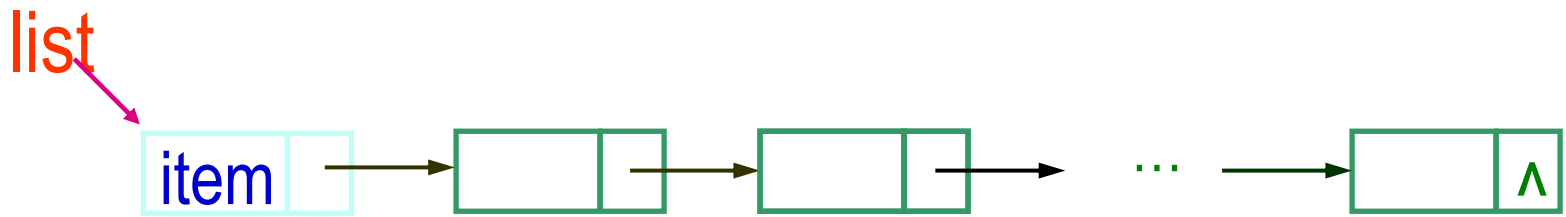
时间复杂度 $O(n)$



3. 在非空线性链表的第一个结点前插入一个数据项为item的新结点



插入后





应使用如下方式调用insertFirst函数：

```
list = insertFirst(list, item);
```

```
Nodeptr insertFirst( Nodeptr list, ElemType item ) {
```

```
    /* list指向链表第一个链结点 */
```

```
    p=(Nodeptr)malloc(sizeof(Node));
```

申请一个新结点

```
    p->data=item;
```

```
    /* 将item赋给新结点数据域 */
```

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

```
    /* 将新结点指向原链表第一个结点*/
```

```
    return p;
```

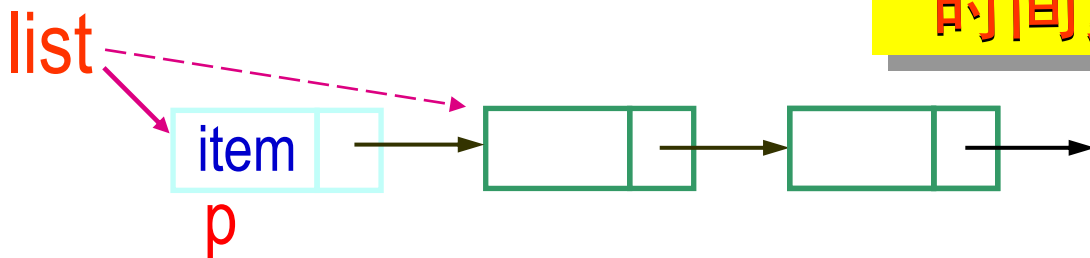
```
    /* 将链表新头指针返回 */
```

```
}
```

请思考一下为什么要返回p，而不用：

```
list = p?
```

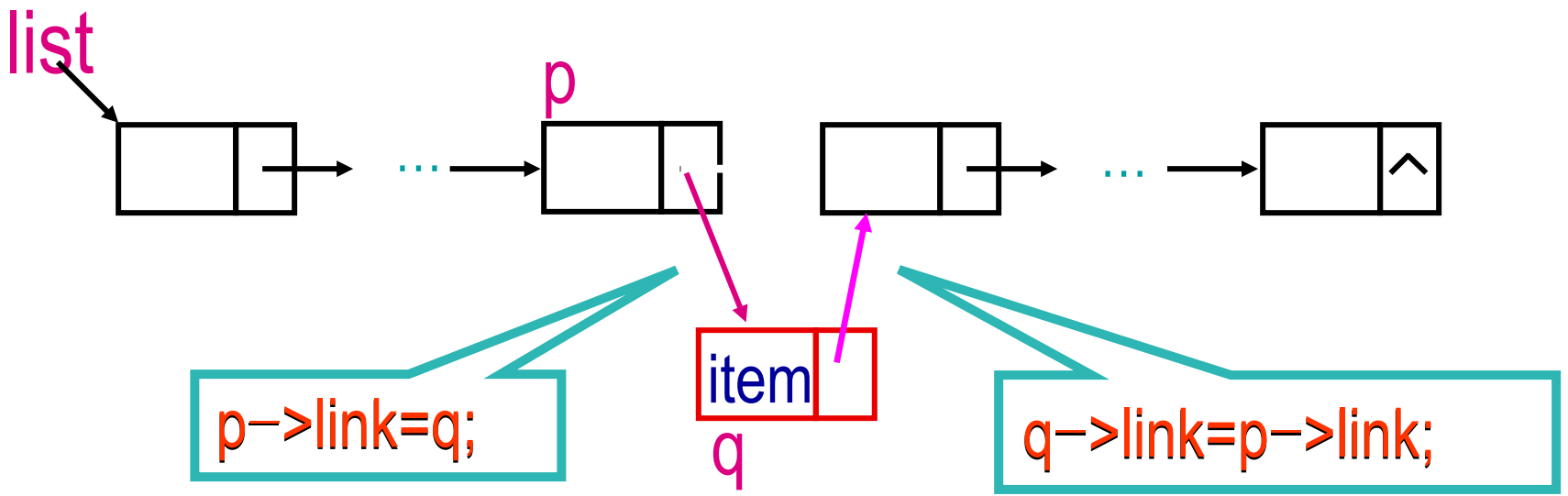
时间复杂度：O(1)





4. 在线性链表中由指针p指的链结点之后插入一个数据项为item的链结点

插入过程





```
void insertNode(Nodeptr p, ElemType item){
```

```
Nodeptr q;
```

构造一个新结点

```
q=(Nodeptr)malloc(sizeof(Node));
```

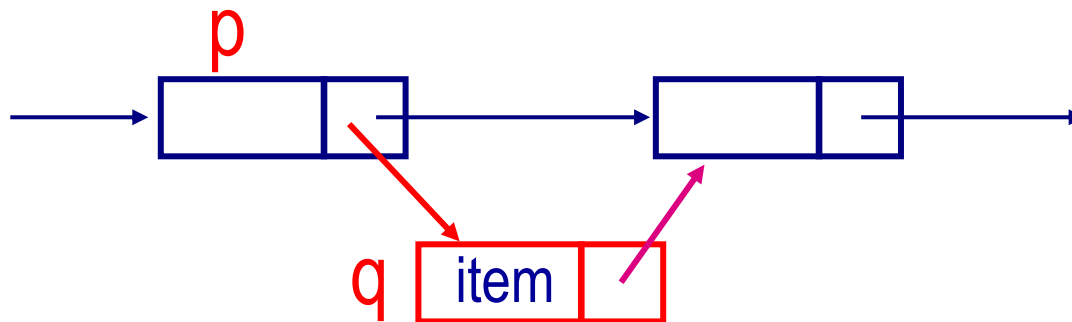
```
q->data=item; /* 将item送新结点数据域 */
```

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

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

```
}
```

时间复杂度 $O(1)$

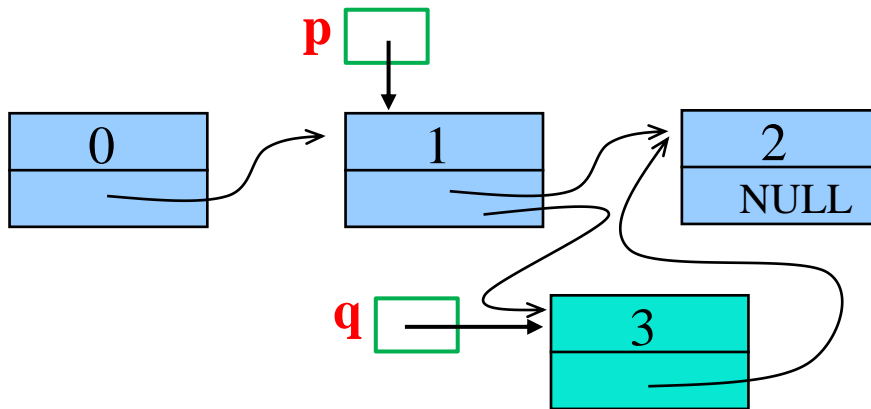




常见错误

非空线性链表中，若要在由p所指的链结点后面插入新结点q，则应执行语句 q->link=p->link; 和 p->link=q; 。

```
typedef struct Node{  
    int value;  
    struct Node* link;  
}TNode;  
TNode *p, *q;
```



错的五花八门:

q=p->link; p=q;

p->link=q->link; p->link=q;

q->link=p->link p->link=q

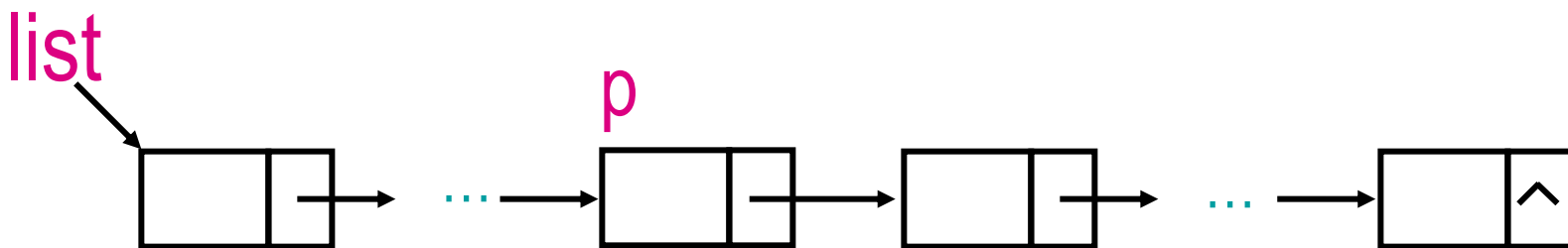
q->next=p->next; p->next=q;



2021-2022学年期末考题

设 p 指针指向单链表（单链表长度为 n ）中的某个结点（ $p \neq \text{NULL}$ ），若只知道指向该单链表第一个结点的指针和 p 指针，则在 p 指针所指结点之前和 p 指针所指结点之后插入一个结点的时间复杂度分别是_____。

- A. $O(1)$ 和 $O(n)$ B. $O(n)$ 和 $O(1)$
C. $O(n)$ 和 $O(n)$ D. $O(1)$ 和 $O(1)$

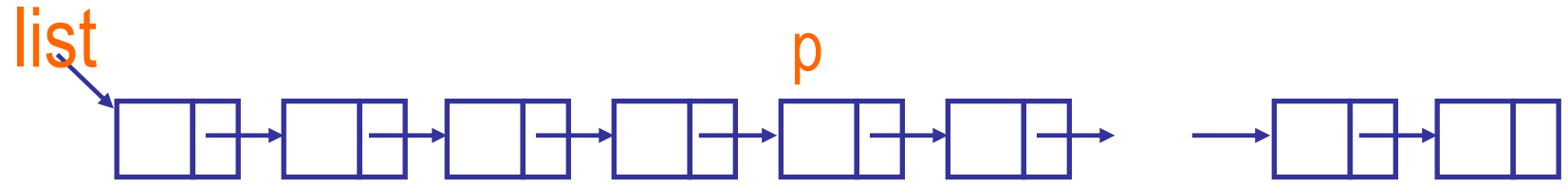




5. 在线性链表中第 $n(n>0)$ 个结点后面插入一个数据项为item的新结点

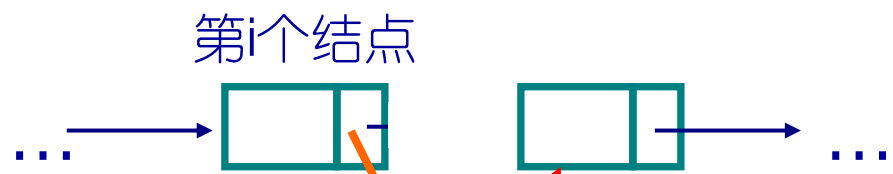
寻找第 n 个结点

如何找到第 i 个结点?



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

执行 $n-1$ 次!



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

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



```
void insertNode1( Nodeptr list, int n, ElemType item ){
    Nodeptr p=list, q;
    int i;
    for(i=1;i<=n-1;i++){    /* 寻找第i个结点 */
        if(p->link==NULL)
            return;        /* 不存在第i个结点 */
        p=p->link;
    }
    q=(Nodeptr)malloc(sizeof(Node));
    q->data=item;    /* 将item送新结点数据域 */
    q->link=p->link;
    p->link=q;    /* 将新结点插入到第i个结点之后 */
}
```

构造一个新结点

时间复杂度 $O(n)$



5a . 在有序线性链表结点后插入数据项为item的新结点

```
/* 设list是一个有序增序链表，将元素elem插入到相应位置上 */
Nodeptr insertNode(Nodeptr list, ElemType elem){
    Nodeptr p,q, r;
    r = (Nodeptr)malloc(sizeof(Node)); //创建一个数据项为elem的新结点
    r->elem = elem; r->link = NULL;
    if(list == NULL) /* list是一个空表 */
        return r;
    for(p=list; elem > p->elem && p != NULL; q = p, p = p->link) /* 找到插入位置 */
        ;
    if( p == list){ /* 在头结点前插入 */
        r->link = p;
        return r;
    }else { /* 在结点q后插入 */
        q->link = r;
        r->link = p;
    }
    return list;
}
```

时间复杂度 $O(n)$

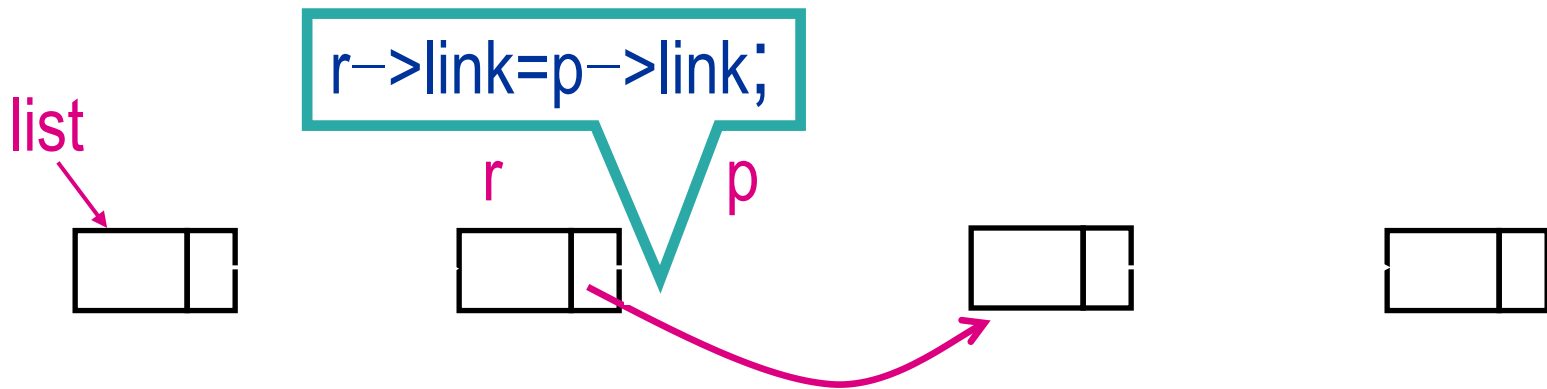
1. 在结点p前插入一个结点，必须要知道该结点的前序结点指针，在本程序中，q为p的前序结点指针；
2. 应使用如下方式调用insertNode函数：
list = insertNode(list, item);



6. 从非空线性链表中删除p指向的链结点, (已知p的直接前驱结点由r指出)



情况1：删除链表的第一个结点

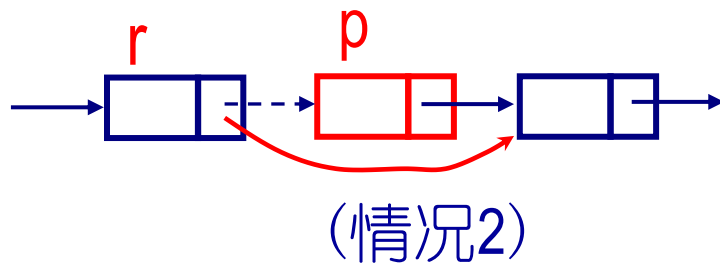
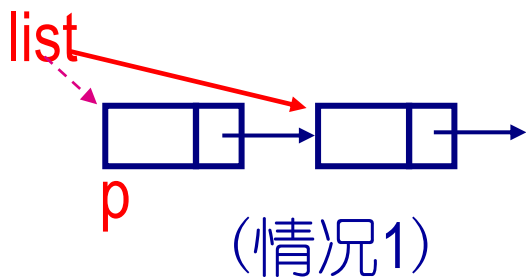


情况2：删除链表中非第一个结点



```
Nodeptr deleteNode1( Nodeptr list, Nodeptr r, Nodeptr p ){  
    if(p==list)  
        list=p->link;           /* 删除链表的第一个链结点*/  
    else  
        r->link=p->link;       /* 删除p指的链结点*/  
    free(p);                   /* 释放被删除的结点空间*/  
    return list  
}
```

时间复杂度 $O(1)$

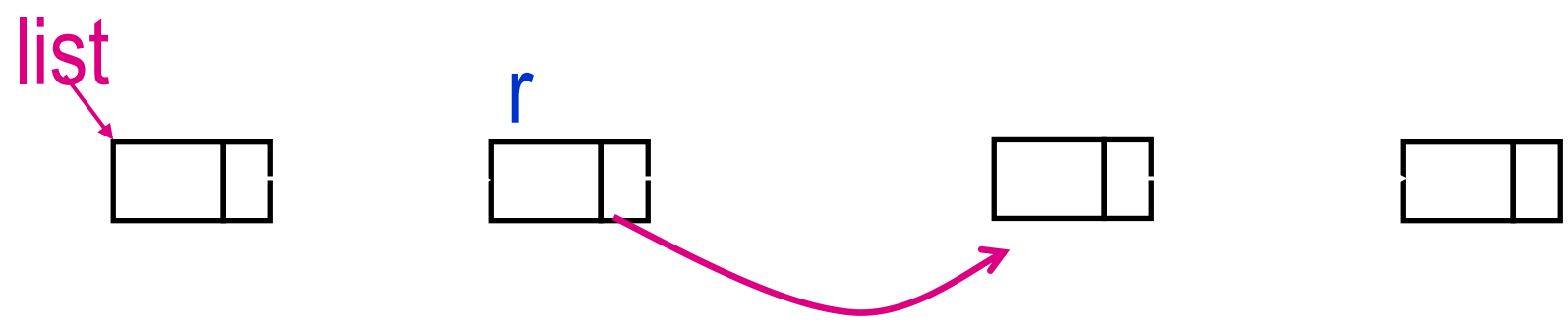
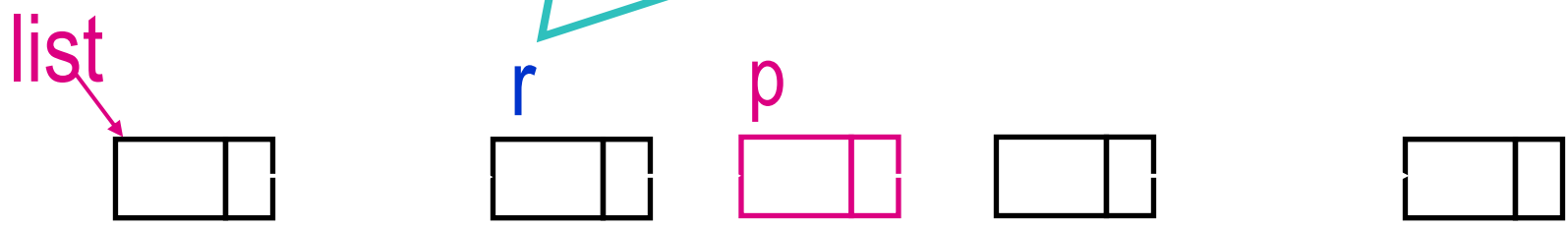




7. 从非空线性链表中删除p指向的链结点

设r是p的直接前驱结点指针，初始时r未知

```
for(r=list; r->link!=p; r=r->link)  
;
```





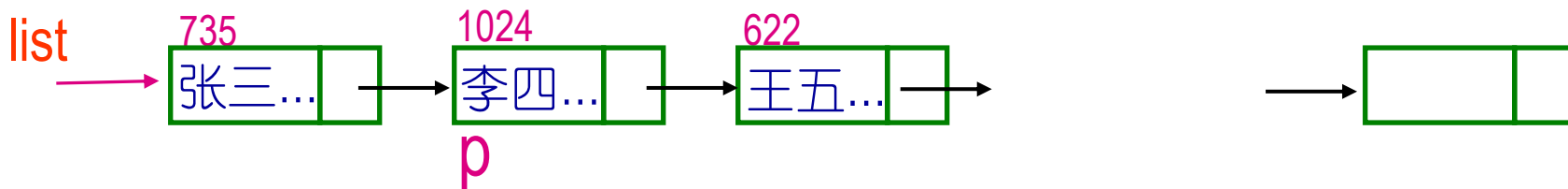
```
Nodeptr deleteNode2( Nodeptr list, Nodeptr p ){
    Nodeptr r;
    if(p==list){                                /*当删除链表第一个结点*/
        list=list->link;
        free(p);                                /*释放被删除结点的空间*/
    }
    else{
        for(r=list; r->link!=p && r->link!=NULL; r=r->link)
            ;                                    /*移向下一个链结点*/
        if(r->link!=NULL){
            r->link=p->link;
            free(p);
        }
    }
    return list;
}
```

时间复杂度 $O(n)$

寻找 p 结点的直接前驱 r



线性链表常用操作小结



指向下一个结点:

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

遍历一个链表:

```
for(p=list; p != NULL; p=p->link)
```

....

插入一个结点(在p后插入q):

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

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

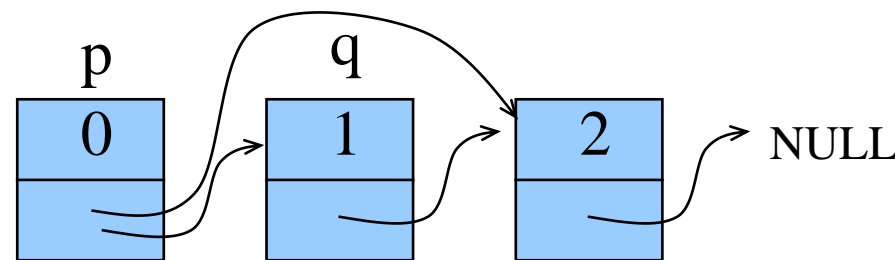
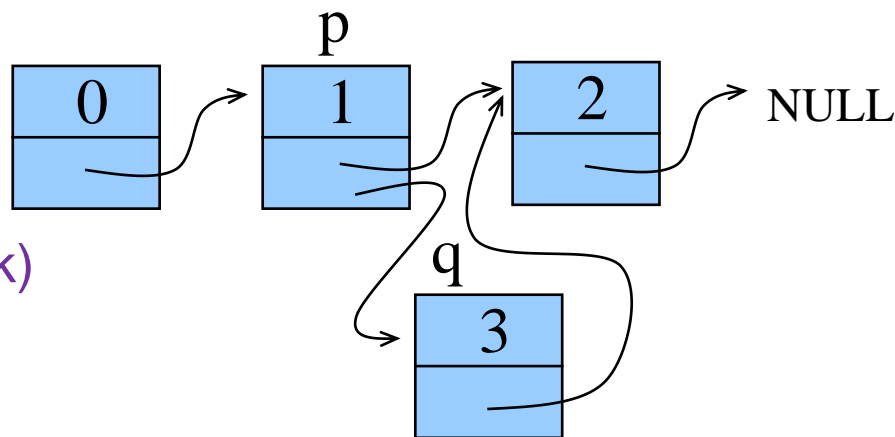
删除一个结点 (删除p的下一结点) :

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

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

```
(或 p->link = q->link;)
```

```
free(q);
```





练习

若有一单链表，定义其结点的结构类型名为`struct node`，其中指向下一结点的指针为`link`。下面定义一递归函数实现该链表的删除操作（即：删除整个链表的所有结点），形参为链表头指针，请将语句补充完整：

```
void delete(struct node *head) {  
    if(head!=NULL) {  
        delete(_____);  
        free(head);  
    }  
}
```



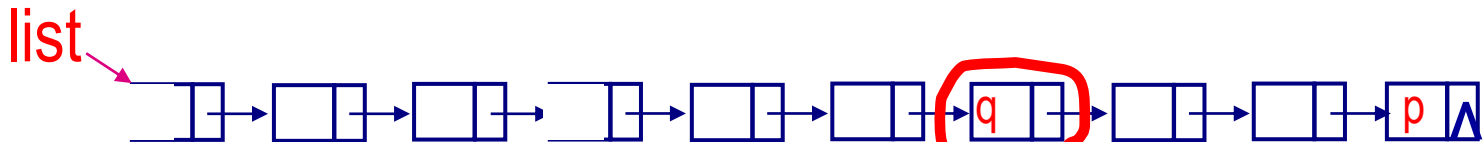
8

2009年硕士研究生入学考试*

请写一算法，该算法用尽可能高的时间效率找到由list所指的线性链表的倒数第k个结点。若找到这样的结点，算法给出该结点的地址，否则，给出NULL。

限制

1. 算法中不得求出链表长度;
2. 不允许使用除指针变量和控制变量以外的其他辅助空间。



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

```
q=q->link;
```

倒数第k=4个结点



步骤

1. 设置一个指针变量 p ，初始时指向链表的第1个结点；
2. 然后令 p 后移指向链表的第 k 个结点；
3. 再设置另一个指针变量 q ，初始时指向链表的第1个结点；
4. 利用一个循环让 p 与 q 同步沿链表向后移动；当 p 指向链表最后那个结点时， q 指向链表的倒数第 k 个结点。



```
Nodeptr searchNode(Nodeptr list,int k){
    Nodeptr p,q;
    int i;
    if(list!=NULL && k>0){
        p=list;
        for(i=1;i<k;i++){    /* 循环结束时, p指向链表的第k个结点 */
            p=p->link;
            if(p==NULL){
                printf("链表中不存在倒数第k个结点! ")
                return NULL;
            }
        }
        for(q=list; p->link!=NULL; p=p->link,q=q->link){
            ;    /* p指向链表最后一个结点, q指向倒数第k个结点 */
        }
        return q;    /* 给出链表倒数第k个结点(q指向的那个结点)的地址 */
    }
}
```




步骤

1. 设置一个指针变量 p ，初始时指向链表的第1个结点；
- ② 然后令 p 后移指向链表的第 k 个结点： 执行 $k-1$ 次
3. 再设置另一个指针变量 q ，初始时指向链表的第1个结点；
- ④ 利用一个循环让 p 与 q 同步沿链表向后移动； 当 p 指向链表最后那个结点时， q 指向链表的倒数第 k 个结点。 执行 $n-k$ 次

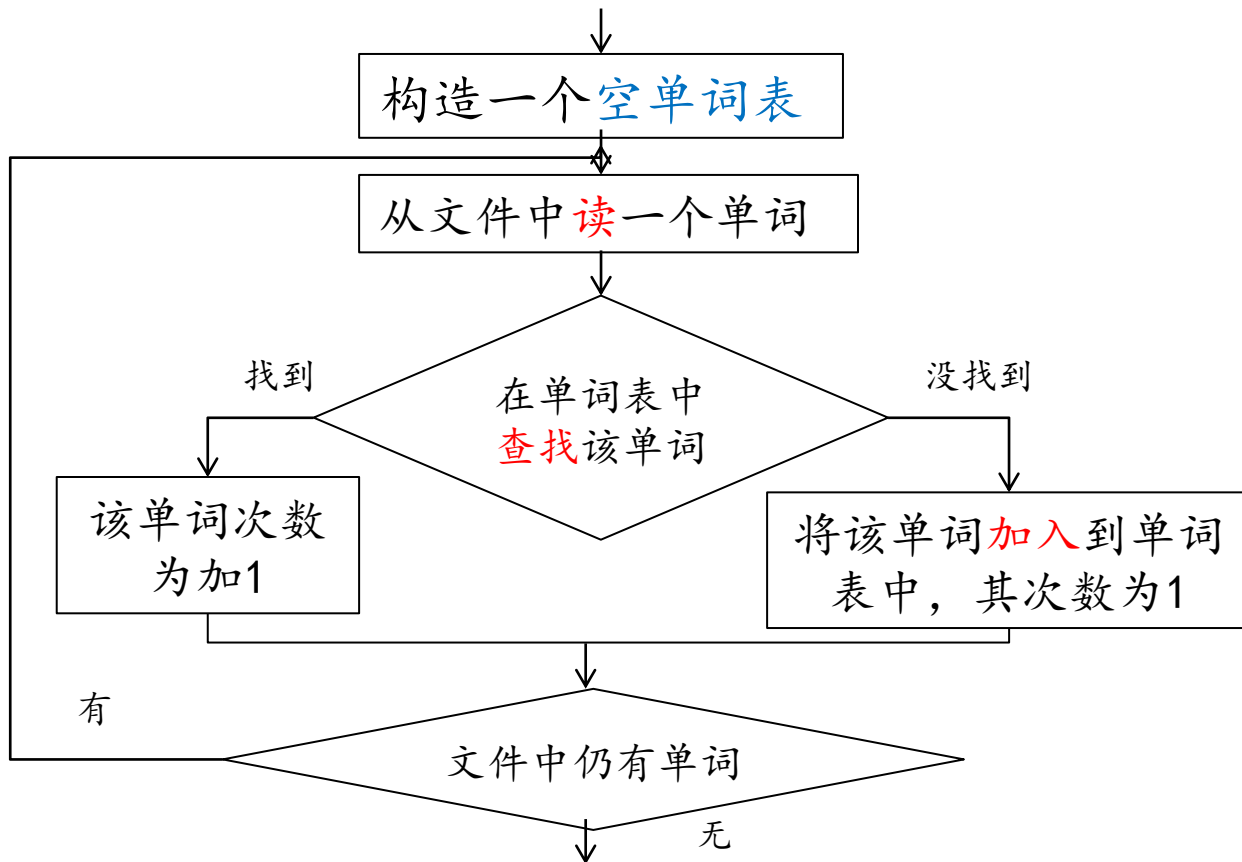
若用 n 表示链表中结点的个数, 对于任意 k ($1 \leq k \leq n$)

$O(n)$



问题2.1：词频统计 - 链表

- 问题：编写程序统计一个文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数。
- 算法分析：算法很简单，基本上只有查找和插入操作。





问题2.1：词频统计 – 链表

由于本问题有如下特点：

1. 问题规模不知（即需要统计的单词数量未知）
2. 单词表需要频繁的执行插入操作

因此，采用顺序表（数组）来构造单词表面临如下**问题**：

1. 单词表长度太小，容易满，太大，空间浪费
2. 插入操作效率低（经常需要移动大量数据）

链表具有动态申请结点（空间利用率高），插入和删除结点操作不需要移动结点，插入和删除算法效率高！
但单词查找效率低（不能用折半等高效查找算法）



问题2.1：词频统计 – 链表（代码实现）

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXWORD 32
struct node {
    char word[MAXWORD];
    int count;
    struct node *link;
}; //单词表结构
struct node *Wordlist = NULL; //单词表

int getWord(FILE *bfp, char *w);
int searchWord(char *w);
int insertWord( struct node *p, char *w);
```

```
int main(){
    char filename[32], word[MAXWORD];
    FILE *bfp;
    struct node *p;

    scanf("%s", filename);
    if((bfp = fopen(filename, "r")) == NULL){ //打开一个文件
        fprintf(stderr, "%s can't open!\n", filename);
        return -1;
    }
    while( getWord(bfp, word) != EOF) //从文件中读入一个单词
        if(searchWord(word) == -1) { //在单词表中查找插入单词
            fprintf(stderr, "Memory is full!\n");
            return -1;
        }
    for(p=Wordlist; p != NULL; p=p->link) //遍历输出单词表
        printf("%s %d\n", p->word, p->count);
    return 0;
}
```



问题2.1：词频统计 - 链表（代码实现）

/*在链表中p结点后插入包含给定单词的结点，同时置次数为1*/

```
int insertWord(struct node *p, char *w){  
    struct node *q;
```

```
    q = (struct node *)malloc(sizeof(s  
    if(q == NULL) return -1; //没有内  
    strcpy(q->word, w);
```

```
    q->count = 1;
```

```
    q->link = NULL;
```

```
    if(Wordlist == NULL) //空链表
```

```
        Wordlist = q;
```

```
    else if (p == NULL){ //插入到头结
```

```
        q->link = Wordlist;
```

```
        Wordlist = q;
```

```
    }
```

```
    else {
```

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

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

```
    }
```

```
    return 0;
```

```
}
```

/*在链表中查找一单词，若找到，则次数加1；
否则将该单词插入到有序表中相应位置，同时
次数置1*/

```
int searchWord(char *w){
```

```
    struct node *p, *q=NULL; //q为p的前序结点指针
```

```
    for(p=Wordlist; p != NULL; q=p,p=p->link){
```

```
        if(strcmp(w, p->word) < 0)
```

```
            break;
```

```
        else if(strcmp(w, p->word) == 0){
```

```
            p->count++;
```

```
            return 0 ;
```

```
        }
```

```
    }
```

```
    return insertWord(q, w);
```

在本程序中：
查找算法复杂度为 $O(n)$
插入算法复杂度为 $O(1)$



问题2.1: 词频统计 - 链表



■ 采用链表方式构造单词表具有如下特点:

● 优点

- 由于采用动态申请结点，能够适应不同规模的问题，空间利用率高
- 算法简单，插入操作效率高

● 不足

- 由于采用顺序查找，单词查找效率低

还有更好的单词表的构造及查询方法吗？



问题2.2: 多项式相加 (链表实现) *

【问题描述】 编写一个程序实现任意 (最高指数为任意正整数) 两个一元多项式相加。

【输入形式】 从标准输入中读入两行以空格分隔的整数, 每一行代表一个多项式, 且该多项式中各项的系数均为0或正整数。对于多项式 $a^n x^n + a^{n-1} x^{n-1} + \dots + a^1 x^1 + a^0 x^0$ 的输入方法如下: $a^n \quad n \quad a^{n-1} \quad n-1 \quad \dots \quad a^1 \quad 1 \quad a^0 \quad 0$

即相邻两个整数分别表示表达式中一项的系数和指数。在输入中只出现系数不为0的项。

【输出形式】 将运算结果输出到屏幕。将系数不为0的项按指数从高到低的顺序输出, 每次输出其系数和指数, 均以一个空格分隔。最后要求换行。



问题2.2: 多项式相加 (链表实现) *

【样例输入】

54 8 2 6 7 3 25 1 78 0

43 7 4 2 8 1

【样例输出】

54 8 43 7 2 6 7 3 4 2 33 1 78 0

【样例说明】 输入的两行分别代表如下表达式:

$$54x^8 + 2x^6 + 7x^3 + 25x + 78$$

$$43x^7 + 4x^2 + 8x$$

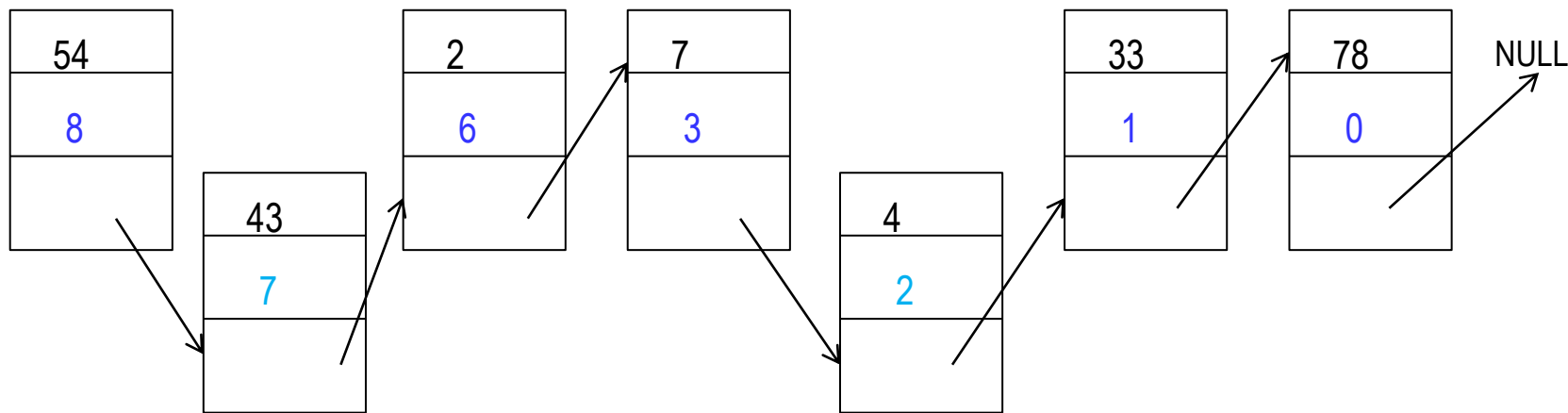
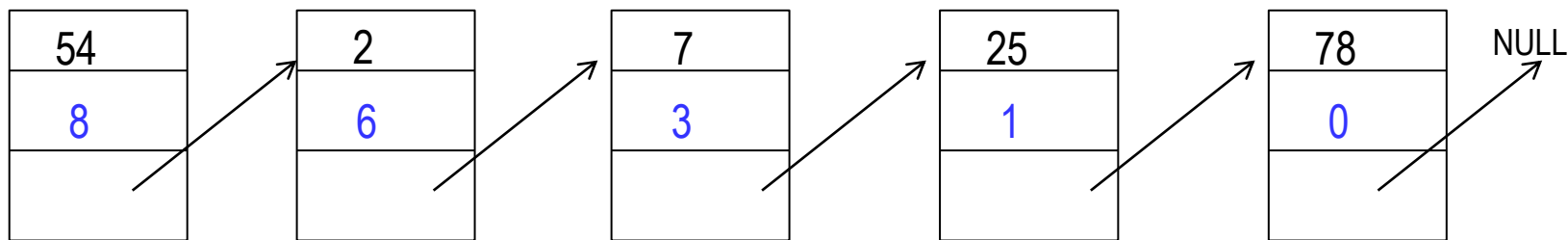
其和为

$$54x^8 + 43x^7 + 2x^6 + 7x^3 + 4x^2 + 33x + 78$$



问题2.2: 算法设计*

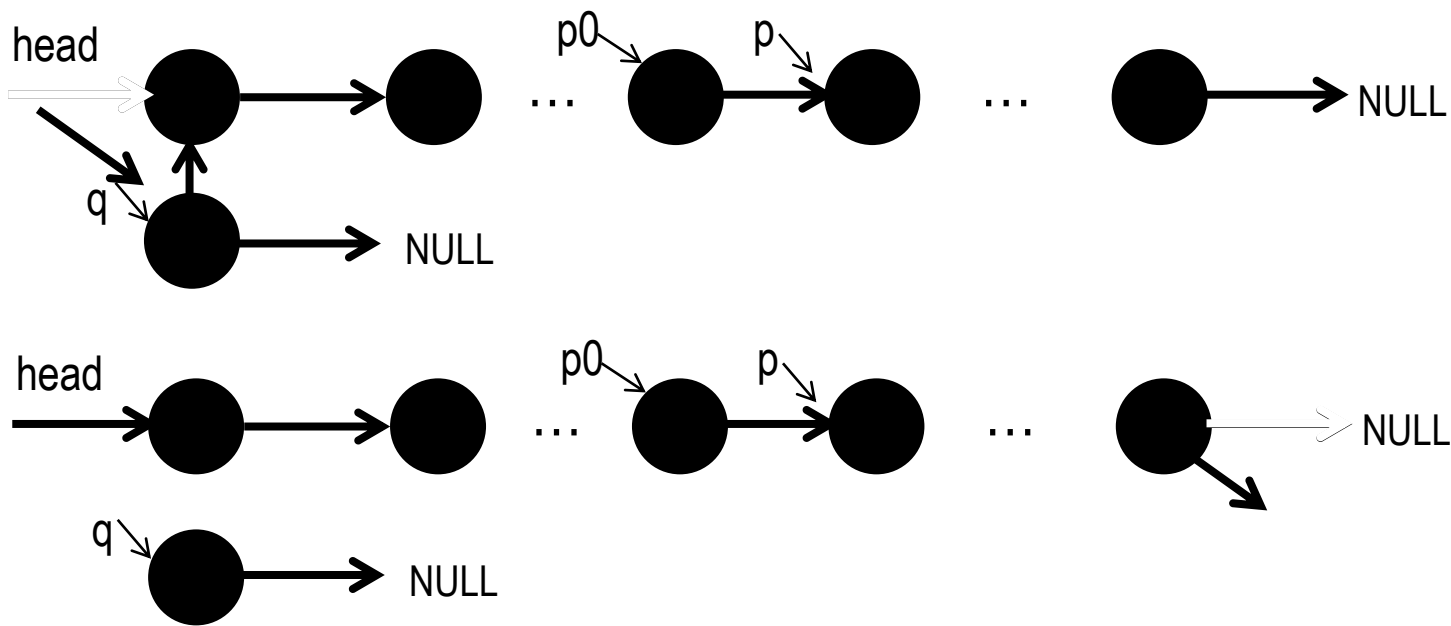
- 首先读入第一个多项式，并将其生成一个链表；
- 然后依次将第二个多项式每一项插入第一个多项式中
(或为结果多项式建立第三个链表)





问题2.2: 算法设计*

- 将第二个多项式的一个节点加到第一个多项式中有下面几种情况:
 - 第一个多项式中存在相同指数的节点: 系数直接相加
 - 第一个多项式中不存在相同指数的节点: 插入到合适的位置 (头节点前、某个节点前, 或尾节点后)





问题2.2: 多项式

//c3_4c.c

#include <stdio.h>

#include <stdlib.h>

struct Node { //一个多项式节点结构

int coe; //系数

int pow; //幂

struct Node *next;

};

int main(){

int a,n;

char c;

struct Node *head,*p,*q,*p0;

head = p = NULL;

do { //创建一个链表存放第一个多项式

scanf("%d%d%c", &a, &n, &c);

q = (struct Node *)malloc(sizeof(struct Node));

q->coe = a; q->pow = n; q->next = NULL;

if(head == NULL)

head = p = q;

else {

p->next = q;

p = p->next;

}

} while (c != '\n');

```
do { //将第二个多项式的每个项插入到第一个多项式链表中
    scanf("%d%d%c", &a, &n, &c); //生成第二个多项式的一个节点
    q = (struct Node *)malloc(sizeof(struct Node));
    q->coe = a; q->pow = n; q->next = NULL;
    for(p=head; p!=NULL; p0=p,p=p->next) {
        if(q->pow > p->pow) {
            if(p==head) { q->next = head; head = q; break; } //头
            else { q->next = p; p0->next = q; break; } //将q插入到p前
        }
        else if(q->pow == p->pow) { p->coe += q->coe; break; }
    }
    if(p== NULL) p0->next = q; //将q插入到尾节点后
} while ( c != '\n');
for(p=head; p!=NULL; p=p->next)
    printf("%d %d ", p->coe,p->pow);
return 0;
```

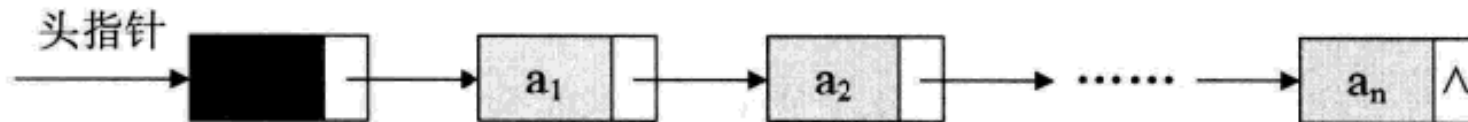


- 线性链表还有哪些经典应用？
 - 空闲内存管理
 - 文件块管理
 -



有时在构造链表时会给链表设置一个**头结点** (header) 或**哑结点** (dummy node), 指向头结点的指针称为**头指针**:

带头结点的非空链表



带头结点的空链表



设置头结点的最大好处是对链表结点的插入及删除操作统一了（不用考虑是否在链表的起始端）。其数据域一般无意义（有时也可存放链表的长度）。



2.3.4 链式存储结构的特点

无论位于链表何处，无论链表的长度如何，插入和删除操作的时间都是 $O(1)$ 。

1. 优点

- (1) 存储空间动态分配，可以根据实际需要使用。
- (2) 不需要地址连续的存储空间(不需要大块连续空间)。
- (3) 插入/删除操作只须通过修改指针实现，不必移动数据元素，操作的时间效率高。

2. 缺点

- (1) 每个链结点需要设置指针域(占用存储空间小)。
- (2) 是一种非连续存储结构，查找、定位等操作要通过顺序遍历链表实现，时间效率较低。

时间为 $O(n)$



思考

线性表可以采用顺序存储结构，也可以采用链存储结构，在实际问题中，应该根据什么原则来选择其中最合适的一种存储结构？



顺序存储结构与链式结构的比较

存储分配方式

- 顺序存储用一段连续的存储单元依次存储线性表的数据元素
- 链表采用链式储存结构，用一组不连续的存储单元存放线性表的元素

时间性能

- 查找
 - 顺序存储下，无序 $O(n)$ ，有序 $O(\log_2 n)$ ；链表 $O(n)$
- 插入和删除
 - 顺序存储平均需移动表长一半的元素，时间为 $O(n)$
 - 链表在给出结点位置后，插入和删除时间仅为 $O(1)$

空间性能

- 顺序存储需要事先分配存储空间，分大了浪费，分小了易发生溢出
- 链表不需要事先分配存储空间，需要时分配结点，元素个数不受限制

结论：

1. **时间角度**：若线性表需要频繁查找，较少进行插入和删除操作时，宜采用顺序存储结构。反之，宜采用链表结构。
2. **空间角度**：当线性表中的元素个数变化较大或者根本不知道有多大时（如单词表），最好用链表结构。而如果事先知道线性的大致长度，用顺序结构次效率会高些。

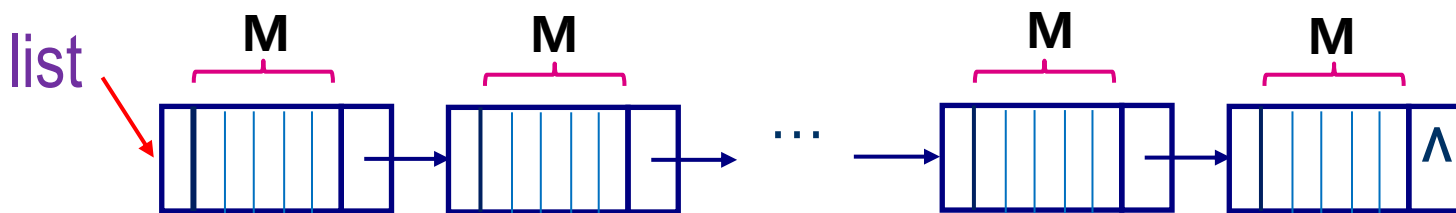


思考

在文本文件词频统计的例子中，可否结合链表和顺序表的优点设计单词表？

一种思路：

- ①每个链结点保存固定长度为 M 的数组；
- ②链结点以及结点数组中的元素按字典排序；
- ③新单词插入数组中。数组满时，该链结点分裂为两个链结点。



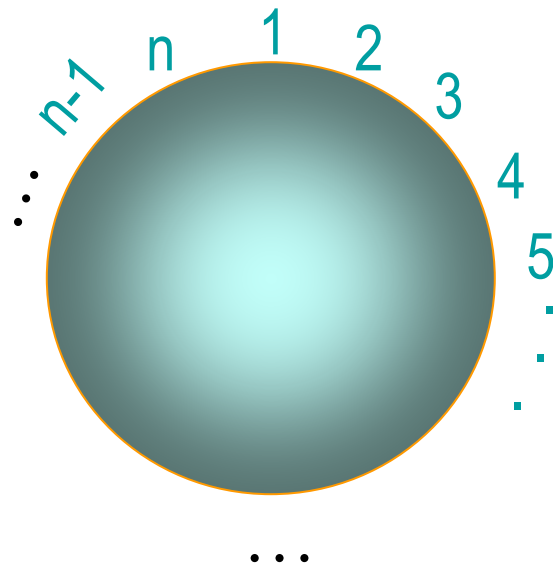


例 约瑟夫(JOSEPHU)问题

已知 n 个人(不妨分别以编号 $1, 2, 3, \dots, n$ 代表)围坐在一张圆桌周围, 编号为 k 的人从1开始报数, 数到 m 的那个人出列, 他的下一个人又从1开始继续报数, 数到 m 的那个人出列, \dots , 依此重复下去, 直到圆桌周围的人全部出列。

直到圆桌周围只剩一个人。

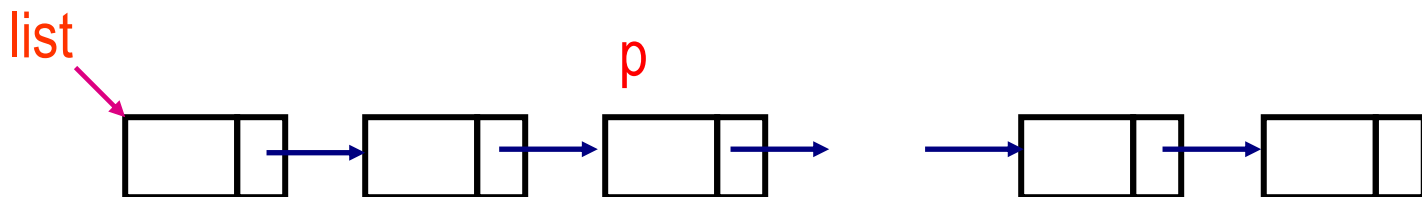
圆桌问题



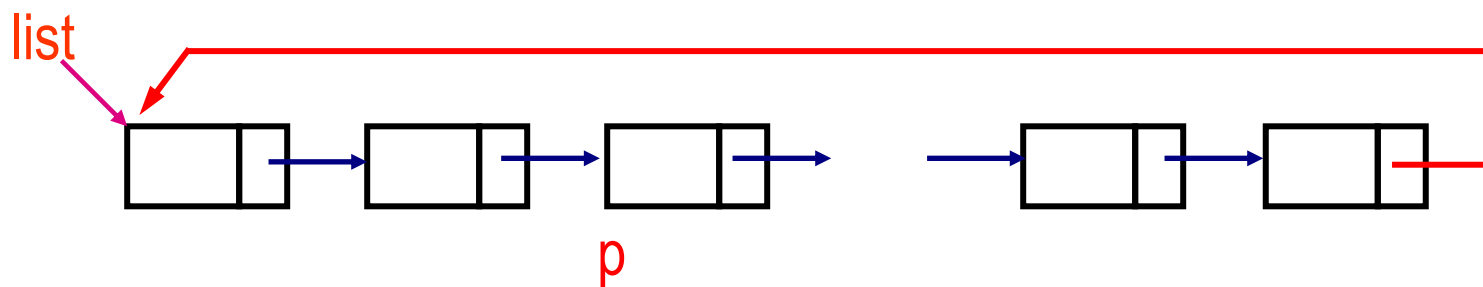


2.5 循环链表

线性链表



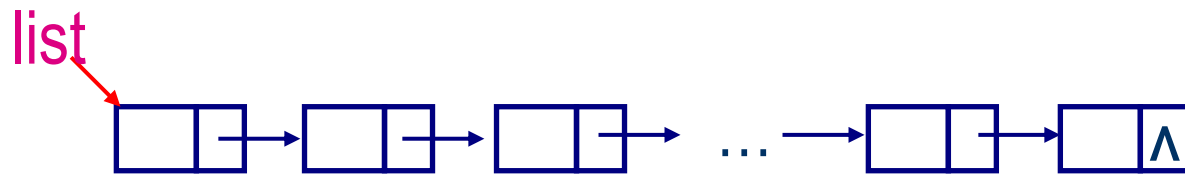
循环链表



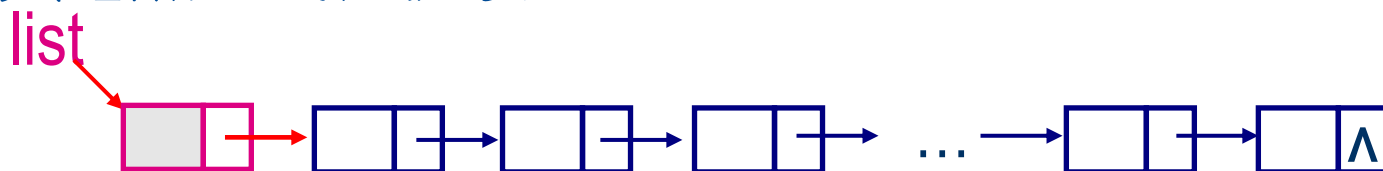
循环链表是指链表中最后那个链结点的指针域存放指向链表最前面那个结点的指针，整个链表形成一个环。

注意，循环链表的list指针可固定指向第一个结点、最后一个结点或其他结点

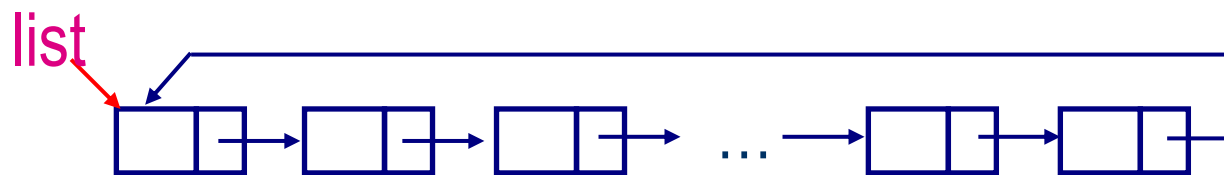
线性链表(单链表)



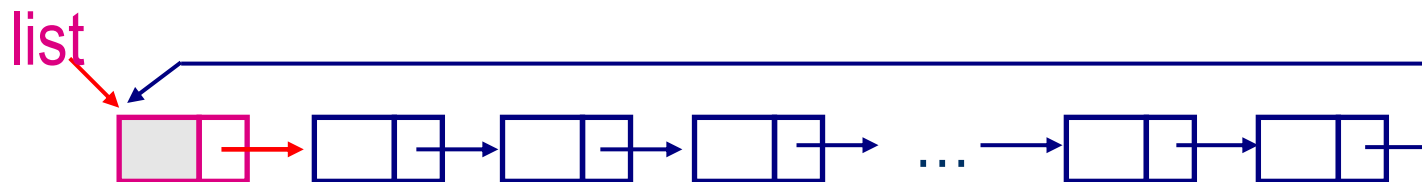
带头结点的线性链表



循环链表



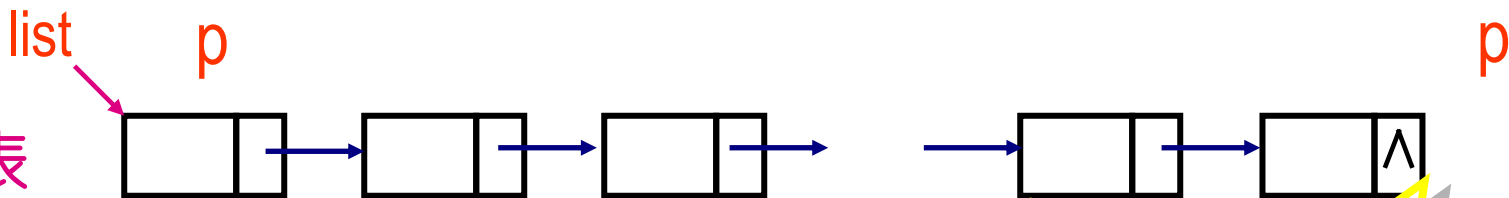
带头结点的循环链表





遍历

线性链表

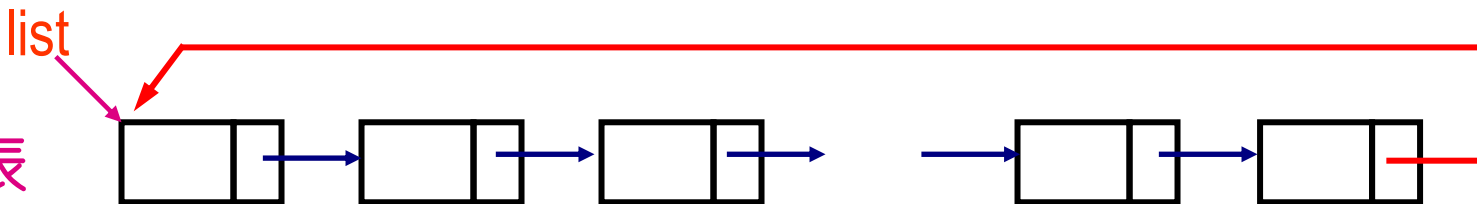


初始 $p = list$

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

结束条件: $p == NULL$

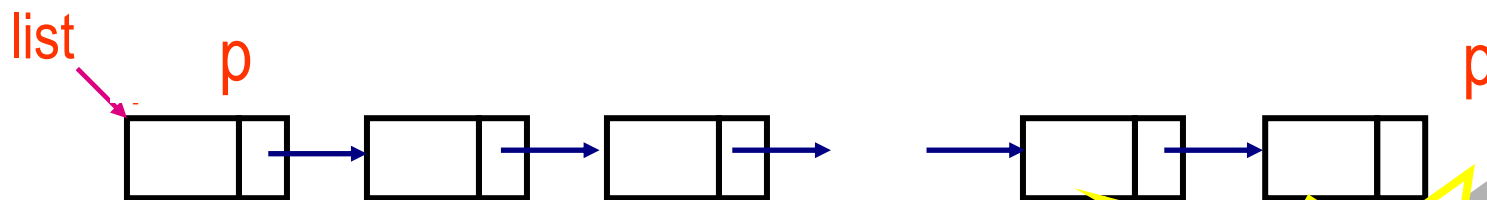
循环链表



对于循环链表，如何判断是否遍历了链表一周？



线性链表

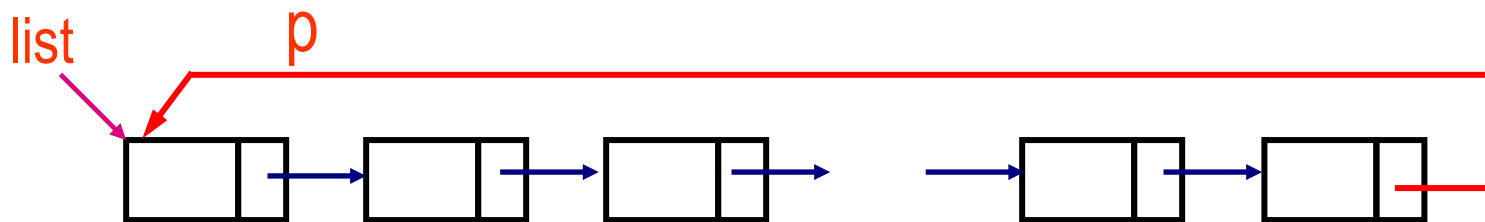


初始 $p=list$

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

结束条件: $p==NULL$

循环链表



若初始 $p=list$;

则结束条件为: $p==list$



求非空线性链表的长度

非
循
环
链
表

```
int length( NodePtr list ){
    Nodeptr p;
    int n;          /* 链表的长度置初值0 */
    for(p=list, n=0; p!=NULL; p=p->link, n++)
        ;
    return n;      /* 返回链表的长度n */
}
```

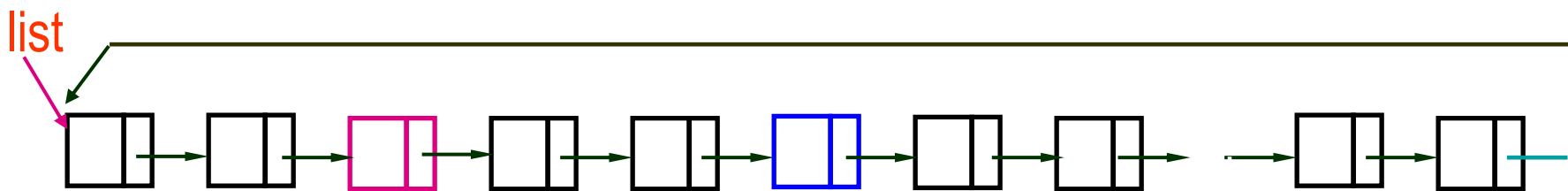
循
环
链
表

```
int length( Nodeptr list ){
    Nodeptr p=list;
    int n=0;      /* 链表的长度置初值0 */
    if(list == NULL) return 0;
    do{
        p=p->link;      n++;
    }while(p!=list);
    return n;      /* 返回链表的长度n */
}
```



例 约瑟夫(JOSEPHU)问题求解

利用一个不带头结点的
循环链表



n : 链表中链结点的个数;
 k : 第一个出发点;
 m : 报数。

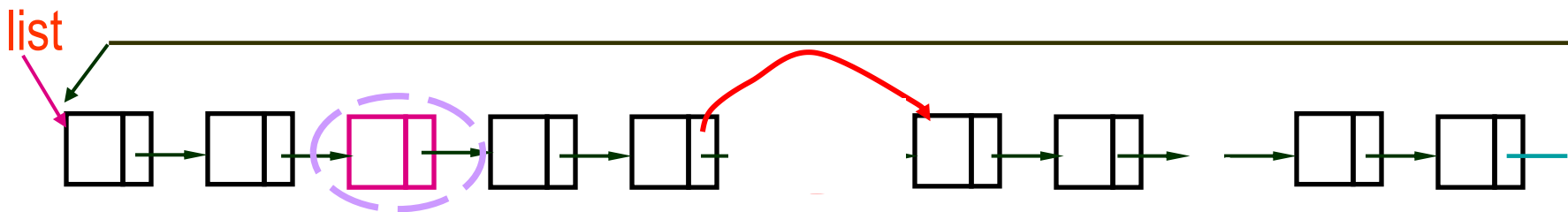
如 $k=3$, $m=4$



例 约瑟夫(JOSEPHU)问题求解

需要做的工**作**：

1. 找到第一个出发点；
2. **反复**删除第m个链结点。



假设 $k=3$, $m=4$

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

$k-1$ 次 $m-1$ 次



例 约瑟夫(JOSEPHU)问题参考实现 (待验证)

```
void josephu( int n, int k, int m ){
```

```
    Nodeptr p,r; //r为p的前驱
```

```
    int i;
```

```
    r = createList(n); //创建循环链表, 此时r指向表尾结点
```

```
    p=r->link; //p指向表头, 此时r为p的前驱
```

```
    //令p指向编号为k的点, r为p前驱
```

```
    for(i=1; i<k; r=p,p=p->link, i++)
```

```
        ;
```

```
    //反复寻找并删除报数为m的点
```

```
    p=deleteM(p, r, m);
```

```
    printf("%3d", p->data);
```

```
}
```

```
Nodeptr deleteM(Nodeptr p, Nodeptr r, int m){
    while(p->link!=p){
        for(i=1;i<m; r=p, p=p->link, i++)
            ;
        r->link=p->link;
        printf("%3d",p->data);
        free(p);
        p=r->link;
    }
    return p;
}
```



例

约瑟夫(JOSEPHU)问题实现——创建链表

```
Nodeptr createList( int n){ //返回循环链表最后一个结点
    Nodeptr head=NULL, p=NULL, q;
    Datatype a;           /* 创建一个空链表 */
    for(i=0;i<n;i++){
        READ(a);          /* 取一个数据元素 */
        q=(NodePtr)malloc(sizeof(Node));
        q->data=a;        q->link=NULL;
        if (head==NULL)
            head =p=q;
        else {
            p->link=q;    //将新结点链接在链表尾部
            p=p->link;
        }
    }
    p->link=head; //构成循环
    return p; //注意： 这里返回链表尾结点
}
```



问题2.3： 显示文件最后n行

- 问题： 命令tail用来显示一个文件的最后n行。其格式为：

```
tail [-n] filename
```

其中：

-n : n表示需要显示的行数，省略时n的值为10。

filename : 给定文件名。

如，命令tail -20 example.txt 表示显示文件example.txt的**最后20行**。

实现该程序，该程序应具有一定的**错误处理能力**，如能处理非法命令参数和非法文件名。



问题2.3： 问题分析

若从命令行输入:

tail -20 test.txt

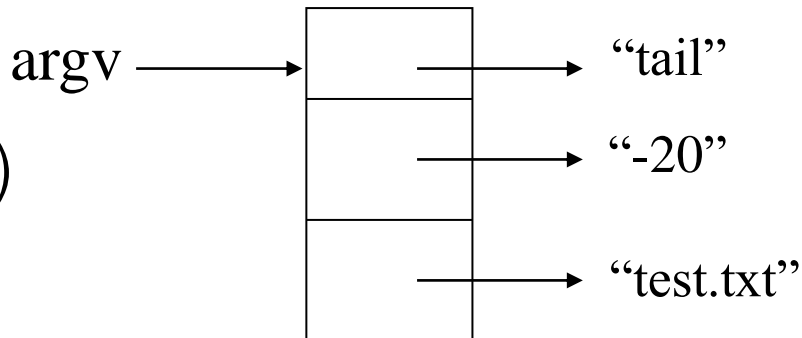
以显示文件test.txt的最后20行。

■ 如何得到需要显示的行数和文件名？

- 使用命令行参数

```
int main(int argc, char *argv[])
```

- 行数 $n = \text{atoi}(\text{argv}[1]+1)$
- 文件名 **filename** = `argv[2]`

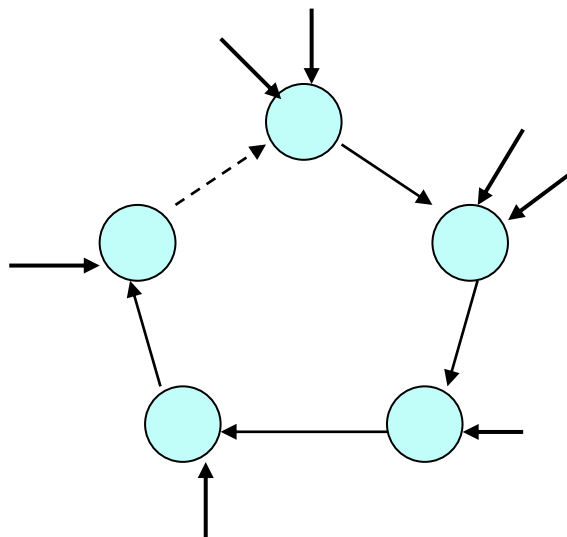


■ 如何得到最后n行？



问题2.3：算法设计

- **方法一：**使用 n 个节点的循环链表。链表中始终存放最近读入的 n 行。
 1. 首先创建一个空的循环链表；
 2. 然后再依次读入文件的每一行挂在链表上，最后链表上即为最后 n 行。





问题2.3: 代码实现 (循环链表)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DEFLINES 10
#define MAXLEN 81
struct Node {
    char *line;
    struct Node *next;
};
```

命令行输入中没有指定打印行数时, 获取文件名, 此时行数为缺省10。如 tail test.txt

```
int main(int argc, char *argv[ ]){
    char curline[MAXLEN], *filename;
    int n = DEFLINES, i;
    struct Node *first, *ptr;
    FILE *fp;
    if( argc == 3 && argv[1][0] == '-' ) {
        n = atoi(argv[1]+1);
        filename = argv[2];
    }else if( argc == 2){
        filename = argv[1];
    }else {
        printf("Usage: tail [-n]
filename\n");
        return (1);
    }
}
```

命令行输入中指定打印行数时, 获取行数及文件名。如 tail -20 test.txt



```
if((fp = fopen(filename, "r")) == NULL){  
    printf(" Can't open file: %s !\n", filename);  
    return (-1);  
}
```

//创建循环链表

```
first = ptr = (struct Node *)malloc(sizeof ( struct Node));  
first->line = NULL;  
for(i=1; i<n; i++){  
    ptr->next = (struct Node *)malloc(sizeof ( struct Node));  
    ptr = ptr->next;  
    ptr->line = NULL;  
}  
ptr->next = first;  
ptr = first;
```

将链表的最后一个节点指向头节点，以构成一个循环链表。



```
while(fgets(curline, MAXLEN, fp) != NULL){
    if(ptr->line != NULL)    /*链表已经满了，需要释放掉不需要的行*/
        free(ptr->line);
    ptr->line = (char *) malloc ( strlen(curline)+1);
    strcpy(ptr->line, curline);
    ptr = ptr->next;
}

for(i=0; i<n; i++) {
    if(ptr->line != NULL)
        printf("%s", ptr->line);
    ptr = ptr->next;
}

fclose(fp);
return 0;
}
```

测试考虑点：准备一个包含内容（如11~20行）的正文文件test.txt

- 1) tail -5 test.txt (正常)
- 2) tail test.txt (正常)
- 3) tail -30 test.txt (非正常)
- 4) tail -0 test.txt (非正常)
- 5) tail -1 test.txt (边界)



问题2.3: 其它方法*

■ 方法2: 两次扫描文件。

- 第一遍扫描文件, 用于统计文件的总行数 M ;
- 第二遍扫描文件时, 首先跳过前面 $M-n$ 行, 只读取最后 n 行。

■ 如何开始第二遍扫描?

`fseek(fp, 0, SEEK_SET);` -- 将文件读写位置移至文件头
或（关闭后）再次打开同一个文件

请同学们考虑是否还有其它方法? 甚至更好的方法?

如: 我们可设计这样两个函数:

`fbgetc(fp);` //从文件尾开始, 每次倒读一个字符

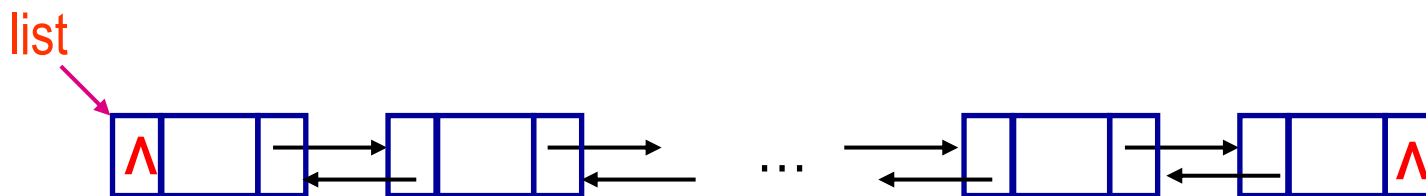
`fbgets(buf, size, fp);` //从文件尾开始每次倒读一行



2.6 双向链表及其操作

本节主要内容

1. 双向链表的构造
2. 双向链表的插入与删除





2.6.1 双向链表的构造

所谓**双向链表**是指链表的每一个结点中除了数据域以外设置两个指针域，其中之一指向结点的直接后继结点，另外一个指向结点的直接前驱结点。

链结点的实际构造可以形象地描述如下：



其中，data 为数据域

rlink, llink 分别为指向该结点的直接后继结点与直接前驱结点的指针域

分别称为右指针和左指针

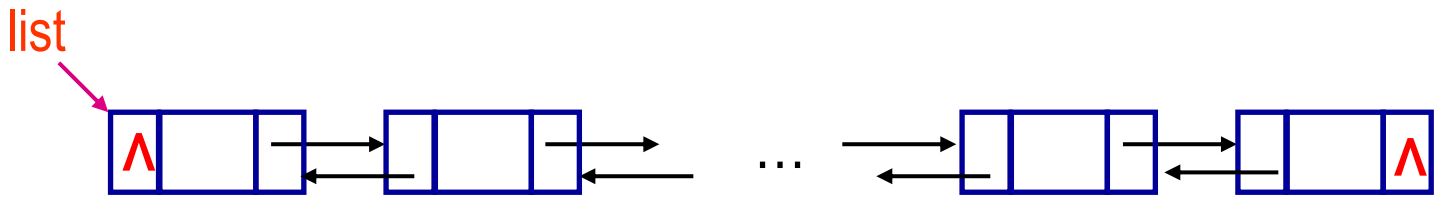


类型定义

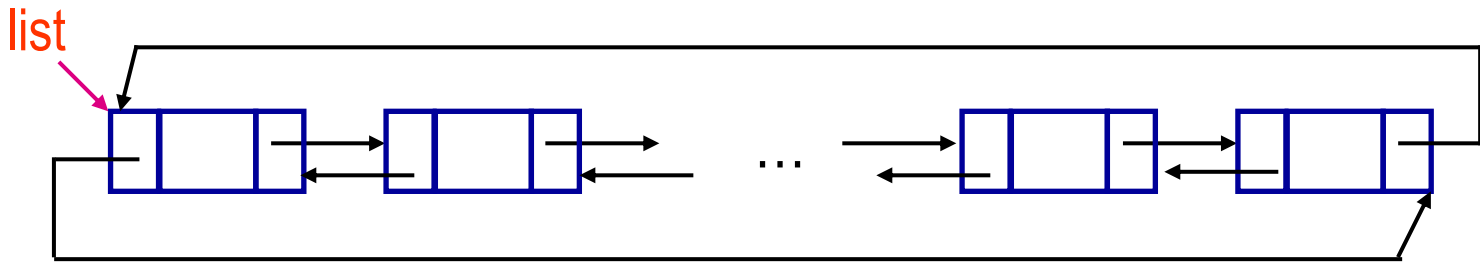
```
struct node {  
    ElemType data;  
    struct node *rlink, *llink;  
};  
typedef struct node *DNodeptr;  
typedef struct node DNode;
```



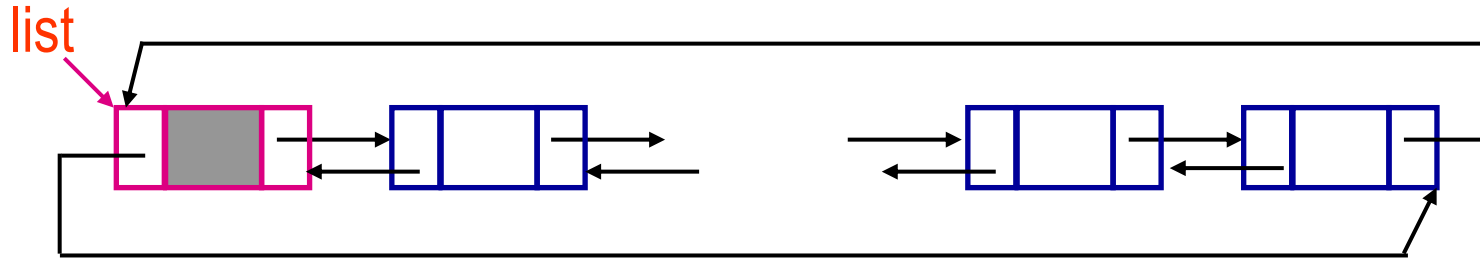
双向链表的几种形式



双向链表



双向循环链表

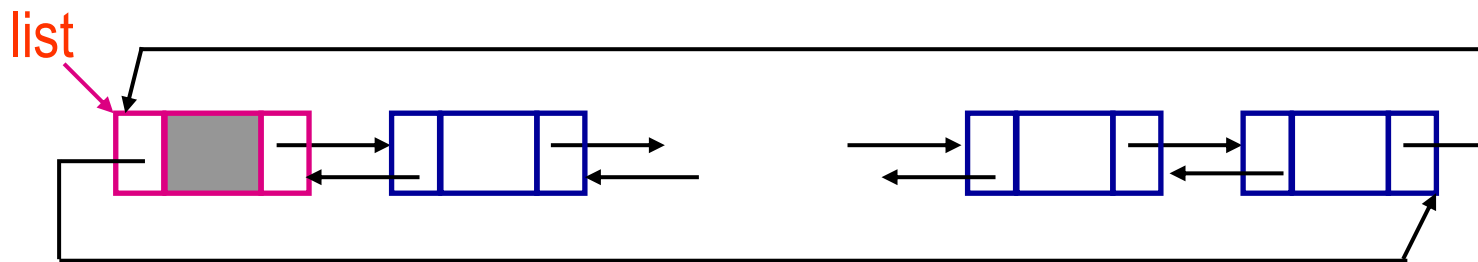


带头结点的双向循环链表



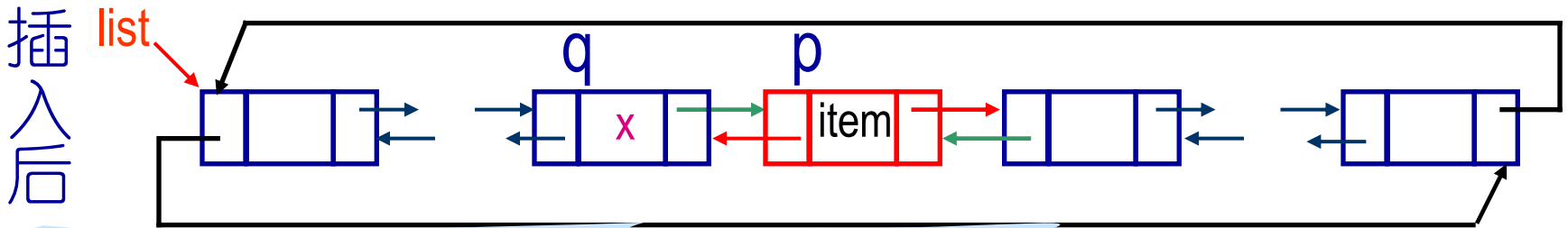
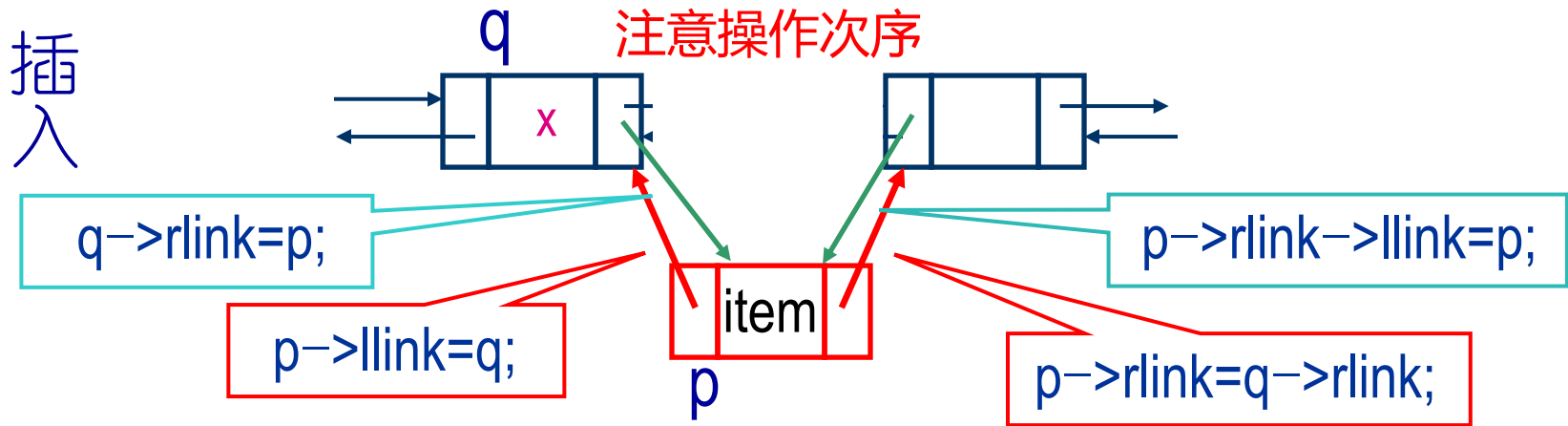
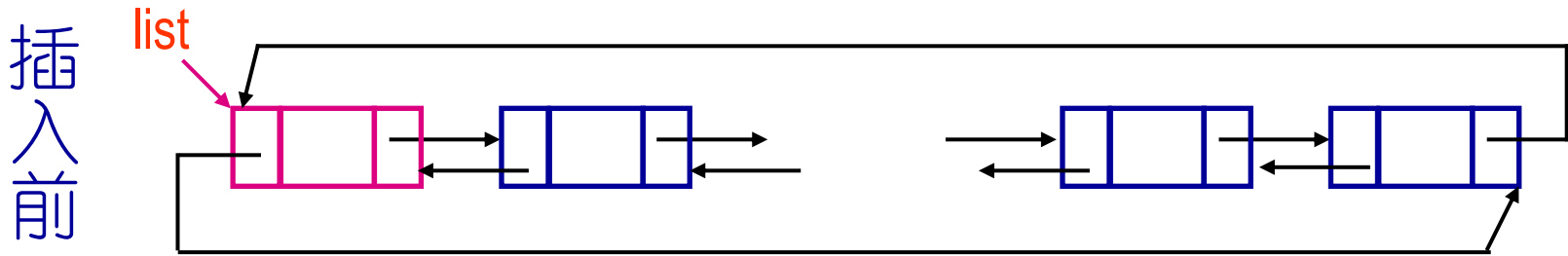
2.6.2 双向链表的插入

功能 在非空双向循环链表中某个数据域的内容为 x 的链结点右边插入一个数据信息为 $item$ 的新结点。



需要做的工作:

1. 找到满足条件的结点;
2. 若找到, 构造一个新的链结点;
3. 将新结点插到满足条件的结点后面。



注意： 在头(第一个)结点前插入一个结点时，步骤如下：

```

p->rlink = list;           p->llink = list->llink;
list->llink->rlink = p;    list->llink = p;
list = p;

```




时间复杂度 $O(n)$

```
int insertDNode(DNodeptr list, ElemType x, ElemType item){
```

```
int DNodeptr p,q;
```

查找满足条件的结点

```
for(q=list; q!=list && q->data!=x; q=q->rlink) /* 寻找满足条件的链结点 */
```

```
if(q==list)
```

```
return -1;
```

/* 没有找到满足条件的结点 */

```
p=(DNodeptr)malloc(sizeof(DNode)); /* 申请一个新的结点 */
```

```
p->data=item;
```

```
p->llink=q;
```

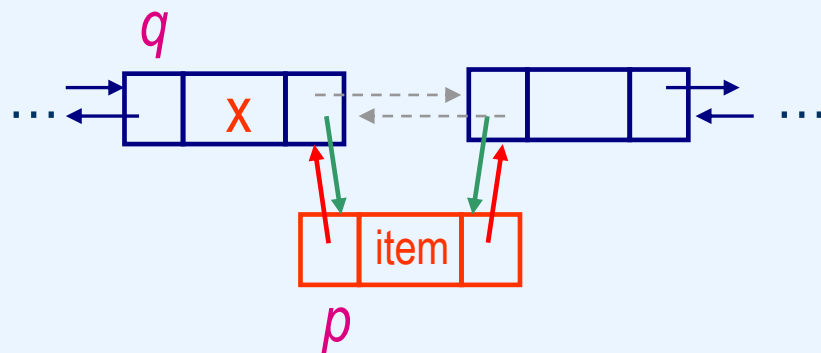
```
p->rlink=q->rlink;
```

```
q->rlink->llink=p; 或 p->rlink->llink=p;
```

```
q->rlink=p;
```

```
return 1; /* 插入成功 */
```

```
}
```





附：常见错误

在非空双向循环链表中由 q 所指的那个链结点前面插入一个由 p 所指的链结点的动作所对应的语句依次为：

$p \rightarrow rlink = q;$

$p \rightarrow llink = q \rightarrow llink;$

$q \rightarrow llink = p;$

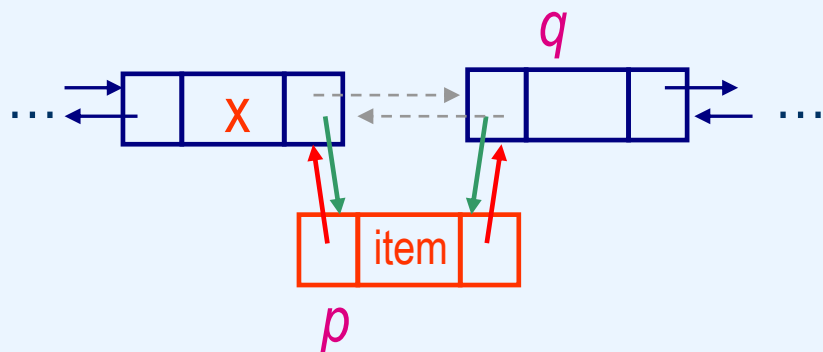
_____ (空白处为一条赋值语句)

A. $q \rightarrow rlink = p;$

B. $q \rightarrow llink \rightarrow rlink = p;$ X

C. $p \rightarrow rlink \rightarrow rlink = p;$

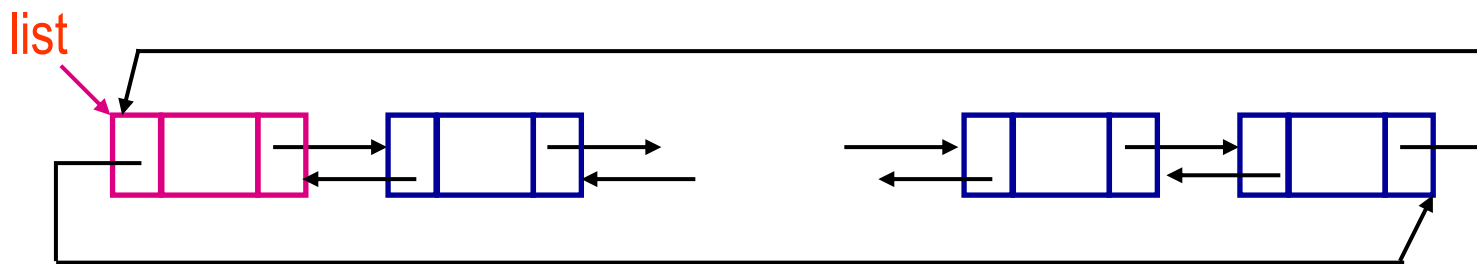
D. $p \rightarrow llink \rightarrow rlink = p;$





2.6.3 双向链表的删除

功能 删除非空双向循环链表中数据域的内容为x的链结点。



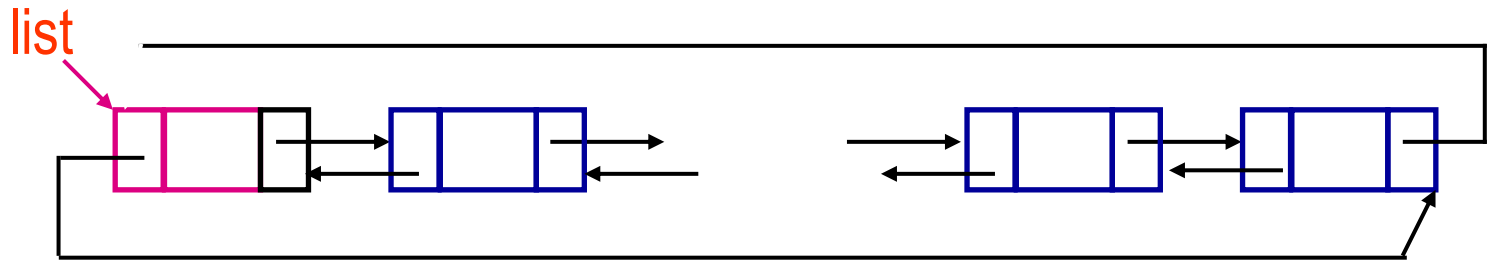
步骤:

1. 找到满足条件的结点；
2. 若找到，删除（并释放）满足条件的结点。

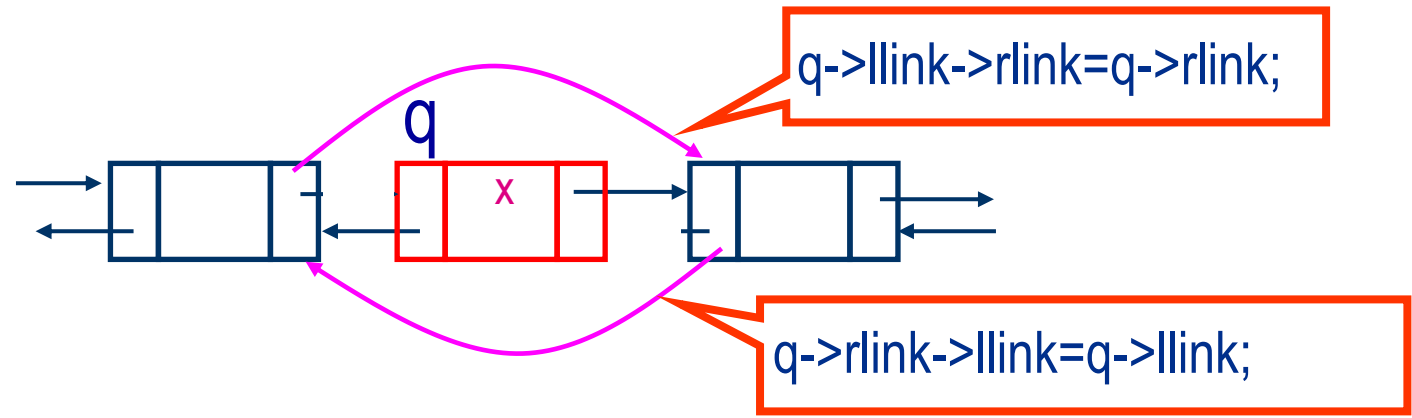


注意：删除头(第一个)结点时，步骤如下：
`list->rlink->llink = list->link;`
`list->llink->rlink = list->rlink;`
`q = list; list = list->rlink; free(q);`

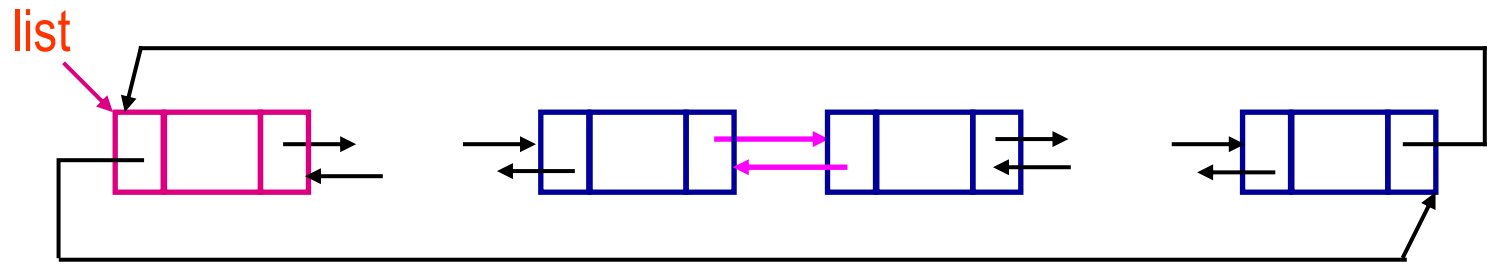
删除前



删除



删除后





时间复杂度 $O(n)$

```
int deleteDNode(DNodeptr list, ElemType x) {  
    DNodeptr q;
```

寻找满足条件的结点

```
    for(q=list; q!=list && q->data!=x; q=q->rlink) /*找满足条件的链结点*/  
        ;
```

```
    if(q==list)
```

```
        return -1;
```

/* 没有找到满足条件的结点 */

```
    q->llink->rlink=q->rlink;
```

```
    q->rlink->llink=q->llink;
```

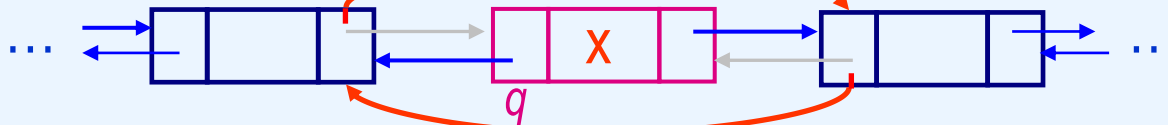
```
    free(q);
```

/* 释放被删除的结点的存储空间 */

```
    return 1;
```

/* 删除成功 */

```
}
```





双向链表

- ✓ 双向链表由于多了前驱结点的指针，使得结点的插入和删除时需要做更多的操作；
- ✓ 双向链表需要保存前驱和后续结点的指针，要比单向链表多占用一些空间；
- ✓ 双向链表由于很好的对称性，使得对某个结点的前后结点访问带来了方便，简化了算法，可以提高算法的时间性能，以空间换时间。



练习

某线性表中最常用的操作是在**最后一个元素之后**插入一个元素和**删除第一个元素**，则采用(_____)存储方式最节省运算时间。

- A. 单链表
- B. 有头和尾指针的单链表
- C. 仅有头指针的单循环链表
- D. 仅有尾指针的单循环链表
- E. 双链表
- F. 双向循环链表



链表应用实例：动态内存管理*

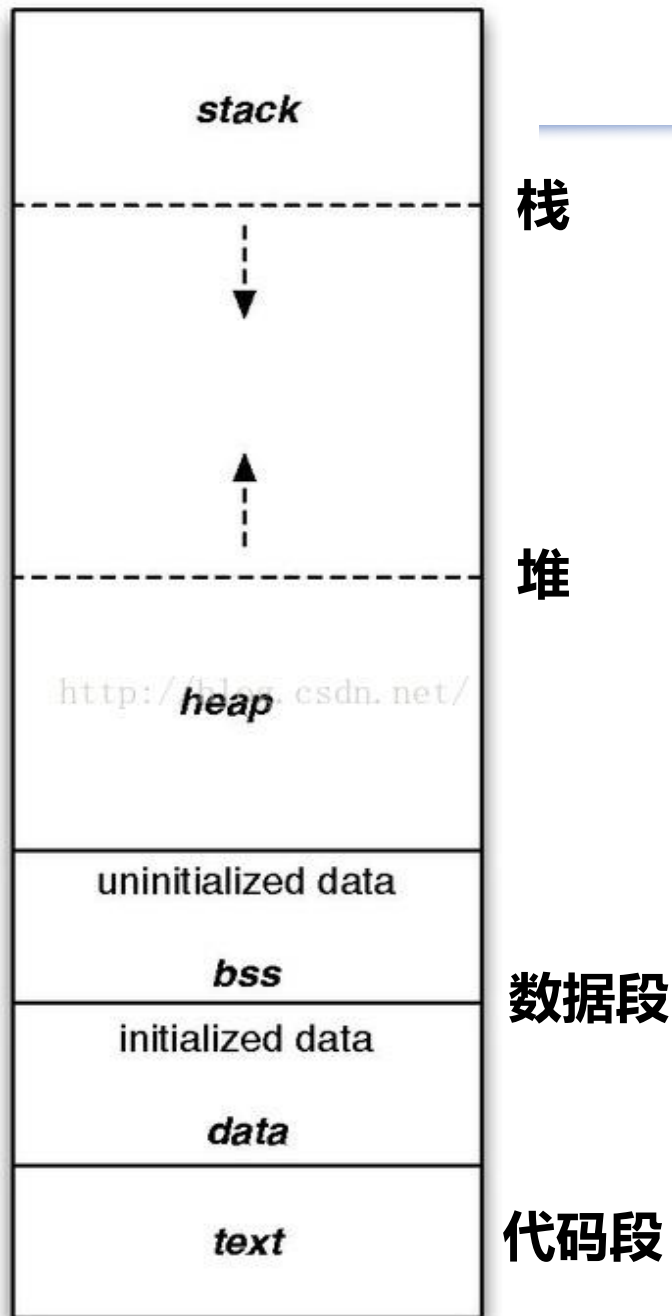
进程对应的内存空间划分：

代码段 (text segment)：又称为文本段。存储可执行文件的指令；也可能包含一些只读的常量，如字符串常量等。操作系统通常将该段的数据页设为只读，防止意外的改写。

数据段 (data segment)：静态存储区。data段存储经过初始化的全局和静态变量。bss段 (Block Started by Symbol) 存储未初始化或初始化为0的全局变量和静态变量。

堆 (heap)：动态管理的内存空间。用户由malloc/calloc申请，realloc调整，free释放。

栈 (stack)：记录程序的执行状态（函数调用等），是局部变量的存储空间。





链表应用实例：动态内存管理*

对于初始状态下的内存来说，整个空间都是一个空闲块（在编译程序中称为“堆”）。但是随着用户不断地提出存储请求，系统依次分配。

U1, U2, ..., U5先后分配给用户



U2, U4先后被释放

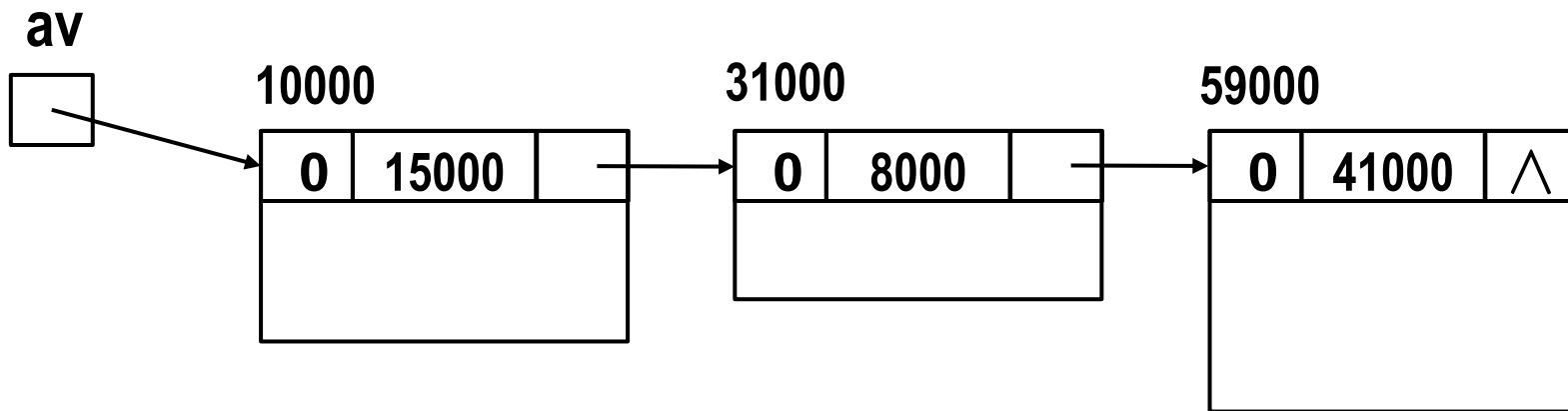




链表应用实例：动态内存管理*

系统一般用一个链表记录所有空闲块信息：

- 链结点直接建立在空闲块上（如低地址的前数个字节）
- 每个结点记录空闲块的块大小和下一空闲块指针等。
- 按照管理方式的不同，链结点可能按块大小或地址大小排序



内存空闲块链表示意



链表应用实例：动态内存管理*

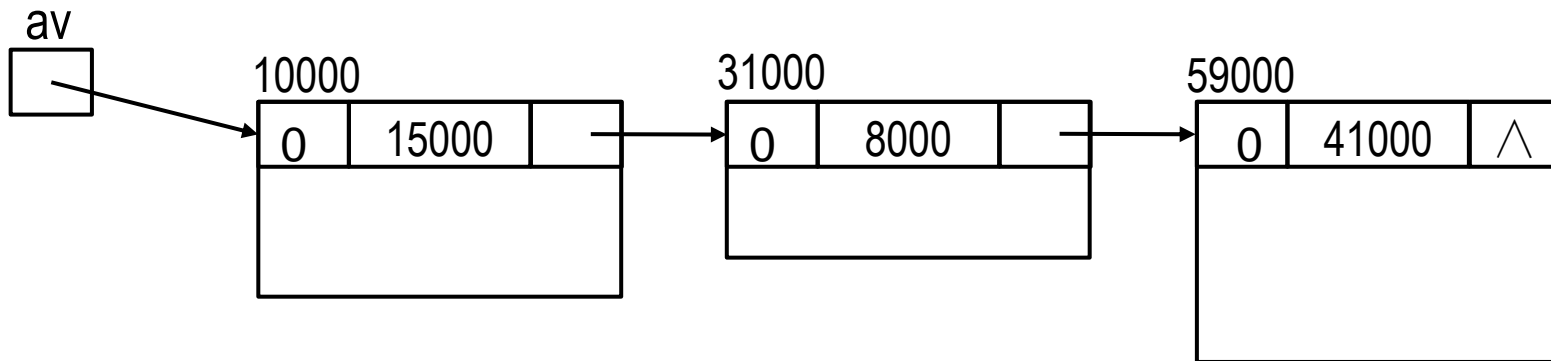
动态内存空间的分配和回收方式： 在用户申请内存空间时，需要从可利用块中找出一个合适的结点，有三种管理方式：

- **首次拟合法：** 在可利用空间表中从头开始依次遍历，将找到的第一个内存不小于用户申请空间的结点分配给用户，剩余空间仍留在链表中；回收时只要将释放的空闲块插入在链表的表头即可。
- **最佳拟合法：** 从不小于用户申请大小的内存块中选择最接近的一个结点分配给用户。为了实现这个方法，首先要将链表中的各个结点按照存储空间的大小进行从小到大排序，由此，在遍历的过程中只需要找到第一块大于用户申请空间的结点即可进行分配；用户运行完成后，需要将空闲块根据其自身的大小插入到链表的相应位置。
- **最差拟合法：** 在不小于用户申请空间的所有结点中，筛选出存储空间最大的结点，从该结点的内存空间中提取出相应的空间给用户使用。为了实现这一方法，可以在开始前先将可利用空间表中的结点按照存储空间大小从大到小进行排序，第一个结点自然就是最大的结点。回收空间时，同样将释放的空闲块插入到相应的位置上。

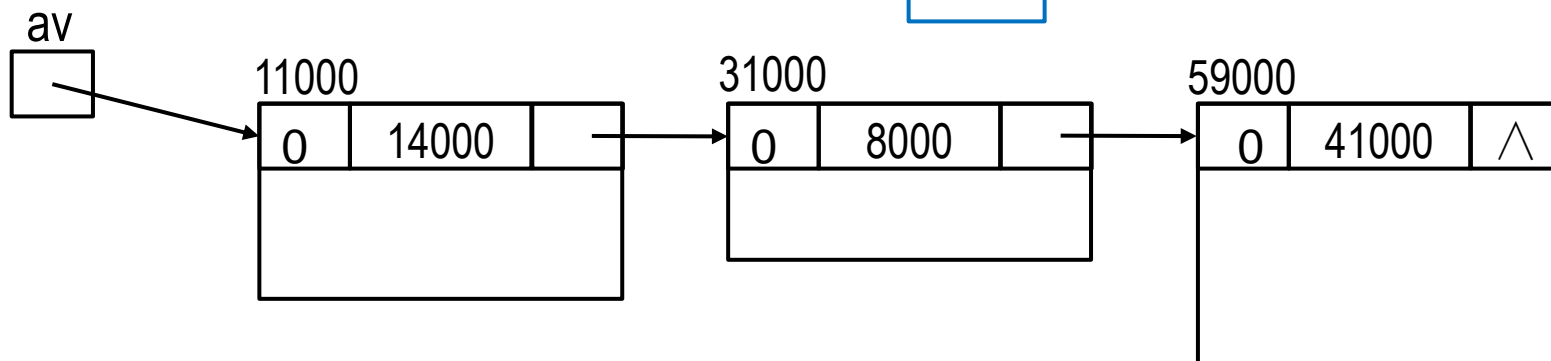
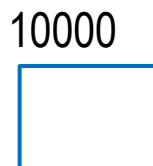
为防止内存空间碎片化，在回收时需考虑将地址相邻的空闲块合并。



链表应用实例：动态内存管理*



申请(首次拟合法): 申请1000字节。
若成功则返回空闲块地址



思考: 如何模拟动态内存管理?

(定义链表, 提供malloc, realloc和free函数)



本章内容小结

线性表

线性表的基本概念

- 什么是线性关系？
- 什么是线性表？
- 线性表的基本操作有哪些？其中最主要的有哪些？

线性表的顺序存储结构

- 构造原理。
- 插入、删除操作对应的算法设计。
- 特点(优点、缺点)。

线性表的链式存储结构

- 线性链表、循环链表和双向链表的构造原理。
- 各种链表中进行插入、删除操作对应的算法设计。
- 头结点问题。



结束!



附：动态内存使用中常见的**错误**

对NULL指针的解引用操作

```
1 void test()
2 {
3     int* p = (int*)malloc(INT_MAX / 4);
4
5     //没有进行判空，就直接使用空间
6     *p = 20;//如果p的值是NULL，就会有问题
7     free(p);
8 }
```

```
1 void test()
2 {
3     int* p = (int*)malloc(INT_MAX / 4);
4     if (p==NULL){
5         exit(EXIT_FAILURE);//异常处理并退出
6     }
7     *p = 20;
8     free(p);
9 }
```



附：动态内存使用中常见的错误

对动态开辟空间的越界访问

```
1 void test()
2 {
3     int i = 0;
4     int* p = (int*)malloc(10 * sizeof(int));
5     if (NULL == p)
6     {
7         exit(EXIT_FAILURE);
8     }
9     for (i = 0; i <= 10; i++)
10    {
11        *(p + i) = i; //当i是10的时候越界访问
12    }
13    free(p);
14 }
```




附：动态内存使用中常见的错误

动态开辟内存忘记释放（内存泄漏）

```
1 void test()
2 {
3     int* p = (int*)malloc(100);
4     if (NULL != p)
5     {
6         *p = 20;
7     }
8 }
9
10 int main()
11 {
12     test();
13     while (1);
14 }
```



附：动态内存使用中常见的错误

对非动态开辟内存使用free释放

```
1 void test()  
2 {  
3     int a = 10;  
4     int *p = &a;  
5     free(p);  
6 }
```

使用free释放一块动态开辟内存的一部分或重复释放

```
1 void test()  
2 {  
3     int* p = (int*)malloc(100);  
4     p++;  
5     free(p); // p不再指向动态内存的起始位置  
6             //这样做一般编译器也会报错的  
7     free(p); //重复释放, 一般来说, 编译器会报错  
8 }
```



附：动态内存使用中常见的错误

操作释放后的指针

```
1 void Test(void)
2 {
3     char* str = (char*)malloc(100);
4     strcpy(str, "hello");
5     free(str);
6     if (str != NULL)
7     {
8         strcpy(str, "world");
9         printf(str);
10    }
11 }
```