

数据结构

Data Structure

2017年秋季学期
刘鹏远

链式存储结构

线性表之链表

用任意存储单元存储线性表的数据元素
保持线性表元素间的逻辑上的相邻关系
存储单元可连续可以不连续，物理相邻信息直接利用不上

其余一些基本概念参见教材

线性表之链表

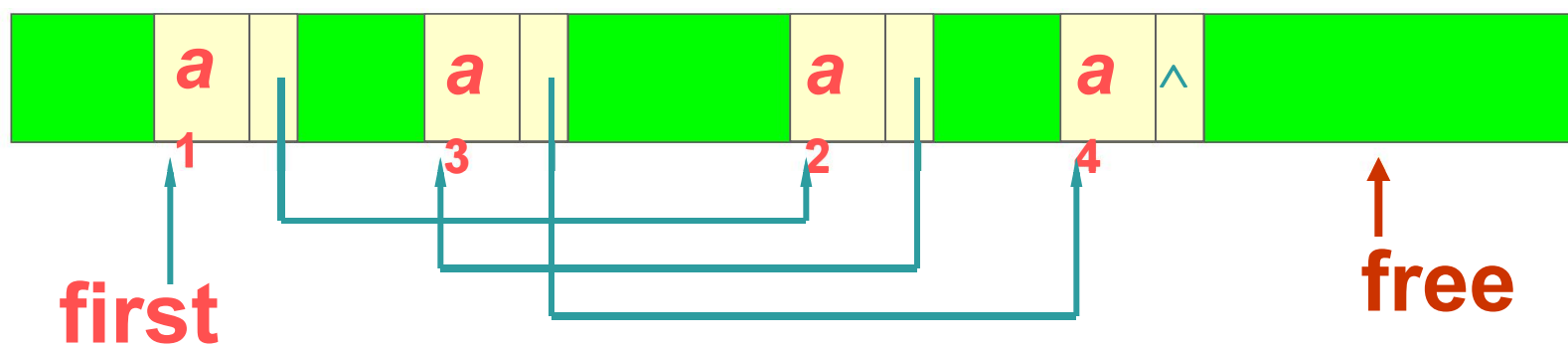
需要额外表示相邻关系的数据

链式存储节点=

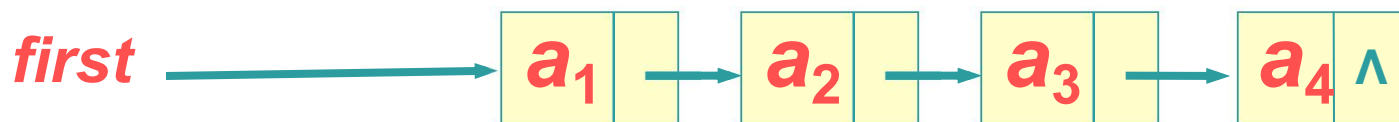
数据元素(Elem_type)+相邻关系信息(*)

单链表

○单链表：所有节点均只含有一个指针域的链表。

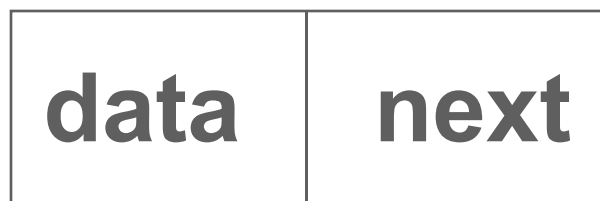


单链表内存存储示意
对应单链表示意



单链表

节点的存储结构



data : 数据域, 存放结点的值。

next : 指针域, 存放结点的直接后继的地址。

也有next域写成link域的写法。

单链表

```
typedef struct node
{
    Elem_type data;          /*数据域，保存结点的值 */
    struct node *next;      /*指针域*/
}list_node;                /*结点的类型 */
```

- 1、定义是递归的
- 2、指向该结构的指针即可得该节点及之后的节点(如有)
- 3、整个单链表可用一个指向第一个节点的指针来标识/定义

单链表

节点的基本操作（各类数据结构教程均未封装）

1、声明空节点指针(空链表，长度为0)

```
list_node *my_node = NULL;  
my_node → NULL
```

2、分配节点(动态内存分配)，指向之

```
my_node = (list_node*)malloc(sizeof(list_node));  
my_node → 

|     |   |
|-----|---|
| 随机数 | → |
|-----|---|


```


单链表

节点赋值

```
my_node->data = 20;
```

```
my_node->next = NULL;
```



销毁节点（释放内存）

```
free(my_node);
```

my_node → NULL

单链表

- 单链表的存储结构定义

```
typedef list_node* link_list;    //单链表
```

- 基本操作的实现(实现线性表ADT操作)

- 初始化，插入元素，建表，清空，遍历
- 查找某元素位置，删除元素，求长度
- 销毁，判断空表，取表中第i个元素等。

单链表

取某位置元素

分析：

- 非随机存取，无法直接利用位置
- 方法：

从第一个节点开始，依次找向下一个节点
当找到第 i 个节点时，返回节点值
如链表长度不足 i ，则返回ERROR

单链表

```
Status get_elem_eval(link_list first, int i, Elem_type *e){  
    if(!first) return ERROR;  
    int k=1; link_node *p = first;  
    while(k<i && p){p = p->next;    k++;}  
    if (!p) return ERROR;  
    else *e = p->data;  
    return OK;  
} //大O
```

单链表

插入元素(给定插入位置i)

分析：

- 与顺序表相同，插入元素一般默认在某位置/某元素的前方插入，插入后的元素占据了原来元素的位置
- 思考：给定链表中某节点指针p（代表在第i个位置的原节点），给定新节点q，q插入到p前还是p后实现起来是否有区别？

单链表

基本方法：

- 对在某节点前插入新节点
- 找到其前一个节点位置
- 在该节点后插入
- 保持原链表不断续

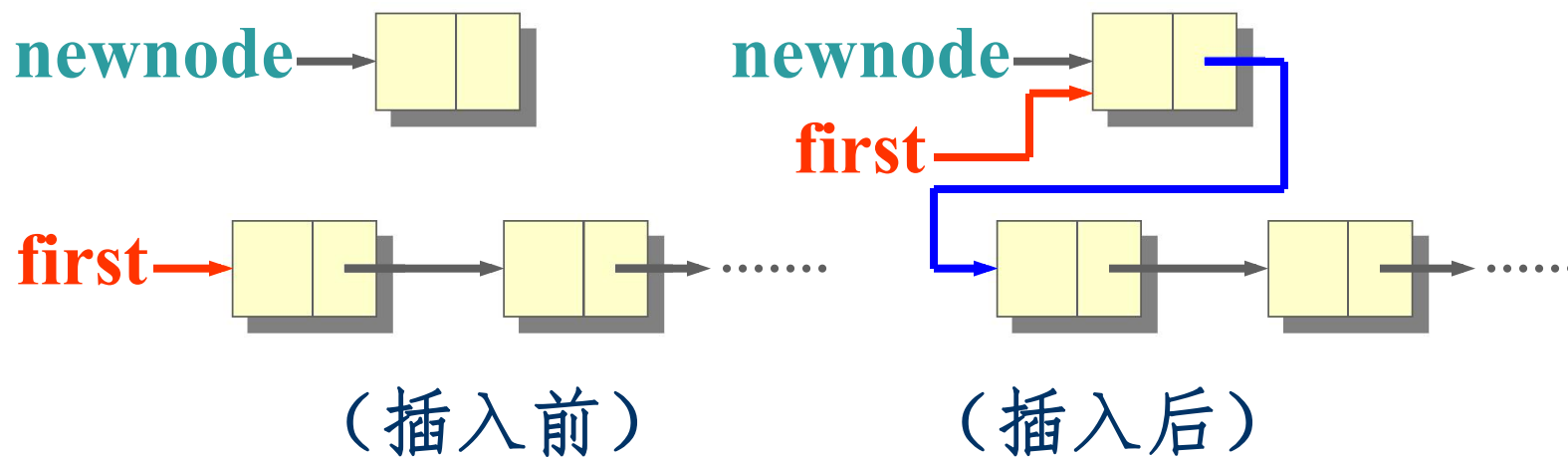
单链表

- 第一种情况：在第 1 个结点前插入：

`newnode->next = first ;`

`first = newnode;`

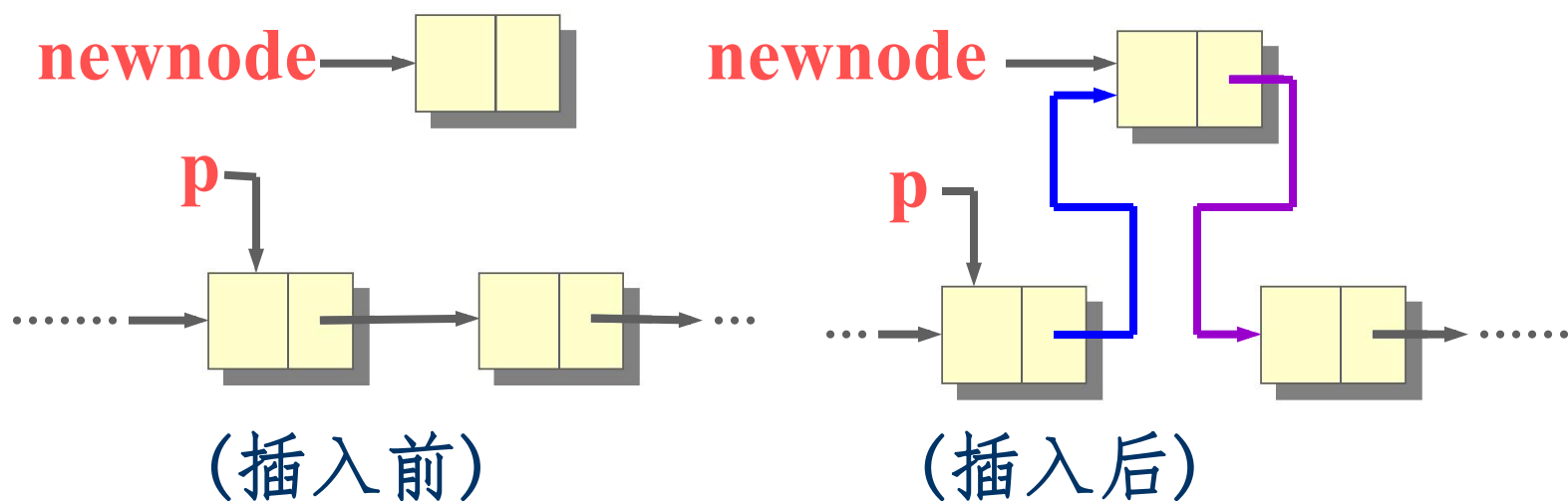
//修改first自身！



单链表

- 第二种情况：在链表中间插入：
首先定位指针 p 到插入位置，再将新结点插在其后：

$\text{newnode} \rightarrow \text{next} = p \rightarrow \text{next};$
 $p \rightarrow \text{next} = \text{newnode};$

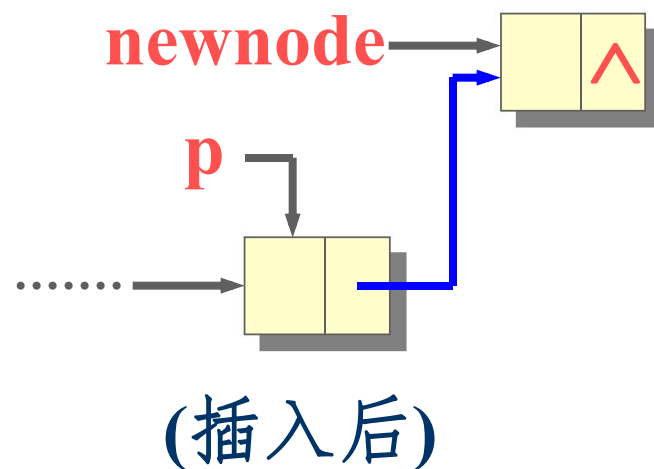
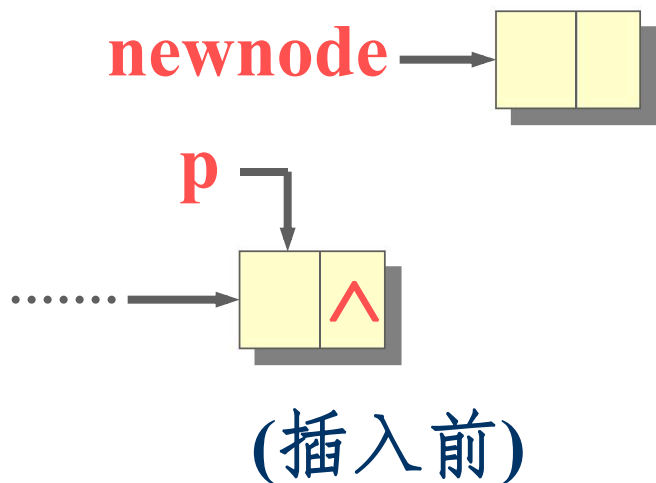


单链表

- 第三种情况：在链表末尾插入：
首先定义指针 p 到尾结点位置，再将新结点
插在其后，新结点成为新的尾结点。

$\text{newnode} \rightarrow \text{next} = p \rightarrow \text{next};$

$p \rightarrow \text{next} = \text{newnode};$



单链表

```
bool Insert ( link_list *first, Elem_type x, int i ) {  
    //在链表第 i ( $\geq 1$ ) 个结点处插入新元素 x  
    link_list * p = *first;  int k = 1;  
    while ( p && k < i-1 )  { p = p->next; k++; }  
                                //找第 i-1 个结点, 此时, k=i-1  
    if ( !p && *first ) {  
        printf ( “无效的插入位置!\n” );  
        return ERROR;           //给的 i 太大  
    }  
}
```

单链表

//创建新结点

```
list_node * newnode =
```

```
    (list_node*)malloc(sizeof(list_node));
```

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

```
if (i == 1) {    //插在表前端，是不是空链表均可
```

```
    newnode->next = *first;
```

```
    *first = newnode;
```

```
}
```

单链表

```
else if(first){           //插在表中或末尾
```

```
    newnode->next = p->next;
```

```
    p->next = newnode;
```

```
}
```

```
else
```

```
    return ERROR; //空链表，插入位置非1
```

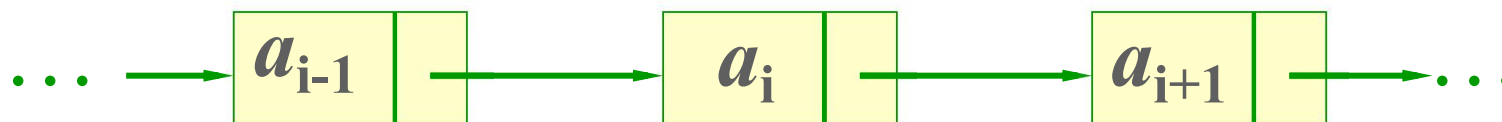
```
    return OK;
```

```
}查找位置 $O(n)$ ，插入 $O(1)$ ，空间 $O(1)$ 
```

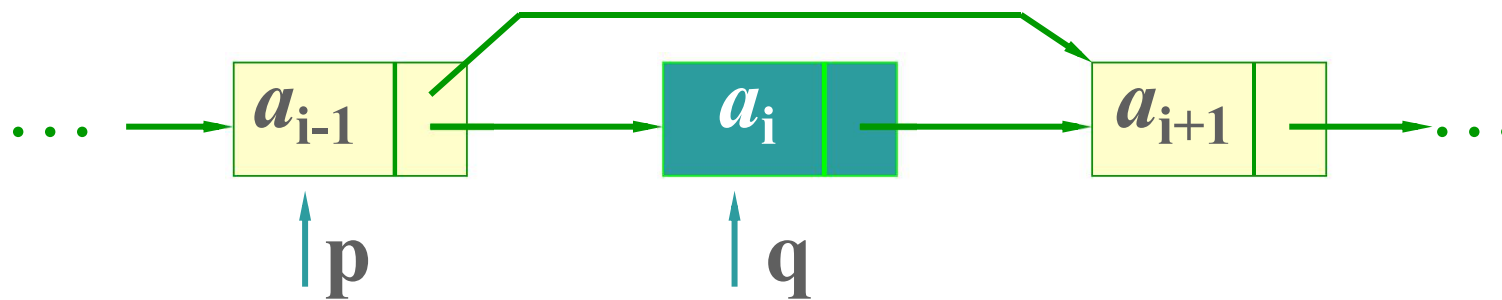
单链表

○删除

- 第一种情况: 删除表中第 1 个元素 (改变指向链表第一个元素指针)
- 第二种情况: 删除表中或表尾元素



删除前



删除后

在单链表中删除含 a_i 的结点

单链表

```
Status Delete (link_list *first, int i, Elem_type *x) {  
    //删除指定位置元素，并返回该元素值  
    link_list *p, *q;  
        //p指向操作节点，q指向被删除节点  
    if (i == 1){ //删除表中第 1 号结点  
        q = *first; //先保留节点指针  
        *first = (*first)->next;  
    }
```


单链表

```
else {
```

```
    p = *first; int k = 1;
```

```
    while ( p && k < i-1 ) {
```

```
        p = p->next;
```

```
        k++;
```

```
    }//找到第 i-1 个结点
```

单链表

```
if ( p || p->next ) { //无i-1/i节点
    printf ( “无效的删除位置!\n” );
    return 0;
}
else { //删除表中或表尾元素
    q = p->next; //保留被删除节点指针
    p->next = q->next;
}
}
```


单链表

```
x = q->data;    //返回被删除元素值  
free(q);        //删除q  
return OK;  
}查找元素 $O(n)$ , 删除 $O(1)$ 
```

总体而言，插入删除元素等比较繁琐。
单链表其他操作课后自行实现。

单链表

带头结点的单链表实现基本操作相对方便。

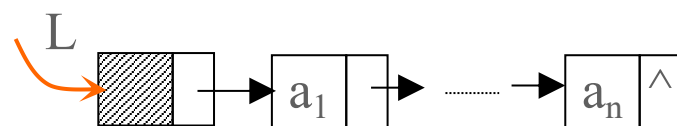
对带头结点的链表，为避免混淆，对一些概念做如下区分：

头结点-----Head 节点

头结点指针-----指向Head节点的指针

首元素指针----指向头结点直接后继（节点）---首元素节点

末元素指针---指向尾节点直接前驱（节点）---末元素节点



英文"head, first, last, tail", 对应中文的"头、首、第一、尾、末、最后"等二义性。

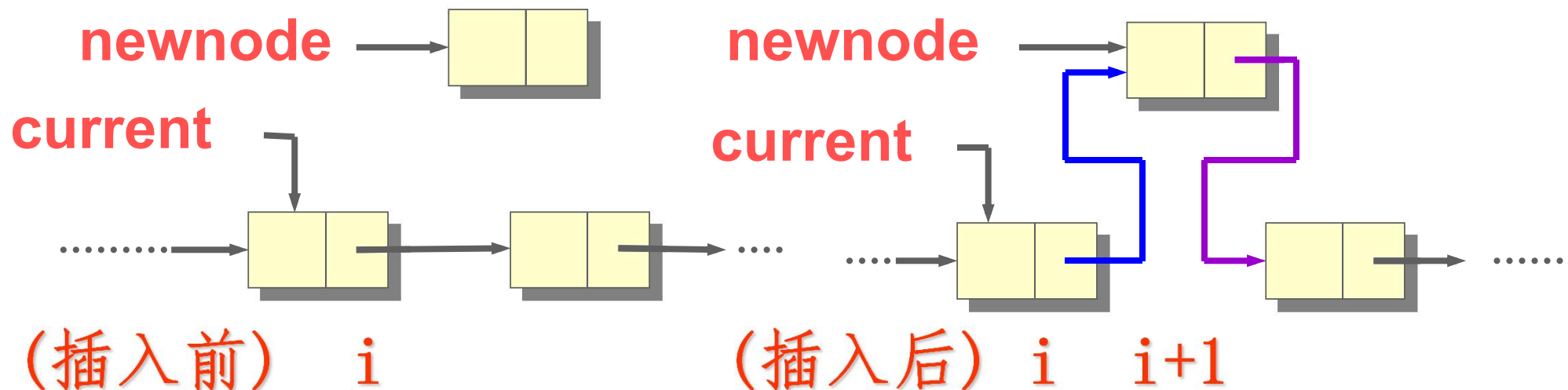
单链表

带不带头结点，是对链表存储结构/操作实现时可选的一种约定。

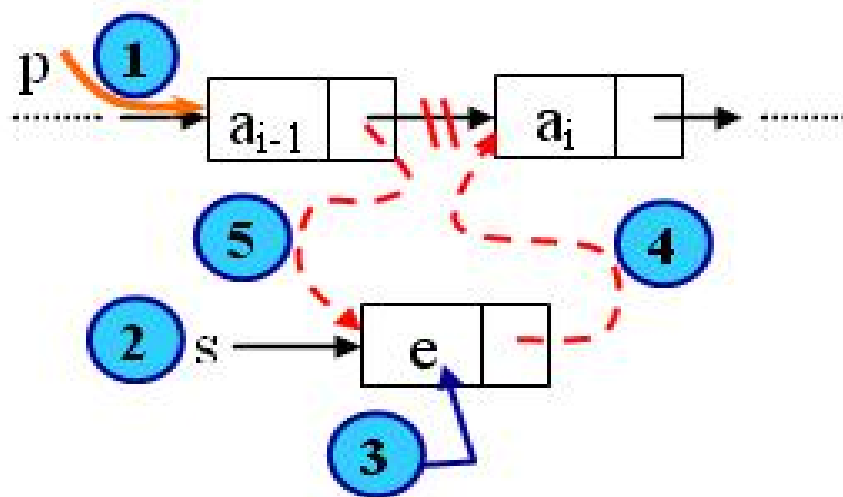
```
Status init(link_list *my_list){  
    *my_list = (link_list)malloc(sizeof(link_node));  
    if(*my_list == NULL)        exit(OVERFLOW);  
    (*my_list)->next = NULL;  
    return OK;  
}  
//带头节点的单链表初始化。
```

可简单的在实现单链表init时，初始化一个头结点，也可以P37重定义

单链表



1. 找到current节点 (i-1位置)
 2. 生成节点new_node
 - 3, 4. new_node赋值
 5. current->next = new_node
- 时间复杂度 $O(n)$
还须判断边界/内存分配是否成功



单链表

Status insert(link_list L,int i, ElemType e)

```
{    link_list p = L;    int j=0;//一般均copy下L指针，不直接操作之
```

```
while(p&& j<i-1) {p = p->next; j++; }
```

```
if(!p || j>i-1) return ERROR; //(右、左越界)
```

```
link_list new_node = (link_list)malloc(sizeof(link_node));
```

```
new_node->data = e; new_node->next = p->next;
```

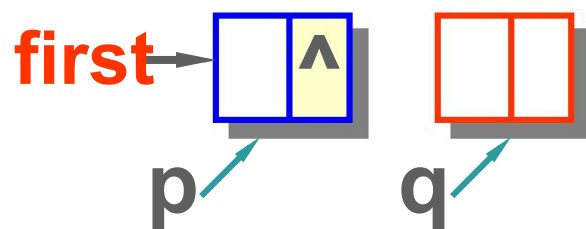
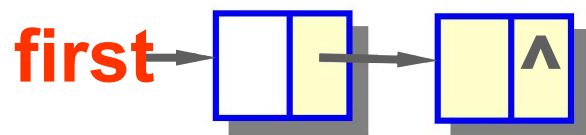
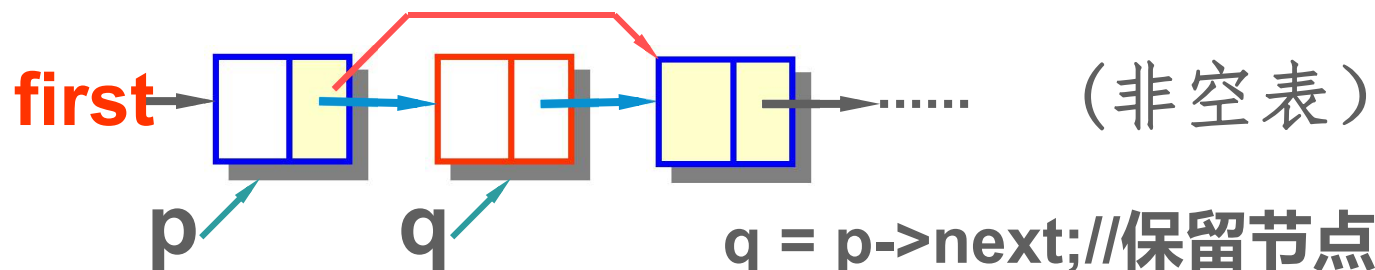
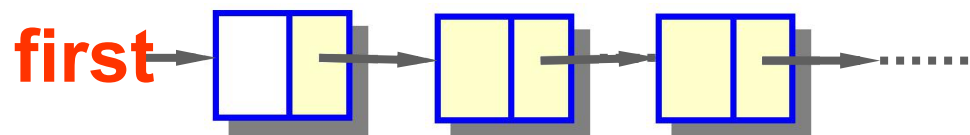
```
p->next = new_node;
```

```
return OK;
```

```
}//O (n)    寻找i-1节点 插入O (1) , 空间O(1)
```

单链表

删除元素



```
q = p->next; //保留节点  
p->next = q->next; //改链  
x=q.data; //保存值  
free(q); //删除  
时间复杂度O ( n )  
(空表)
```


单链表

```
Status delete_index(link_list L,int i, ElemType *e)//给定位置，删除
{
    link_list p=L, q;   int j=0;
    while(p&&j<i-1)  {p = p->next;    j++;  }
    if(!p||!p->next||j>i-1)    return ERROR;
    q=p->next; p->next = q->next;
    *e = q->data;
    free(q);
    return OK;
}//找节点位置O（n），插入O（1）//删除给定值节点操作自行实现
```

单链表

取第i个元素的值

```
Status get_elem_eval(link_list L, int i, Elem_type *e)
```

```
{
```

```
    //自行实现
```

```
    //大O
```

```
}
```


单链表

取第i个元素的地址（指针）

```
Status get_elem_ptr (link_list L,int i, link_node*p)
```

```
{
```

```
    //自行实现，如没有，则返回NULL
```

```
    //大O
```

```
}
```

//可用此操作简化前面插入/删除操作等操作

单链表

按值查找某元素第一次出现位置

1. int find (link_list L, Elem_type e)

//位置序号

2. list_node * find (link_list L, Elem_type e)

//指向某节点的指针

//O(n), 自行实现

单链表

单链表生成 头插法

Status creat_from_head (link_list L) //类C伪码

```
{    while(condition)           //condition为条件表达式，自行定义
    {
        scanf("%d", &e);
        insert(L, 1, e);
    }
    return OK;
```

//L为带头节点的空单链表， $O(n)$ ， n 为节点个数

//新节点总是作为链表首元素节点，结果为输入的倒序。如何解决？

单链表

- 1、倒序输入节点；
- 2、采用尾插法

Status append(link_list L, Elem_type e)

//该操作是单链表末尾插入元素的操作

//找到最后一个非空节点p

//新生成一个节点q，赋值q->data = e;q->next=p->next;

//链接p->next = q;

单链表

假设采用append函数，仿照头插法及insert函数则：

Status creat_from_tail (link_list L) //类C伪码

```
{    while(condition)
    {    scanf("%d", &e);
        append(L, e);
    }
```

```
    return OK;
```

```
}//L为带头节点的空单链表
```

```
//算法时间复杂度O ( ? )
```

单链表

因此，需要保存记录最后一个节点指针

.....

```
//先找到最后一个节点指针current
while(condition) //condition条件自行定义
{
    scanf("%d", &e);
    new_node = (list_node*)malloc(sizeof(list_node));
    if(new_node == NULL) return ERROR;
    new_node->data = e; new_node->next = NULL;
    current->next = new_node;
    current = new_node;
}
```

.....

链表与顺序表比较

- 顺序表数据元素存储时物理地址相邻
 - 一般用数组实现
- 链表数据元素存储物理地址**不要求一定**相邻
 - 一般利用指针

链表与顺序表比较

- 基于空间的比较

- 存储分配的方式

- 顺序表的存储空间是静态/动态分配的

- 链表的存储空间(一般来说)是动态分配的

- 存储密度

存储密度=结点数据本身所占的存储量/结点结构所占的存储总量

- 顺序表的存储密度 = 1

- 链表的存储密度 < 1

链表与顺序表比较

- 实际空间分析

- 顺序表未利用空间

- $$= (\text{MAX_SIZE} - \text{DATA_LENGTH}) * \text{sizeof}(\text{ElemType})$$

- 链表未利用空间

- $$= (\text{ELEMENT_SIZE} - \text{DATA_SIZE}) * \text{DATA_LENGTH}$$

链表与顺序表比较

- 基于时间的比较
 - 存取元素
 - 给定位置 i ，顺序表可以随机存取（效率高） $O(1)$
 - 给定位置 i ，链表无法随机存取（效率低） $O(n)$
- 插入/删除时移动元素个数
 - 顺序表平均需要移动近一半元素（效率低） $O(n)$
 - 链表不需移动元素，只需修改指针（效率高） $O(1)$

链表与顺序表比较

各有优劣，不同情形下用不同的结构

思考1：生成链表后，再利用单链表尾插入操作来插入多个数据时间复杂度高，因访问尾元素节点 $O(n)$

思考2：从任意节点开始遍历整个单链表较麻烦

思考：是否可以在现有逻辑及链表存储结构下，解决上述问题？

上机与作业

带头结点的单链表实现，与顺序表类似：

1、上机实现主要功能与验证

- 1、实现定义
- 2、主函数建立一个单链表
- 3、头插法插入10个整数，删除1,3,5位置的整数，位置7插入整数7
- 4、求此时表长
- 5、遍历此时的链表表

2、模课上交时间照常(仅交上机实现部分，其余部分下次上机继续)