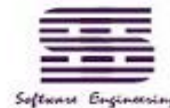


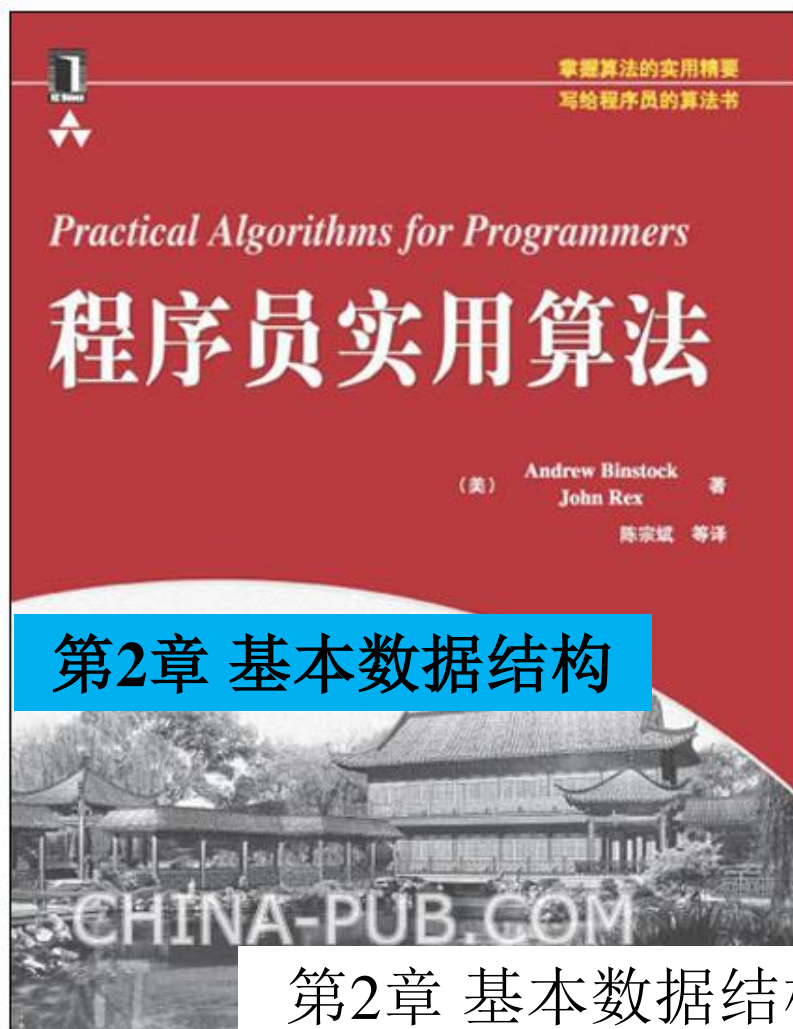
# 实用算法设计——线性表

主讲：余艳玮

[ywyu@ustc.edu.cn](mailto:ywyu@ustc.edu.cn)



# 参考资料



2019/9/28



# 本章重难点

- 重点：
  - 理解线性表的逻辑特点；
  - 分别掌握线性表两种存储结构的特点、实现（C定义）及其基本操作的实现、适用范围；
  - 掌握算法的描述方法：伪代码和算法流程图；
  - 会粗略分析算法的时间复杂度（三种情况下）和内存开销。
- 难点：理论的落地应用。对于给定的应用需求，
  - 会判断是否需要用到DS？
  - 会判断是否适合用线性表来刻画同类数据之间的关系？（逻辑结构）
  - 会设计线性表的存储结构（包括：存储哪些数据（包括数据类型和取值）？顺序存储/链式存储的选择？）（数据+存储结构）
  - 会编码实现数据的存储结构；
  - 会利用线性表的基本操作来解决问题。

# 2 线性表

## 2.1 数据结构

- 逻辑结构、存储结构/物理结构

## 2.2 什么是线性表？

## 2.3 顺序表的定义及实现

## 2.4 链表的定义及实现

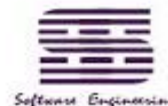
- 单链表、双向链表

## 2.5 案例分析

## 2.6 顺序表的延伸：位向量/位图



2019/9/28

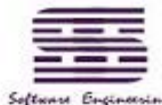


## 2.1 数据结构

- 数据结构：性质相同的数据元素的有限集合及其上的关系的有限集合。（数据+结构）
- 是描述现实世界实体的数据模型及其上的操作在计算机上的表示和实现。
- 数据结构包括逻辑结构和存储结构/物理结构。



2019/9/28



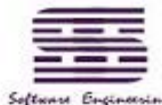
# 2.1 数据结构——逻辑结构

- 逻辑结构：
  - 从逻辑关系上对数据结构进行描述
  - 是从具体问题抽象出来的数据模型
  - 与数据本身的存储（数据元素的存储位置、类型和具体取值）无关。

如：线性结构：线性表；  
非线性结构：树、图；



2019/9/28



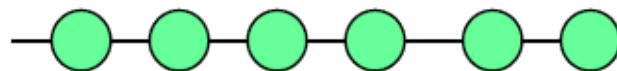
## • 四类数据结构（D+S）：

### – 集合结构：

- 元素间无任何关系，即关系集合是空集： $S=\{\}$

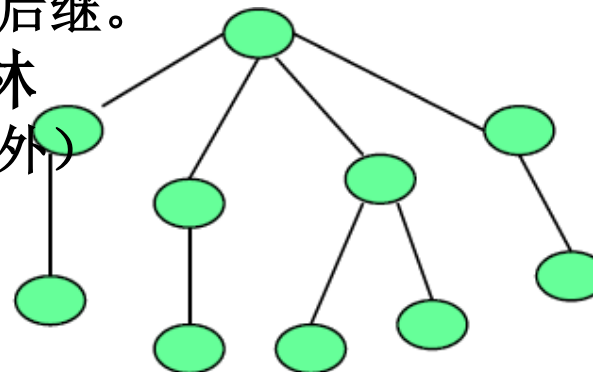
### – 线性结构：如，线性表。

- 元素间的关系是1:1
- 除头结点外，所有结点有且仅有一个直接前驱；
- 除尾结点外，所有结点有且仅有一个直接后继。



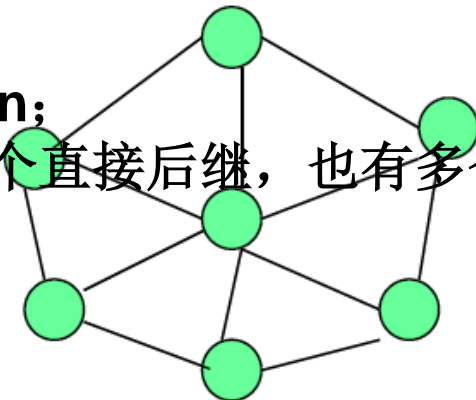
### – 树形结构：如，一般树、二叉树、森林

- 一个结点可有多个直接后继（除叶子结点外）
- 但只有一个直接前驱（除根结点外）；



### – 图形结构：

- 元素间的关系是m:n；
- 一个结点可以有多个直接后继，也有多个直接前驱。



# Fun Time



2019/9/28



Software Engineering



## 2.1 数据结构——存储结构/物理结构

- 数据的存储结构/物理结构：
  - 数据结构在计算机中的表示（或映像）
    - 顺序映像：顺序存储结构
    - 非顺序映像：链式存储结构
    - 如，线性表有顺序表和链表两种物理存储方式
  - 它可以借助于具体某程序语言中的“数据类型”来定义它。也可采用**typedef**将类型名重命名，以增加代码的可读性。

- `int Sqlist[100];`

- `struct Node{  
    int data;  
    struct Node *next`

`};``Typedef struct Node *Link;``Link head;`

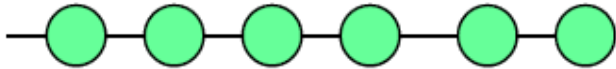
“顺序表”数据存储结构的实现；  
描述了100个int型变量组成的集合，且隐含着可利用下标[]来描述两个int型变量间的联系。

“单链表”数据存储结构的实现；  
隐含着可利用指针next来描述两个struct Node类型的变量之间的联系。



2019/9/28

## 2.2 线性表

- 什么是线性表？
  - 有限个数据元素组成的序列，记作 $(a_1, a_2, \dots, a_n)$
- 逻辑特征：线性特征 
- 存储结构：顺序表和链表两种方式
- 基本操作：
  - 创建空的线性表
  - 销毁已有线性表
  - 查找直接后继和直接前驱
  - 插入一个元素
  - 删除一个元素

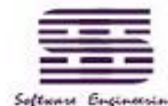


# 课堂练习1

- 问题1：读入一个包含城市和气温信息的数据文件，要求将记录按照气温和城市名称升序地插入到一个链表中，丢弃重复的记录，然后打印该有序链表，并指示位于中间的条目。接着，逐渐缩短链表并重新打印显示中间条目。其中，文件中每行为一条数据记录，它的前3个字符表示气温，其后的最多124个字符表示城市名称。如“-10Duluth”。
- 请指出上述问题中：
  - 1) 是否需用到数据结构？
  - 2) 会用到哪种类型的逻辑结构？
  - 3) 存储结构是怎样的？
    - 存储哪些数据（包括数据类型和取值）？
    - 顺序存储/链式存储的选择？



2019/9/28

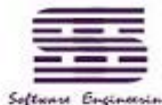


# 课堂练习2

- **问题2**：读入一个包含学生姓名和年龄的数据文件，要求将记录按照年龄和学生姓名升序地插入到一个链表中，丢弃重复的记录，然后打印该有序链表。若键盘输入一个学生的信息为“**A24**张三”，则将该学生的信息插入到有序链表中，并丢弃重复的记录，然后打印该有序链表。若键盘输入一个学生的信息为“**D24**张三”，则将该学生的信息从该有序链表中删除，然后打印该有序链表。其中，文件中每行为一条数据记录，它的前**2**个字符表示年龄，其后的最多**20**个字符表示学生姓名。如“**22**斯琴高娃”。
- 请指出上述问题中是否需用到数据结构？会用到哪种类型的数据结构（包括逻辑结构和存储结构）？



2019/9/28



12

# 课堂练习1 扩展

- 问题1：读入一个包含城市和气温信息的数据文件，要求将记录按照气温和城市名称**升序地打印**出来，并指示位于中间的条目。其中，文件中每行为一条数据记录，它的前3个字符表示气温，其后的最多**124**个字符表示城市名称。如“-10Duluth”。
- 请指出上述问题中：
  - 1) 是否需用到数据结构？
  - 2) 可以使用哪几种逻辑结构？（不限于线性结构）
  - 3) 对应的存储结构分别是怎样的？
    - 存储哪些数据（包括数据类型和取值）？
    - 顺序存储/链式存储的选择？



# 总结:

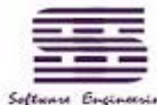
- 什么时候需要用到数据结构？
  - 操作对象为：取值为同种类型的很多数据，且这些数据间存在某种关系 或者 某些共性操作
- 若需要用到**DS**，那什么时候可以使用线性结构？
  - 被操作的数据之间没有天然的一对多 和 多对多的关系
  - 对已存储的数据进行处理时，处理顺序 有明显的唯一的先后次序关系
- 若采用线性结构，具体该使用哪种存储结构（此处，只讨论：顺序表和链表）
  - 取决于数据处理时的**最频繁**操作，为静态操作，还是动态操作？

## 2.3 顺序表的定义及实现

- 顺序表：在内存中连续存储的线性表
- 特性：
  - 逻辑上相邻的元素，物理存储地址必相邻；
  - 可随机存取：通过顺序表的名称和下标可以直接访问顺序表中的任一个元素。
- **Q：**对于顺序表**A**中下标为**i**的元素，它的直接前驱为？直接后继为？
- **结论：**顺序表中用下标来表明线性特征。



2019/9/28



15



# 顺序表——静态定义

- 顺序表的静态定义：利用数组。
- 方案一：  
    **int Sqlist[100];**
- 方案二：  
    **#define List\_Size 100 /\*分配空间的大小\*/**  
    **int Sqlist[List\_Size];**
- 方案三：（通用性最强，类似**STL:vector**）  
    **#define List\_Size 100/\*分配空间的大小\*/**  
    **Typedef Struct{**  
        **int elem[List\_Size]; /\*存储空间\*/**  
        **int len; /\*实际长度\*/**  
    **}SqlList\_static;**

1. 只需定义 单个结点和结点数量，就可以实现“顺序表”数据存储结构的定义；
2. 结点之间的关系隐式地用下标[]来描述。





- 评价：

- 该结构比较机械：分配的内存空间大小固定。
  - **List\_Size** 过小，会导致顺序表上溢；
  - **List\_Size** 过大，会导致空间利用率不高
- 在编译的时候，系统在函数栈中分配连续的内存空间。当静态顺序表所在的函数执行完毕后，由系统来回收所开辟的内存空间。
- 程序运行时，出现上溢问题，将没法修补。



# 顺序表——动态定义

1. 数组≠顺序表.
2. 并不是只有链表中才能有指针.

- 顺序表的动态定义：利用指针。

```
#define List_Size 100 /*分配空间的大小*/
```

```
typedef struct{
```

```
    int *elem;    /*顺序表的存储空间*/
```

```
    int len;      /*实际长度*/
```

```
    int ListSize ; /*当前分配的空间大小*/
```

```
} Sqlist;
```

- 特点：

- 需手动分配存储空间: **malloc()**
- 可以在程序运行过程中，重新分配空间: **realloc()**
- 不再使用顺序表时，需手动释放所占的空间: **free()**
- 可以避免“机械”，但是会增加时间开销。



- **C中的动态分配与释放函数**

- **Void \*malloc(unsigned int size)**

- /\*生成一个大小为**size**的结点空间，将该空间的起始地址赋给**p**\*/

- **Free(void \*p)**

- /\*回收**p**所指向的结点空间\*/

- **Void \*realloc(void \*p, unsigned int size)**

- /\*重新分配大小为**size**的结点空间，并将该空间的起始地址赋给**p** \*/



```
Int InitSqList(SqList *L)//构造一个空的顺序表L
{
```

```
    L->elem=(int *) malloc(List_Size *sizeof(int));
```

```
    if (L->elem==NULL)
```

```
        exit(EXIT_FAILURE);
```

```
    L->len=0;
```

```
    L->ListSize =List_Size;
```

```
    return 1;
```

```
}
```



2019/9/28



Software Engineering

20

# 顺序表——两种定义的对比总结

- 两种顺序表定义的对比：
  - 不同：分配存储空间的方式不同
    - 静态定义的顺序表由系统自动分配和回收存储空间；
    - 动态定义的顺序表需要手动分配和回收存储空间，但是也可以再分配。
    - 因而它们创建和销毁顺序表两种操作的实现不同，
  - 相同点：本质上都是存储在连续空间上，因而对数据的操作方式（查找，插、删）都是一样的。



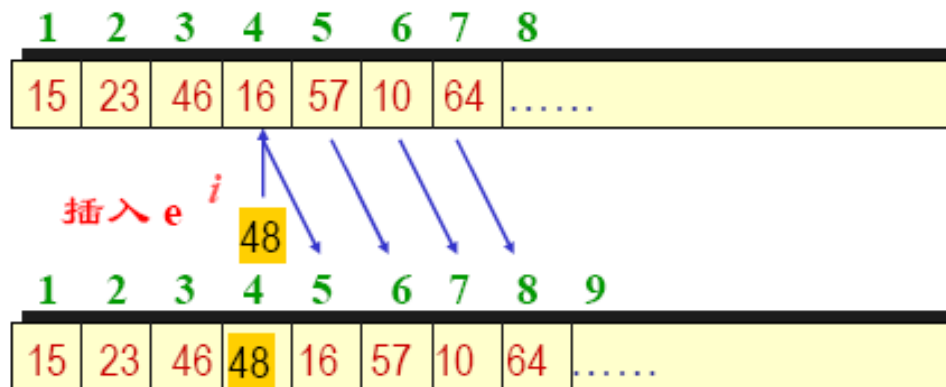
# 课堂练习3

- **Q: 分别利用静态定义的顺序表和动态定义的顺序表，来实现课堂练习1和练习2中的数据结构**

- **问题1:** 读入一个包含城市和气温信息的数据文件，要求将记录按照气温和城市名称升序地插入到一个链表中，丢弃重复的记录，然后打印该有序链表，并指示位于中间的条目。接着，逐渐缩短链表并重新打印显示中间条目。其中，文件中每行为一条数据记录，它的前3个字符表示气温，其后的最多124个字符表示城市名称。如“-10Duluth”。
- **问题2:** 读入一个包含学生姓名和年龄的数据文件，要求将记录按照年龄和学生姓名升序地插入到一个链表中，丢弃重复的记录，然后打印该有序链表。若键盘输入一个学生的信息为“A24张三”，则将该学生的信息插入到有序链表中，并丢弃重复的记录，然后打印该有序链表。若键盘输入一个学生的信息为“D24张三”，则将该学生的信息从该有序链表中删除，然后打印该有序链表。其中，文件中每行为一条数据记录，它的前2个字符表示年龄，其后的最多20个字符表示学生姓名。如“22斯琴高娃”。

# 顺序表——插入操作的实现

- **Status ListInsert\_Sq( SqList &L, int i, ElemType e)**



- 参数：顺序表**&L**、插入位置**i**、插入元素**e**
  - 插入分析：
    - 第**i**个位置放**e**，则原来第**i~L.len**个数据元素必须先后移，以腾出第**i**个位置；
    - 后移的顺序为：从最后一个元素开始，逐个往后移
- ```

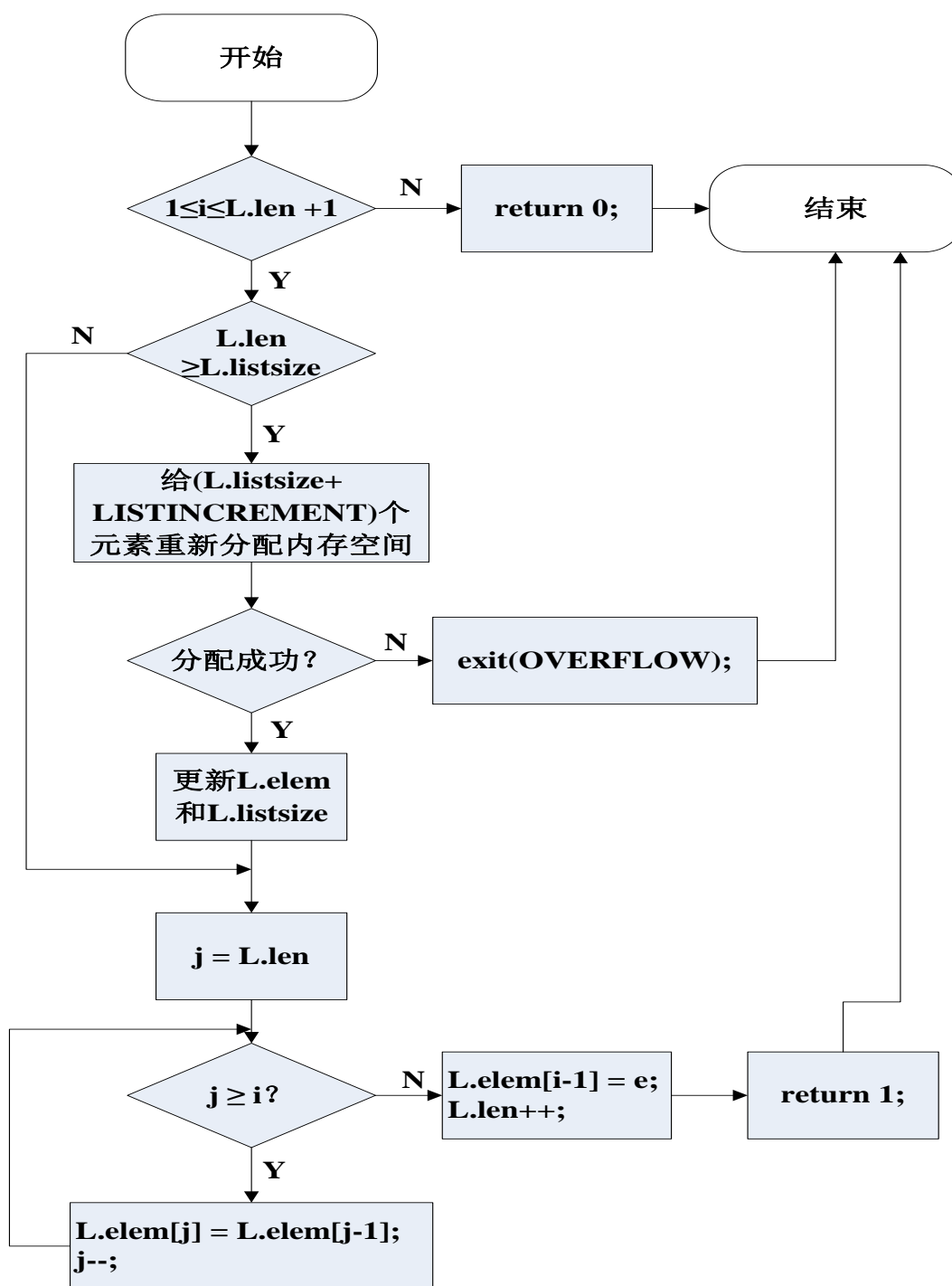
for ( j = L.len; j >= i; j--)
    L.elem[j] = L.elem[j-1];           // 后移
L.elem[i - 1] = e; //第i个元素存放在L.elem[i-1]中，数组下标的起始为0
L.len = L.len+1;

```
- 合法的位置：  $i: 1..L.len+1$
  - 健壮性
  - 上溢及处理：
    - 上溢发生的条件：  $L.len \geq L.ListSize$
    - 处理：要先申请一个有一定增量的空间：申请成功则原空间的元素复制到新空间，修改**L.listsize**，再进行插入工作；否则报错退出。





• 算法流程图为:



• 算法的伪代码:

$C_i$ 表示第*i*条语句  
的实际执行时间

cost

times

ListInsert\_Sq(L, i, e)

▶位置合法性的判断:  $1 \leq i \leq n+1$

1 if  $i < 1$  or  $i > \text{length}[L] + 1$

$C_1$

1

2 then return 0

$C_2$

0/1

▶上溢时增加空间的分配

3 if  $\text{length}[L] \geq \text{listsize}[L]$

$C_3$

1

4 then newbase ← reallocate more memory spaces

$C_4$

0/1

5 if newbase = NIL

$C_5$

0/1

6 then error" OVERFLOW"

$C_6$

0/1

7 elem[L] ← newbase

$C_7$

0/1

8 listsize[L] ← size of new spaces

$C_8$

0/1

▶插入元素

9 for j ← length[L] to i

$C_9$

$n-i+2$

10 do L[j] ← L[j-1]

$C_{10}$

$n-i+1$

11 L[i-1] ← e

$C_{11}$

1

12 length[L] ← length[L]+1

$C_{12}$

1

13 return 1

$C_{13}$

1

# • 算法的伪代码:

ListInsert\_Sq(L, i, e)

$C_i$ 表示第*i*条语句  
的实际执行时间

cost

times

▶位置合法性的判断:  $1 \leq i \leq n+1$

1. 总的实际执行时间  $T = ?$  (分三种情况考虑)
2. 估算最频繁语句的执行次数, 用O来描述该算法的时间复杂度。(分三种情况考虑)
3. 前面的2种描述算法复杂度的结果是否有关联?
4. 尝试总结下前面2种描述算法复杂度的方式分别适用于什么场合。

$C_1$

1

$C_2$

0/1

$C_3$

1

memory spaces  $C_4$  0/1

$C_5$

0/1

$C_6$

0/1

$C_7$

0/1

ces  $C_8$

0/1

▶插入元素

9 for  $j \leftarrow \text{length}[L]$  to  $i$

$C_9$

$n-i+2$

10 do  $L[j] \leftarrow L[j-1]$

$C_{10}$

$n-i+1$

11  $L[i-1] \leftarrow e$

$C_{11}$

1

12  $\text{length}[L] \leftarrow \text{length}[L]+1$

$C_{12}$

1

13 return 1

$C_{13}$

1

## • 时间复杂度分析:

- 频次最高的操作: 移动元素。
- 上溢时的空间的再分配与复制(**realloc**操作)的时间复杂度与**realloc**的算法以及当前的表长相关, 至少为 $O(n)$ ; 但它仅在插入元素会引起上溢时才执行。
- 若线性表的长度为 $n$ , 则:
- 最好情况: 插入位置 $i$ 为 $n+1$ , 此时无须移动元素, 时间复杂度为 $O(1)$ ;
- 最坏情况: 插入位置 $i$ 为 $1$ , 此时须移动 $n$ 个元素, 时间复杂度为 $O(n)$ ;
- 平均情况: 假设 $p_i$ 为在第 $i$ 个元素之前插入一个元素的概率, 则:  
插入时移动次数的期望值:  $E_{is} = \sum_{i=1}^{n+1} p_i (n - i + 1)$   
等概率时, 即,  $E_{is} = \frac{1}{n+1} \sum_{i=1}^{n+1} (n - i + 1) = \frac{n}{2}$   
 **$\therefore T(n) = O(n)$**

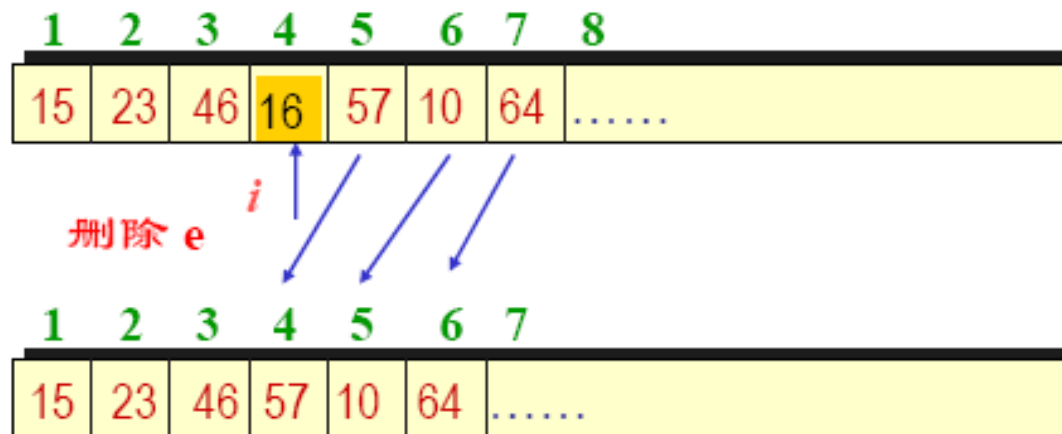
```

Status ListInsert_Sq( SqList &L, int i, ElemType e)
{
// 位置合法性的判断
if ( i<1 || i>L.len +1 )      return 0;
// 上溢时增加空间的分配
if( L.len >= L.listsize)
{
    newbase = (ElemType *) realloc(L.elem,
                                   (L.listsize+ LISTINCREMENT)*sizeof(ElemType));
    if ( newbase == NULL ) exit(OVERFLOW);
    L.elem = newbase;
    L.listsize += LISTINCREMENT;
}
// 插入元素
for ( j = L.len; j >= i; j--)    L.elem[j] = L.elem[j-1];
L.elem[i-1] = e;
L.len++;
return 1;
}

```

# 顺序表——删除操作的实现

- **Status ListDelete\_Sq( SqList &L, int i, ElemType &e)**

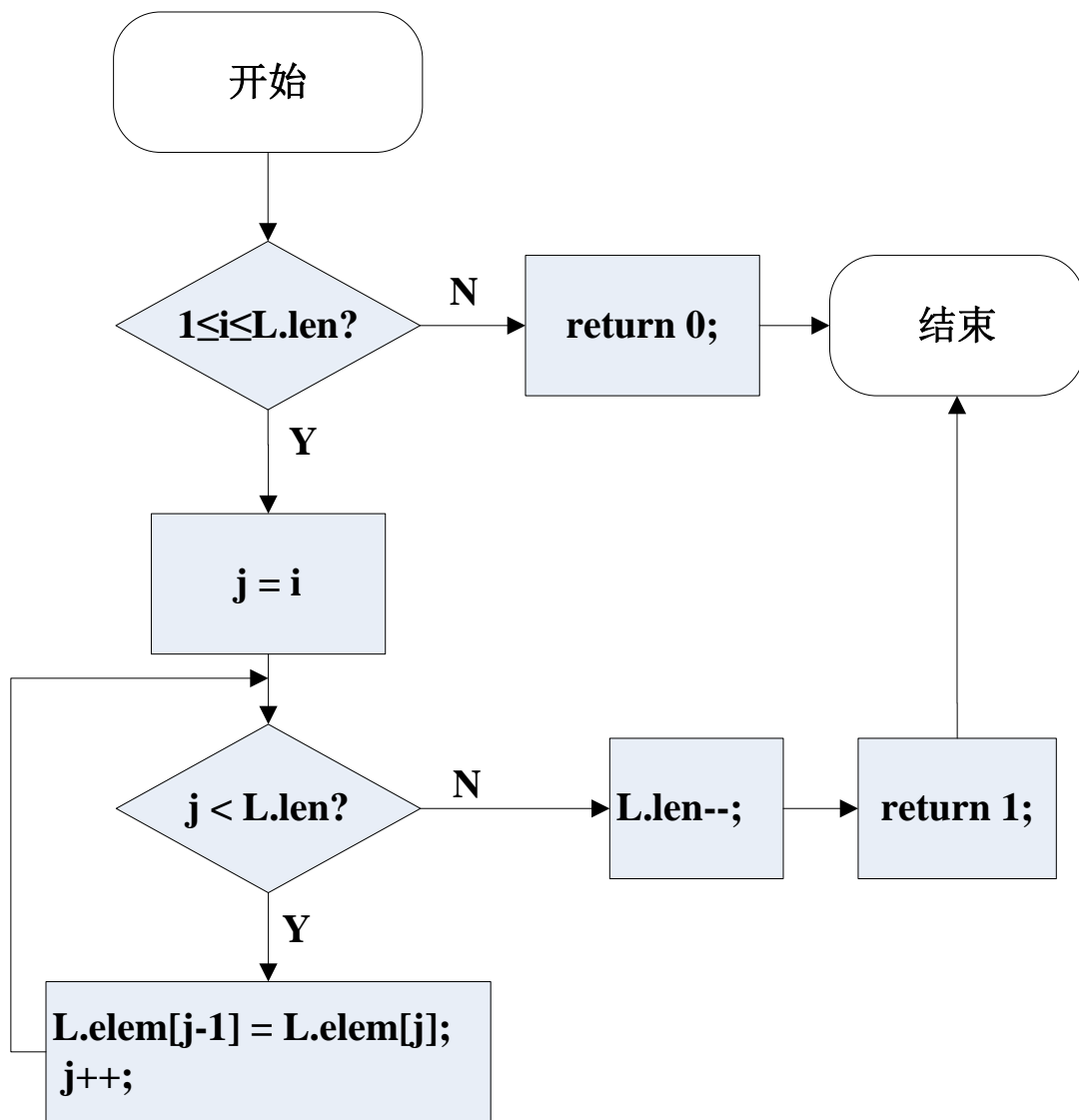


- 参数：顺序表&L、删除位置i
- 删除分析：
  - 去掉第i个元素，则原来第i+1~L.len个数据元素须前移，以覆盖第i个位置；
  - 前移的顺序为：

```
for ( j = i; j < L.len ; j++)  
    L.elem[j-1] = L.elem[j];  
L.len = L.len-1
```
- 合法的位置：i:1..L.len
- 下溢：L.len≤0 —— 隐含在i的条件中



- 算法流程图为:





- 算法伪代码为:

```
Status ListDelete_Sq( SqList &L, int i) {  
    // 位置合法性的判断  
    if ( i<1 || i>L.len )    return 0;  
    // 删除  
    for ( j = i; j < L.len ; j++) L.elem[j-1] = L.elem[j];  
    L.len--;  
    return 1;  
}
```



- 时间复杂度分析:

- 频次最高的操作: 移动元素

- 若线性表的长度为n, 则:

- 最好情况: 删除位置i为n, 此时无须移动元素, 时间复杂度为O(1);

- 最坏情况: 删除位置i为1, 此时须前移n-1个元素, 时间复杂度为O(n);

- 平均情况: 假设 $q_i$ 为删除第i个元素的概率, 则删除时移动次数的期望值:  $E_{de} = \sum_{i=1}^n q_i(n-i)$   
等概率时, 即,  $q_i = \frac{1}{n}$

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

- $\therefore T(n) = O(n)$

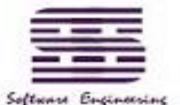


# 顺序表——查找操作

- 直接利用下标[ ]
- **Q1:** 对于第*i*个元素，如何查找其直接前驱和直接后继？
- **Q2:** 查找指定的第*i*个元素。



2019/9/28



35

- 顺序表结构

- 优点：随机存取。

- 弱点：

- 1) 空间利用率不高（预先按最大空间分配）
- 2) 表的容量不可扩充（针对顺序表的静态定义方案）
- 3) 即使表的容量可扩充（针对顺序表的动态定义方案），由于其空间再分配和复制的开销，因而也不允许它频繁地使用
- 4) 插入或删除时需移动大量元素。

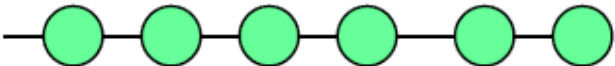


# 顺序表——适用环境

- 例1：顺序文件的查找。（有序顺序表+二分查找）
- 例2：顺序文件的查找，并在文件中添加一个元素。（基于二叉平衡树的查找）
- 结论：顺序表适用于输入数据的大小已知，且无太多动态操作的应用问题。



## 2.4 链表的定义及实现

- 链表是链式存储的线性结构。
  - 由一连串结点组成，结点之间通过链串起来。
- 结点：
  - 是链表元素的存储映像。
  - 每个结点包括数据域和指针域。
- 指针/链：它用于将结点们联系起来。也可以用于唯一定位一个结点。
- 头指针：链表存取的开始。
- 单链表：只有一个指针域的链表。
- 双链表：有两个指针域的链表。
- **Q：**对于单链表中指针**P**所指向的结点，它的直接前驱为？直接后继为？ 对于双链表呢，情况又如何？

**结论：链表中用链来表明线性特征。**

## 2.4 单链表——定义

```
• struct Node{  
    int data;  
    struct Node *next};  
};  
Typedef struct Node *Link;  
Link head;
```

1. 只需要定义结点，就可以实现“单链表”数据存储结构的定义；
2. 结点之间的关系隐式地用指针next来描述。



2019/9/28

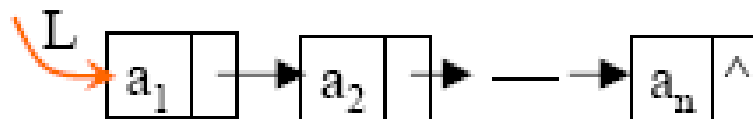


Software Engineering

39

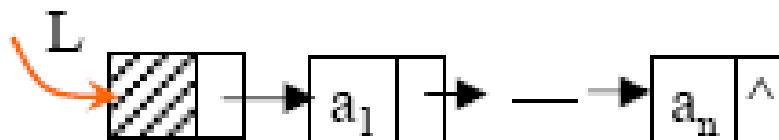
## • 无头结点的单链表

- 头指针为L，则空表时， $L == \text{NULL}$
- 由于第一个结点无前驱结点，所以只能通过某指针变量来指向，如L；其余结点均有前驱结点，故可通过其直接前驱结点的next域来指向，即.....->next;
- 表示方法的不同，会造成对结点的操作处理的不同。



## • 有头结点的单链表

- 空表时，L指向一结点(称为头结点)，该结点的数据域可以不存储信息，也可存储如表长等的附加信息，结点的指针域存放NULL，即 $L \rightarrow \text{next} == \text{NULL}$ 。
- 第一个结点和其余结点均可统一表示为其直接前驱结点的next域所指向的结点，即.....->next。



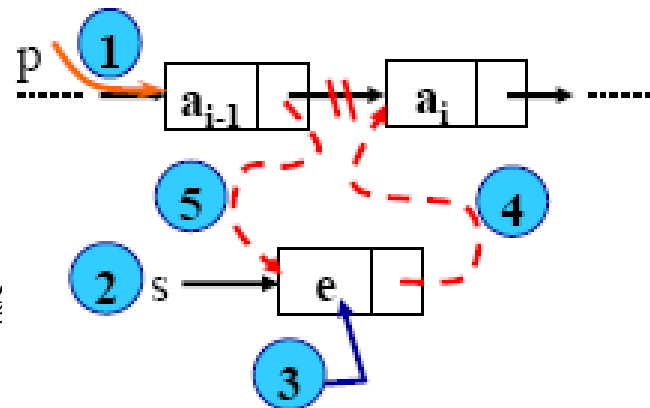


## 2.4 单链表——插入操作

- `ListInsert_L(LinkList &L, int i, ElemType e)`

### 【设计思路】

- 相关结点:  $a_{i-1}$  和  $a_i$
- 结点的表示: 引入指针变量 `LinkList p`;  
     $\because$  链表结点的指针域是指向该结点的直接后继;  
     $\therefore$  当  $p$  指向  $a_{i-1}$  时,  $a_i$  可用  $*(p \rightarrow \text{next})$  表示
- 关键步骤:
  - ① 找到  $a_{i-1}$  的位置, 即使  $p$  指向  $a_{i-1}$  结点, 可参考 `GetElem_L()` 的处理
  - 若  $p \neq \text{NULL}$ , 则② `s = (LinkList)malloc(sizeof(LNode))`
  - ③ `s->data = e`
  - ④ `s->next = p->next`
  - ⑤ `p->next = s`
- 注意: ④和⑤不能交换, 否则会导致  $a_i$  的位置无法获取。
- 频度最高的操作: 确定  $a_{i-1}$  的位置
- 若线性表长度为  $n$ , 则:
  - 最好情况:  $T(n) = O(1)$
  - 最坏情况:  $T(n) = O(n)$
  - 平均情况:  $T(n) = O(n)$



- 有头结点的单链表中的插入算法

```
Status ListInsert(LinkList &L, int i, ElemType e){
```

```
// 有头结点，无须对i为1的插入位置作特殊处理
```

```
p = L; j = 0; // 对p,j初始化; *p为L的第j个结点
```

```
while( p != NULL && j<i-1){
```

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

```
j++; // 寻找第i-1个结点的位置
```

```
}
```

```
if( p == NULL || j>i-1) return ERROR; // i小于1或大于表长
```

```
s = (LinkList )malloc(sizeof(LNode)); // 生成新结点
```

```
if ( s == NULL )
```

```
exit(OVERFLOW); // 空间分配不成功，报错返回
```

```
s->data = e; s->next = p->next; // 插入L中
```

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

```
return OK;
```

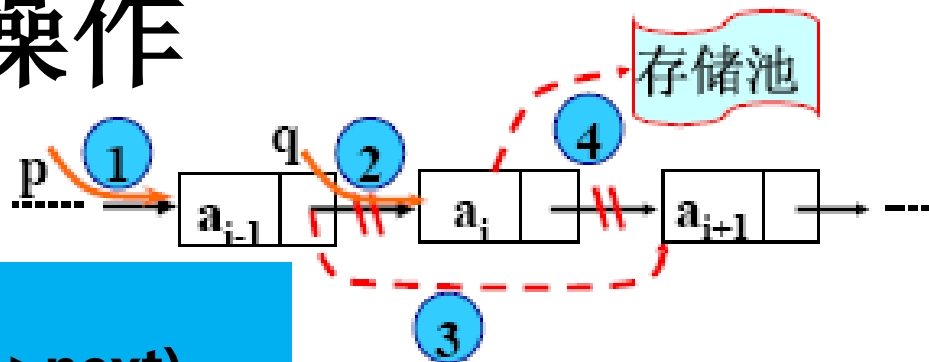
```
}
```

- 无头结点的单链表中的插入算法

```
Status ListInsert(LinkList &L, int i, ElemType e) {  
// 无头结点, 须对i为1的插入位置作特殊处理  
if ( i==1){  
s = (LinkList )malloc(sizeof(LNode));           // 生成新结点  
if ( s == NULL )  exit(OVERFLOW);                // 空间分配不成功, 报错返回  
s->data = e; s->next = L;                          // 插入到链表L中  
L = s;   // 修改链头指针L  
}  
else{  
p = L; j = 1;                                     // 对p,j初始化; *p为链表的第j个结点  
while( p != NULL && j<i-1){  
p = p->next;                                       // 寻找第i-1个结点的位置  
j++;  
}  
if( p == NULL || j>i-1)      return ERROR; // i小于1或大于表长  
s = (LinkList )malloc(sizeof(LNode));           // 生成新结点*s  
if ( s == NULL )  exit(OVERFLOW);                // 空间分配不成功, 报错返回  
s->data = e; s->next = p->next;                  // 插入到链表L中  
p->next = s;  
}  
return OK;  
}
```

【思考】对比无头结点和有头结点在插入算法上的不同, 分析其中的原因。

# 单链表——删除操作



相关结点:  $a_{i-1}$ 、 $a_i$ 和 $a_{i+1}$

结点表示:  $*p$ ,  $*(p \rightarrow next)$ ,  $*(p \rightarrow next \rightarrow next)$

- 设计思路:

- 关键步骤:

- ①找到 $a_{i-1}$ 的位置, 即使 $p$ 指向 $a_{i-1}$ 结点, 可参考 **GetElem\_L()**的处理

$p \rightarrow next \neq NULL$  (有待删除的结点), 则

- ②  $q = p \rightarrow next$  (记录待释放结点的位置)

- ③  $p \rightarrow next = p \rightarrow next \rightarrow next$

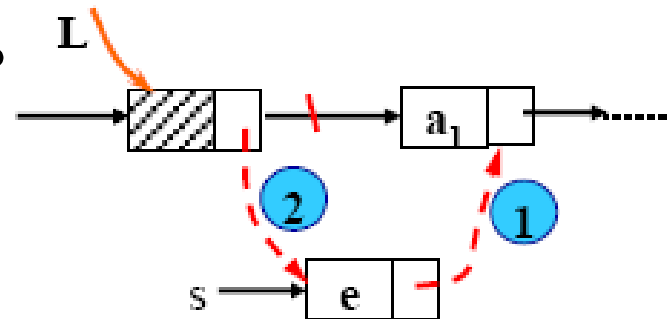
- ④ **free(q)**

**注意:** 必须在③④前增加②步, 否则在执行了③后要释放的结点无法标识。

# 单链表——创建操作

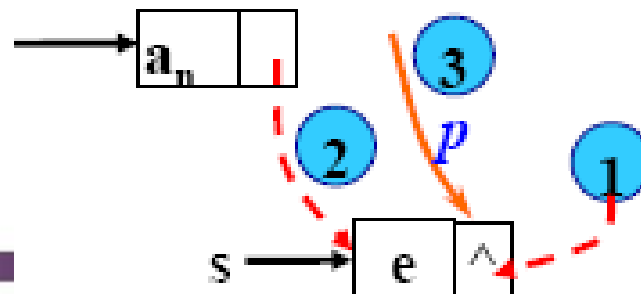
- 头插法

- 思想：每次将待插结点\*s插入到第一个结点之前；当有头结点时，待插结点也可视为插入到第0个结点(头结点)之后。



- 尾插法：

- 思想：待插结点\*s插入到最后一个结点之后



# 单链表——头插法创建操作

- 思想：每次将待插结点\*s插入到第一个结点之前；当有头结点时，待插结点也可视为插入到第0个结点(头结点)之后。
- 插入步骤：以单链表中**有头结点**为例，单个结点的构造和插入步骤如下
  - ①  $s = (\text{LinkedList})\text{malloc}(\text{sizeof}(\text{LNode}))$
  - ②  $\text{scanf}(\&s \rightarrow \text{data})$
  - ③  $s \rightarrow \text{next} = L \rightarrow \text{next}$
  - ④  $L \rightarrow \text{next} = s$
- 算法时间复杂度分析：每次插入一个结点所需的时间为 $O(1)$   
 $\therefore$ 头插法创建单链表的时间复杂度  $T(n) = O(n)$

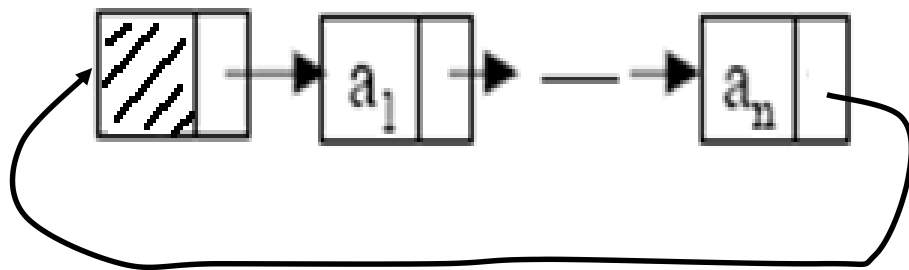
# 单链表——尾插法创建操作

- 思想：待插结点\*s插入到最后一个结点之后
- 插入步骤：
  - ① 获得最后一个结点的位置,使p指向该结点
  - ② `p->next = (LinkedList)malloc( sizeof(LNode))`
  - ③ `p = p->next`
  - ④ `scanf( &p->data )`
  - ⑤ `p->next = NULL`
- 算法时间复杂度分析：要想获取最后一个结点的位置，必须从链头指针开始顺着next链搜索链表的全部结点，该过程的时间复杂度是  $O(n)$ 。如果每次插入都按此方法获取最后一个结点的位置，则整个创建算法的时间复杂度为  $T(n) = O(n^2)$ 。



# 循环链表

- 循环链表
- 循环链表 VS. 单链表:



1) 定义相同;

2) 最后一个结点不同:

- 循环链表最后一个结点的指针不为空，而是指向表头结点（头结点/第一个结点）；

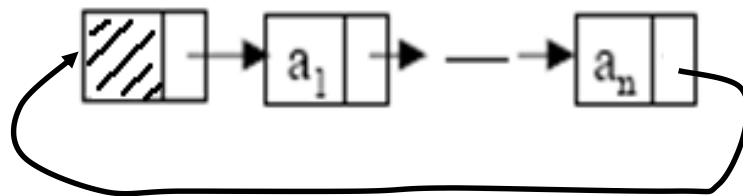
3) 判断表尾结点的条件不同:

- 循环链表中，判断当前是否到达表尾结点的条件，不再是看其是否为**NULL**，而是看其是否等于头指针

- 问题1：如何从一个结点出发，访问到链表中的全部结点？



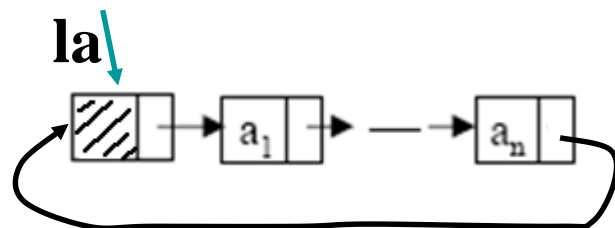
# 循环链表



- **问题2:** 对于循环链表，如何在 $O(1)$ 时间内由链表指针访问到第一个结点和最后一个结点？

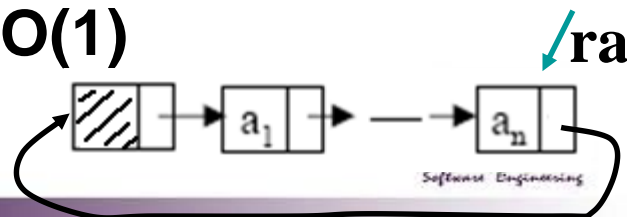
## – 头指针表示法:

- 第一个结点:  $*(la \rightarrow next)$   $T(n)=O(1)$
- 最后一个结点: 需从表头搜索到表尾  $T(n)=O(n)$



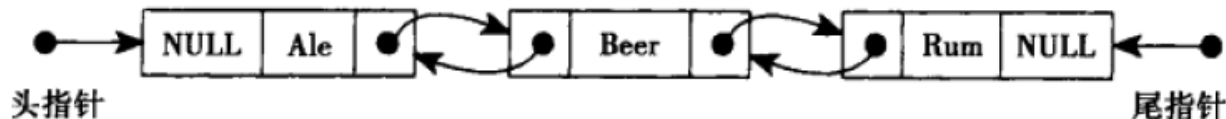
## – 尾指针表示法:

- 第一个结点:  $*(ra \rightarrow next \rightarrow next)$   $T(n)=O(1)$
- 最后一个结点:  $*ra$   $T(n)=O(1)$



# 双向链表

- 单链表的缺点：只知其直接后继结点，不知当前结点的直接前驱。
- 如何 $O(1)$ 时间内找到当前结点的直接前驱？
- 巧妙的解决方案：
  - `Link prev; curr=prev->next;`
- 本质上克服单链表的弱点——双向链表



- 双向链表:
  - 每个结点有**2**个链：分别指向逻辑相邻的**2**个结点。
  - 可在 $O(1)$ 时间内找到一个结点的直接前驱结点和直接后继结点
  - 广泛应用，如**STL: list**



```

Struct Node{
ElemType  data;
struct Node    *prior;
struct Node    *next;
};
typedef struct Node * Link;
typedef struct {
Link          head, tail;
int           len;
}DLinkedList;

```

- 特点：若d指向表中的一个内部结点，则d->next->prior == d->prior->next == d
- 双向循环链表：存在两个环。

# 线性表的应用分析

- 例1：将两个线性表合并为一个线性表；
- 例2：将按值非递减有序的两个线性表合并为一个有序线性表；
- 思路：
  - **Step1**：采用何种数据存储结构？（顺序表？链表？）
  - **Step2**：算法设计思路
  - **Step3**：算法时间复杂度分析

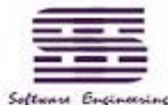


## 2.5 案例分析——案例1

- 程序清单2-1
- 问题：读入一个包含城市和气温信息的数据文件，要求将记录按照气温和城市名称升序地插入到一个链表中，丢弃重复的记录，然后打印该有序链表，并指示位于中间的条目。接着，逐渐缩短链表并重新打印显示中间条目。其中，文件中每行为一条数据记录，它的前3个字符表示气温，其后的最多124个字符表示城市名称。如“-10Duluth”。
- 解题思路：
  - Step1：判断输入参数的数量是否正确。
  - Step2：依据文件名称，打开指定的文件并将内容进行读取，若不能正常打开，则报错；
  - Step3：若能正常读取文件，则 **创建一个空的链表**；
  - Step4：逐行读取文件，对于每一行，
    - 1) 将该行数据保存到缓冲区中；
    - 2) 解析该数据，并分别存储到节点中的对应数据域中
    - 3) 将该节点 **升序地添加**到链表中；
  - Step5：**打印链表**中所有节点，并指示位于中间的条目
  - Step6：依次 **删除链表中的节点**并释放其所占用的内存空间，然后打印当前链表。
  - Step7：关闭文件。

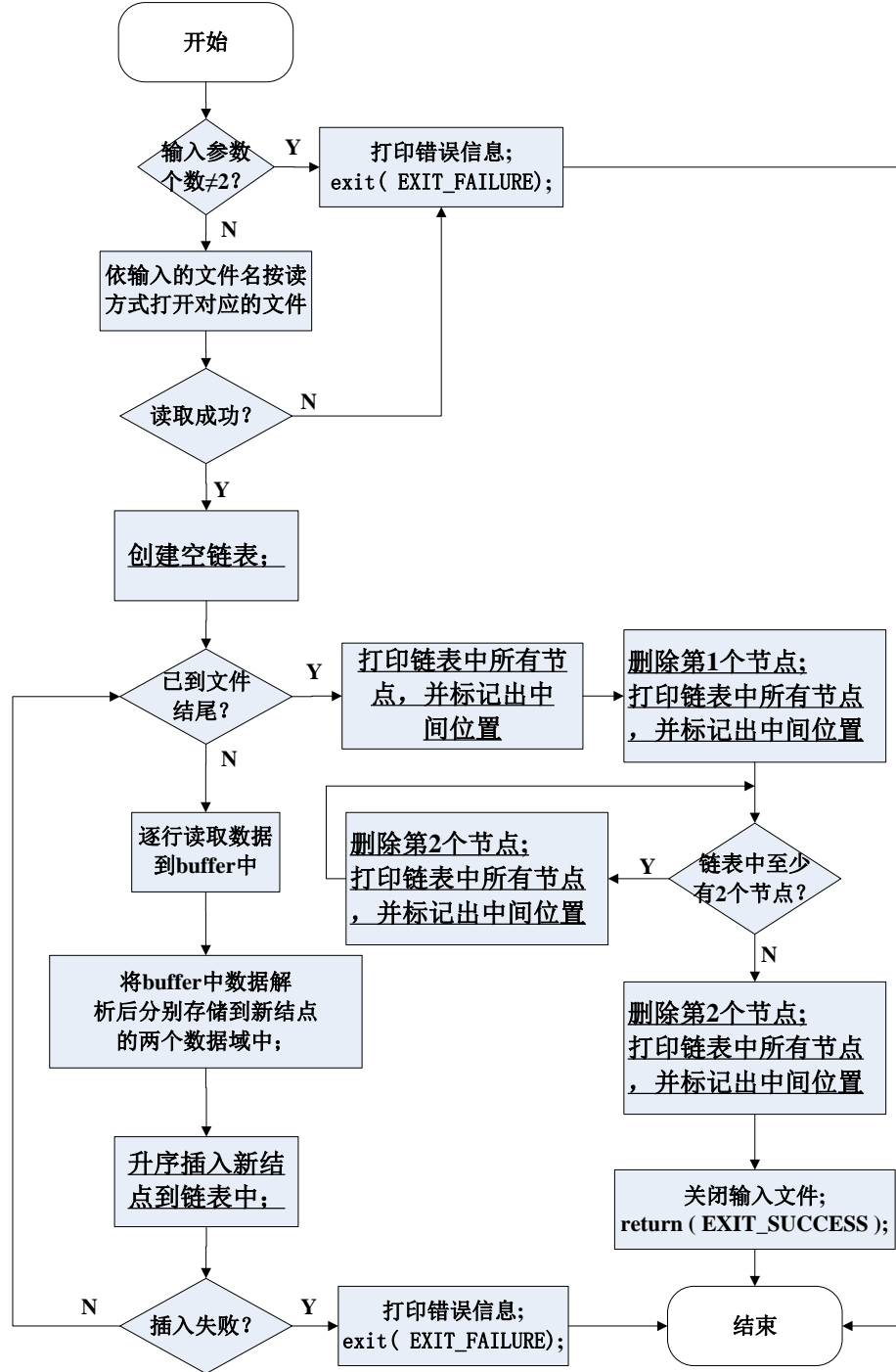


2019/9/28



53

## • 算法流程图:



- 创建空的链表**CreateList(Void);**
  - 头指针指向**NULL**;
  - 设置节点数=0;
- 节点升序地添加到链表中**AddNodeAscend(Link);**
- 打印节点**ShowNodes(void);**
  - 统计实际节点数;
  - 计算出中间节点的序号;
  - 若链表非空, 则依次遍历链表中的所有节点。对于每个节点, 打印, 且若该节点为中间位置, 还需打印出中间位置的标记。
  - 若链表为空, 则打印出链表为空的提示信息。
- 删除节点**DeleteNode(Link);**
  - 找出要删除的节点的位置;
  - 若找到了匹配项, 则删除它, 并释放对应的内存空间, 并将节点数减1



- 评价:

- 程序清单2-1中，直接操纵链表的函数大都是**通用**的：与节点中数据域包含的具体数据无关。但是有**3**处有特定性要求：

- 1) 指向直接后继节点的链称为**Next**;
    - 2) **NodeCmp()**和**DuplicateNode()**都要求设计成适合于将要处理的具体数据，因此在具体应用中需定制这**2**个函数。

- 时间复杂度分析：

- $T(n)=O(n^2)$ ; (最坏情况下)
    - $T(n)=O(n)$ ; (最好情况下)
    - $T(n)=O(n^2)$ ; (平均情况下)





# 课堂练习

- 问题：读入一个包含学生姓名和年龄的数据文件，要求将记录按照年龄和学生姓名升序地插入到一个链表中，丢弃重复的记录，然后打印该有序链表。若键盘输入一个学生的信息为“**A24**张三”，则将该学生的信息插入到有序链表中，并丢弃重复的记录，然后打印该有序链表。若键盘输入一个学生的信息为“**D24**张三”，则将该学生的信息从该有序链表中删除，然后打印该有序链表。其中，文件中每行为一条数据记录，它的前**2**个字符表示年龄，其后的最多**20**个字符表示学生姓名。如“**22**斯琴高娃”。
- 提示：可以利用程序清单2-1中定义的直接操纵链表的函数来实现其他应用需求。（将函数设计为具有通用性的好处）



# 案例分析——案例2

- 程序清单2-5
- 问题：读入一个包含若干个单词的数据文件，要求：
  - 1) 统计出每个单词出现次数并按单词的升序顺序打印；
  - 2) 顺序打印出源文件中的单词；
  - 3) 逆序打印出源文件中的单词；
  - 4) 找到源文件中的重复单词，并每隔一个删除一个重复单词。



## 2.6 顺序表的延伸：位向量/位图

- 采用位向量：即**bit**数组
- 什么是位向量？
  - 示例：{0,1,2,20,32}用位向量存储为： **(33bit)**

**1000000000001000000000000000000000000000000**

**32                  20                                      2 1 0**

假设该位向量用包含2个整数的顺序表X表示。则：

$$\mathbf{X}[0]=2^0+2^1+2^2+2^{20}$$

$$\mathbf{X}[1]=2^0$$

**Q: 若要利用位向量来存储整数64, 该如何存储?**

1) 64存放在顺序表X的第几个整数中?

**A:  $64 / 32 = 2$ , 因而64存放在X[2]中。**

## 2) 64该如何存放在它对应的整数元素中?

**A:**  $64 \% 32 = 0$ , 因而将x[2]的值加 $2^0$ 即可。



```

class IntSetBitVec {
private:
    enum { BITSPERWORD = 32, SHIFT = 5, MASK = 0x1F };
    int    n, hi, *x;
    void set(int i) {      x[i>>SHIFT] |= (1<<(i & MASK)); }
    void clr(int i) {      x[i>>SHIFT] &= ~(1<<(i & MASK)); }
    int test(int i) { return x[i>>SHIFT] & (1<<(i & MASK)); }
public:
    IntSetBitVec(int maxelements, int maxval)
    {
        hi = maxval;
        x = new int[1 + hi/BITSPERWORD];
        for (int i = 0; i < hi; i++)
            clr(i);

        n = 0;
    }
    int size() { return n; }
    void insert(int t)
    {
        if (test(t))
            return;

        set(t);
        n++;
    }
    void report(int *v)
    {
        int j=0;
        for (int i = 0; i < hi; i++)
            if (test(i))
                v[j++] = i;
    }
};

```

作用分别是？

**set(i):**将整数x的第i比特置为1;

**clr(i):**将整数x的第i比特置为0;

位向量

顺序表的长度为何这样计算？

1个整数用BITSPERWORD个bit描述。假设位向量中的最大整数为maxval，意味着描述该位向量需要maxval比特来，即需要 $(1 + \text{maxval} / \text{BITSPERWORD})$ 个整数;

**Q1:** New语句执行了几次？

**Q2:** 位向量的三个基本操作（函数set(),clr(),test()）的功能分别是？

# 作业2（课前）

运行Test文件下的程序`test.sln`，并分析总结C中的各类数据在内存中的分布情况，回答以下问题：

- 1、代码区，数据区（静态变量、全局变量和局部变量）分别对应的地址的相对大小；（哪些位于高地址，哪些位于低地址）
- 2、通过`malloc`函数分配的内存空间 是否与 局部变量的的内存空间在同一个范围？
- 3、函数调用过程中，栈增长的方向是怎样的？形参和实参是如何关联的？
- 4、“一般变量，指针类型变量和`&`”这三类变量之间的关联是怎样的？

**注意：**为了使得验证环境不受其他因素干扰，一定要修改链接选项（见附件中的图），使得不支持ASLR（Address space layout randomization）

