第三章 线性表

§3.1 线性表及其运算

§3.2 线性表的存储表示

§3.3 线性表的应用

一.定义(逻辑上)

一个**线性表**(linear list)是$n \geq 0$个数据元素$a_1, a_2, \ldots, a_n$的有限序列，序列中除第一个和最后一个以外,每个元素有且仅有一个直接前驱和直接后继。

一.定义(逻辑上)

一个**线性表**(linear list)是$n \geq 0$个数据元素$a_1,a_2,\ldots,a_n$的有限序列，序列中除第一个和最后一个以外,每个元素有且仅有一个直接前驱和直接后继。

线性表一般可以简称为“**表**”，可以形式化表示为:

$A=(a_1,a_2,\ldots,a_n)$， 空表： A=()，或 A=∅

一.定义(逻辑上)

一个**线性表**(linear list)是$n \geq 0$个数据元素$a_1,a_2,\ldots,a_n$的有限序列，序列中除第一个和最后一个以外,每个元素有且仅有一个直接前驱和直接后继。

线性表一般可以简称为“**表**”，可以形式化表示为:

$A=(a_1,a_2,\ldots,a_n)$， 空表： A=()，或 A=∅

说明：数据元素a_i的含义在不同具体情况下可以有所不同，一般可以形式化表示为：**datatype**.

它可以是一个单一的具体数字、字符，也可以是一份档案或者一张发票。

二.线性表的特性

二.线性表的特性

◆ 是数据元素的有限序列。

线性表的长度定义为表中元素的个数。空表的长度定义为零。

◆ 元素之间呈线性关系。

即，元素之间的位置只取决于它们自己的逻辑顺序号，当0<i<n时，a_i的直接前驱是a_{i-1},直接后继是a_{i+1}。

综上所述，由于表中元素之间的相对位置呈现线性关系，也就是说线性表属于线性结构。

三.线性表的运算

线性表的运算形式和种类比较多概括起来只要有以下几类:

①确定线性表的长度 n 。应用举例:

②存取线性表的第 i 个数据元素,检验或改变某个数据项的值。
应用举例:

③在第 $i-1$ 个和第 i 个数据元素之间插入一个新的数据元素。
应用举例:

④删除第 i 个数据元素。应用举例:

⑤将两个或两个以上的线性表合并成一个线性表。应用举例:

⑥将一个线性表拆分成两个或两个以上的线性表。应用举例:

⑦重新复制一个线性表。 应用举例:

⑧对线性表中的数据元素依其某个数据项按某规律进行重组。
应用举例:

说明: 对于每种具体运算的形式化命名可以自行命名,或按照某种规则命名。

要注意的是,并非每个线性表都要全部进行以上运算,很多情况下只需要完成以上的几种或几种的变形。

第三章 线性表

§3.1 线性表及其运算

§3.2 线性表的存储表示

§3.3 线性表的应用

一.线性表的向量表示

在这里我们实际采用的是顺序方式中的一种特殊情况，即等长组织模式。这种等长模式恰好与向量特点相吻合，故以下我们采用向量的表述进行讨论。

1.基于向量的存储方法：

在所给地址空间中顺序地分配存储单元，且每个数据元素占据相同大小的存储空间单元。

2、数据元素的访问

要实现对数据元素的运算，就必须要知道元素的地址。设有线性表 $A=(a_1, a_2, \cdots, a_n)$，每个元素占用 l 个单元，现要访问的元素为 a_i。

在向量存储方法下就线性表A来说，在计算机内部实现实际上是一个一维数组，对于a_i的访问可通过下标变量直接完成，其大致原理如下：

$$Loc(a_i)=Loc(a_{i-1})+l$$
$$=Loc(a_1)+l*(i-1)$$

基地址

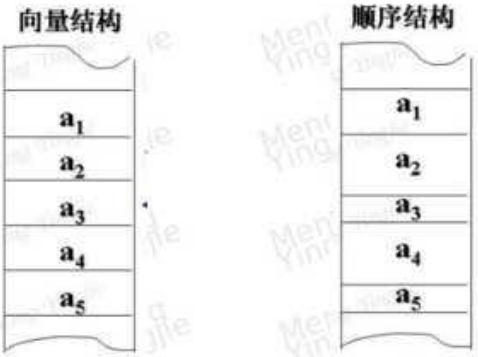
3.向量存储方法的特性：

- 存储分配上呈线性结构。
- 属于随机存储结构。

注：向量存储与顺序存储的关系。

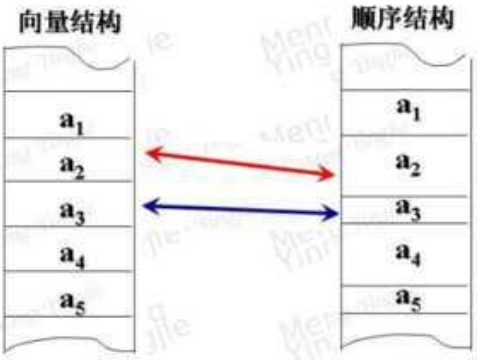
向量存储与顺序存储的异同：

同：



向量存储与顺序存储的异同：

异：a_2，a_3空间占用情况：

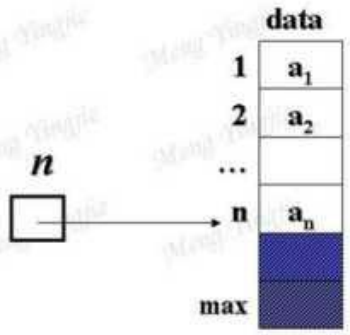


例如，磁带上的歌曲顺序存放，但并不意味着他们播放时间等长。

4.向量存储的形式化描述：

就其具体实现来说向量可用一个一维数组表示，由于数组属于静态结构，其空间规模须事先定义——1..max

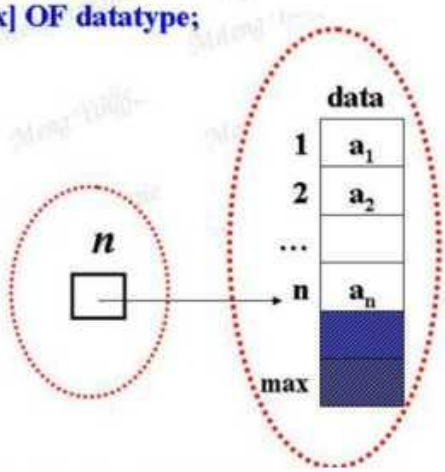
存储图示：



4.向量存储的形式化描述：

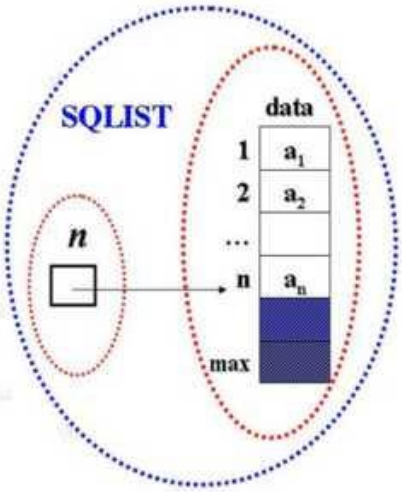
形式化描述：

```
TYPE  SQLIST:  ARRAY [1..max] OF datatype;
VAR  n: 0..max;
```



4.向量存储的形式化描述：

或者，存储结构的形式化描述：




```
TYPE  SQLIST=RECORD
            data: ARRAY [1..max] OF datatype;
            n: 0..max
          END;
```

5. 向量表示下的线性表中数据元素的插入

运算定义：在线性表A=(a_1,a_2,…,a_n)的第 *i-1* 个和第 *i* 个元素之间插入一个新的元素b。

如何对该需求进行算法的设计?

主要的工作阶段:

(1)对问题(即运算或者说需求)的抽象

(2)对需求的满足，即处理过程的分析

运算过程的设计(即处理分析)

◆逻辑上:

A=(a_1,a_2,…, a_{i-1}, a_i, …,a_n)

◆物理上:

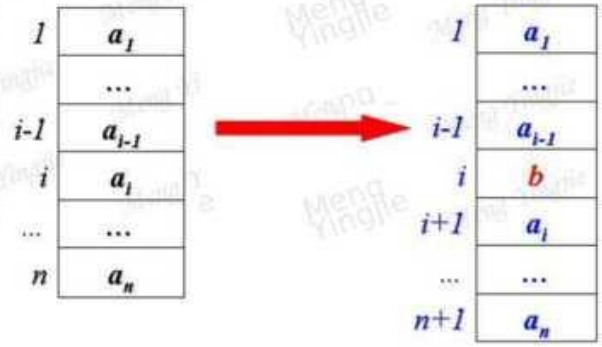
1	a_1
…	…
i-1	a_{i-1}
i	a_i
…	…
n	a_n

运算过程的设计(即处理分析)

◆逻辑上:

$A=(a_1,a_2,\ldots,a_{i-1},a_i,\ldots,a_n)$ → $A=(a_1,a_2,\ldots,a_{i-1},b,a_i,\ldots,a_n)$

◆物理上:



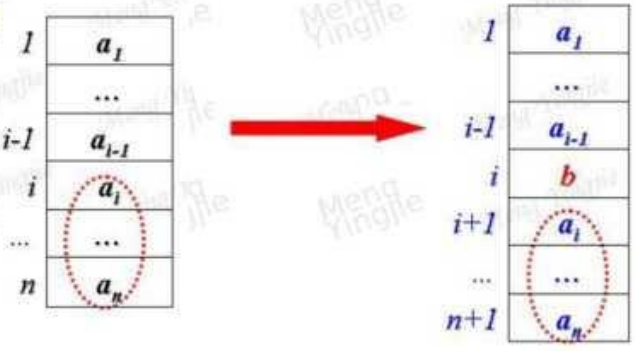
§3 .2 线性表的存储表示 Data Structures

运算过程的设计(即处理分析)

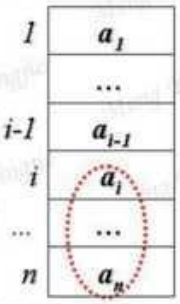
◆逻辑上:

$A=(a_1,a_2,\ldots,a_{i-1},a_i,\ldots,a_n)$ → $A=(a_1,a_2,\ldots,a_{i-1},b,a_i,\ldots,a_n)$

◆物理上:



插入运算过程的步骤归纳：



(1)移动元素，留出空位。

(2)进行插入。

向下求精

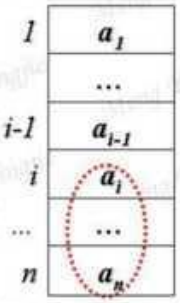
(1) 第a_n个元素到第a_i个元素向下移动一个单元

(2) 插入元送入第 i 个单元

向下求精

(1) 从 n 下降到 i 的元素移动 (即须要一个 for downto 循环)

插入运算过程的步骤归纳：



(1)移动元素，留出空位。

(2)进行插入。

向下求精

(1) 第a_n个元素到第a_i个元素向下移动一个单元

(2) 插入元送入第 i 个单元

向下求精

(1) 从 n 下降到 i 的元素移动 (即须要一个 for downto 循环)

(2) 插入元送入第 i 个单元(即须要一个赋值)

插入运算过程的步骤归纳：

(1)移动元素，留出空位。

(2)进行插入。

向下求精

(1) 第a_n个元素到第a_i个元素向下移动一个单元

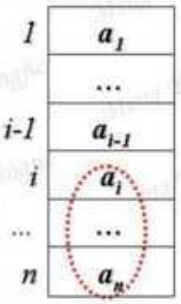
(2) 插入元送入第 i 个单元

向下求精

(1) 从 n 下降到 i 的元素移动 (即须要一个 for downto 循环)

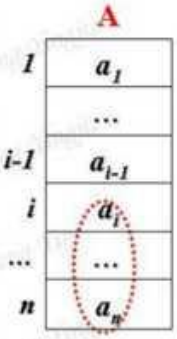
(2) 插入元送入第 i 个单元(即须要一个赋值)

将最终细化后的动作采用PDL指令（即语句）表示即可得到算法过程。



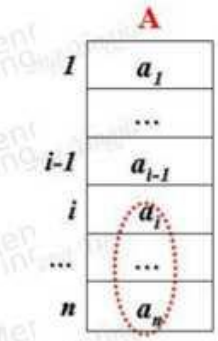
算法过程描述：

```
FOR  j←n  DOWNTO  i  DO
    A[j+1] ←A[j];
A[i] ←b;
```



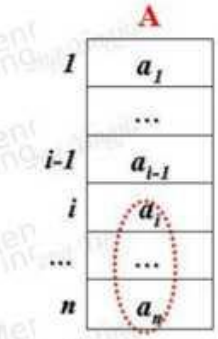
算法过程描述：

```
BEGIN
    FOR j←n DOWNTO i DO
        A[j+1] ←A[j];
    A[i] ←b;
END;
```



算法过程描述：

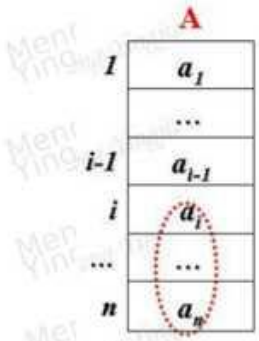
```
PROC InsSQList(VAR A:SQLIST;b:datatype; i,n: integer);
BEGIN
    FOR j←n DOWNTO i DO
        A[j+1] ←A[j];
    A[i] ←b;
END;
```



算法过程描述：

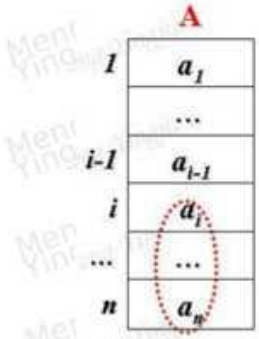
```
PROC InsSQList(VAR A:SQLIST;b:datatype; i,n: integer);
BEGIN
    FOR j←n DOWNTO i DO
        A[j+1] ←A[j];
    A[i] ←b;

END;
```



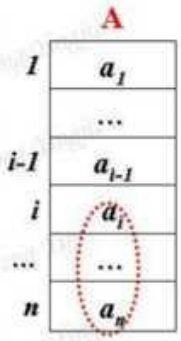
算法过程描述：

```
PROC InsSQList(VAR A:SQLIST;b:datatype; i,n: integer);
BEGIN
    FOR j←n DOWNTO i DO
        A[j+1] ←A[j];
    A[i] ←b;
    n ← n+1
END;
```



算法过程描述：

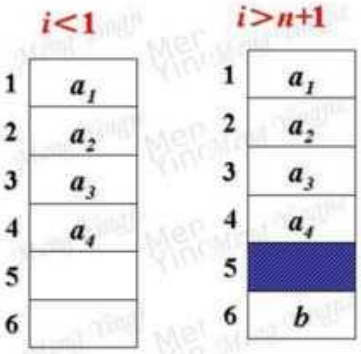
```
PROC InsSQList(VAR A:SQLIST;b:datatype; i,n: integer);
BEGIN
    FOR j←n DOWNTO i DO
        A[j+1] ←A[j];
    A[i] ←b;
    n ← n+1
END;
```



参数异常情况一：

在 $i<1$ 或 $i>n+1$ 下不能插入

即，$1 \leqslant i \leqslant n+1$，可正常插入



参数异常情况二：

$1 \leqslant i \leqslant n+1$

当 n=max，时也不能插入

1	a_1
2	a_2
3	a_3
4	a_4
5	a_5
6	a_6

在算法设计中对于以上的参数异常（即边界条件）导致的运算不能正常执行的情况，我们称为边界情况，也就是说应该对其进行约束。

算法(程序)的设计的大致工作步骤及书写策略：

算法过程体
- xxxxxxxxxxxxxxx
- xxxxxxxxxx

} 相关参数说明

- xxxxxxxxxxxxxx
- xxxxxxxxx
- xxxxxxxxxxxx
- xxxxxxxxxxxxxxx
- xxxxxxxx

} 过程处理语句

算法(程序)的设计的大致工作步骤及书写策略：

(1) 常规(正常情况下)数据处理过程

(2) 边界情况处理(保证健壮性、完备性)

§3 .2 线性表的存储表示 Data Structures

算法(程序)的设计的大致工作步骤及书写策略：

(1) 常规(正常情况下)数据处理过程

(2) 边界情况处理(保证健壮性、完备性)

(3) 将边界情况(作为一种约束)叠加到常规情况

算法(程序)的设计的大致工作步骤及书写策略:

算法过程体
- xxxxxxxxxxxxxxx
- xxxxxxxxxx

} 相关参数说明

- xxxxxxxxxxxxxx
- xxxxxxxxx
- xxxxxxxxxxxx
- xxxxxxxxxxxxxxx
- xxxxxxxx

} 过程处理语句

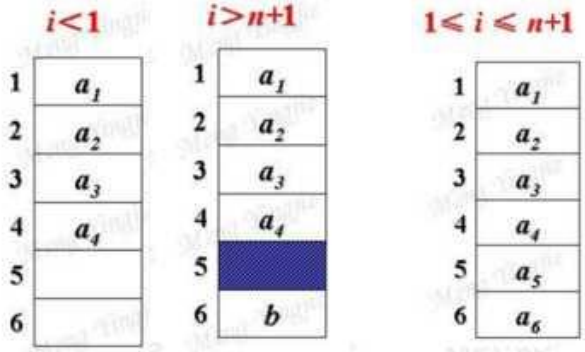
(4) 过程所用相关参数说明

(1) 常规(正常情况下)数据处理过程

(2) 边界情况处理(保证健壮性、完备性)

(3) 将边界情况(作为一种约束)叠加到常规情况

插入运算的边界参数异常情况归纳:



- i<1 或 i>n+1　处理策略: 对 i 报错
- n=max,　处理策略: 提示满，结束算法

插入的算法处理过程：

```
PROC InsSQList(VAR A:SQLIST; b:datatype;
                    i : integer; VAR n :0..max);
 {在线性表A的第i个数据元素之前插入一个新的数据元素b,n为线性表的长度}
BEGIN
  IF i<1 OR i>n+1 THEN 【 WRITE('i error'); EXIT 】;
  IF n=max THEN WRITE('no insert')
           ELSE 【 FOR j← n DOWNTO i DO
                    A[j+1] ← A[j];
                  A[i] ← b;
                  n ← n+1 】
END;
```


6. 线性表中数据元素的删除

操作定义：在线性表A=(a_1,a_2,…,a_n)中删除第 i 个元素。

问题(即需求)的抽象：

概念及表示： DelSQList(A,i,n)

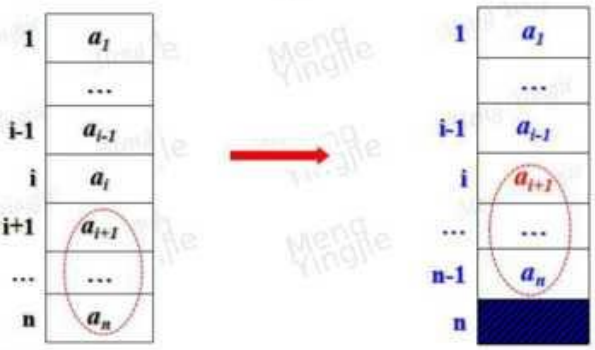
过程本质：一般过程

运算过程的设计(即处理分析)

◆逻辑上:

$A=(a_1,a_2,\ldots,a_{i-1},a_i,\ldots,a_n) \longrightarrow A=(a_1,a_2,\ldots,a_{i-1},a_{i+1},\ldots,a_n)$

◆物理上:



删除算法略

7. 基于向量的运算小结

对于以上两个算法来讲:

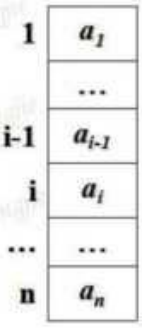
优点: 简单、易随机访问数据元素,存储密度高。

缺点: 需要大量移动数据元素，如果ai是一个文本文件，其时间是相当可观的。需要占用连续存储空间，数据分配只能预先进行(属于静态分配)。

以插入为例进行简单算法分析:

空间复杂度: O(1)

时间复杂度: 最坏情况: O(n)



7. 基于向量的运算小结

对于以上两个算法来讲：

优点：简单、易随机访问数据元素,存储密度高。

缺点：需要大量移动数据元素，如果ai是一个文本文件，其时间是相当可观的。需要占用连续存储空间，数据分配只能预先进行(属于静态分配)。

以插入为例进行简单算法分析：

空间复杂度：O(1)

时间复杂度：平均：等概率($\frac{1}{n+1}$)情况下移动次数为

$$\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)\approx\frac{n}{2}$$

平均时间复杂度为$O(\frac{n}{2})$

1	a_1
	…
i-1	a_{i-1}
i	a_i
…	…
n	a_n

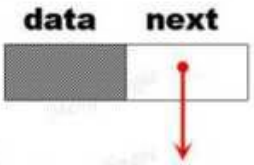
二.线性表的链表表示

链表(chain)：通过指针联系起来的结点的整体(集合)。

链表是用以存储数据的另一种常用存储结构，存储结构中结点之间的关系利用指针而不是存储位置。

1.单链表表示

单指针结点图示

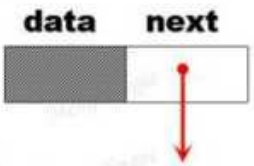


形式化描述：

```
TYPE  pointer=↑node;
      node=RECORD
             data: datatype;
             next: pointer
           END;
      link=pointer;
```


1.单链表表示

单指针结点图示



形式化描述：

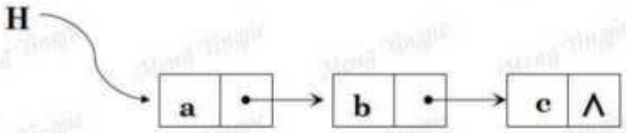
```
TYPE  pointer=↑node;
      node=RECORD
             data: datatype;
             next: pointer
           END;
      link=pointer;
```

类C中：

```
struct pointer {
        datatype data;
        struct pointer *next;};
struct pointer link;
```

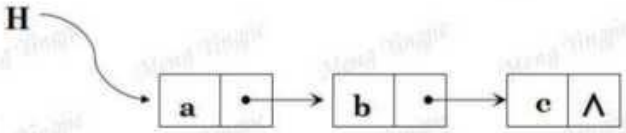
存储示例：

线性表A=(a,b,c)在单链表表示下的存储示意图。

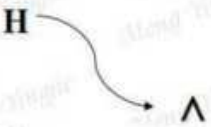


存储示例：

线性表A=(a,b,c)在单链表表示下的存储示意图。

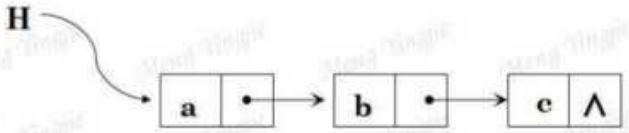


线性表为空：

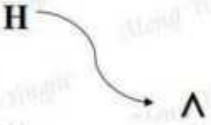


存储示例：

线性表A=(a,b,c)在单链表表示下的存储示意图。



线性表为空：



下面以插入为例讨论在单链表下运算的实现过程及特点。

(1). 链表中插入数据元素

◆操作定义：将数据元素b插入到单链表H中的第一个数据元素为a的结点之前。

问题(即需求)的抽象：

概念及表示：InsLinkList(H ； a; b);

过程本质：一般过程

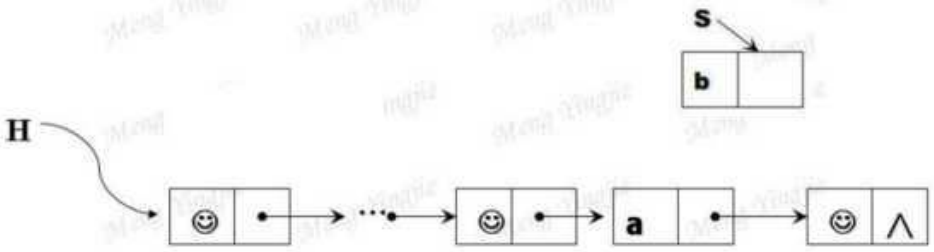
步骤求精，即细化：

①为数据元素b申请空间，并完成数据存储

申请空间过程较为简单，例如，若通过指针S指出所取得的空间：

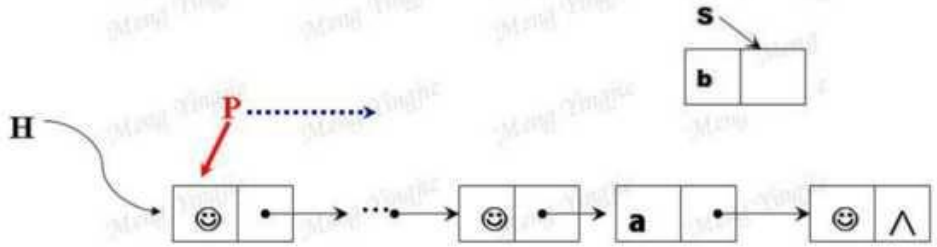
该问题后面讨论的重点放在如何插入上。

◆操作分析：如果插在a之后：

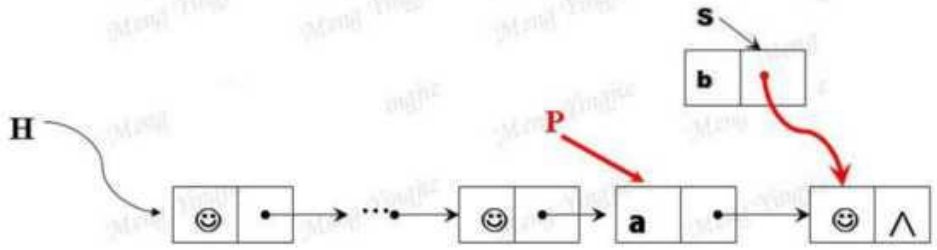


搜索时H可否移动？

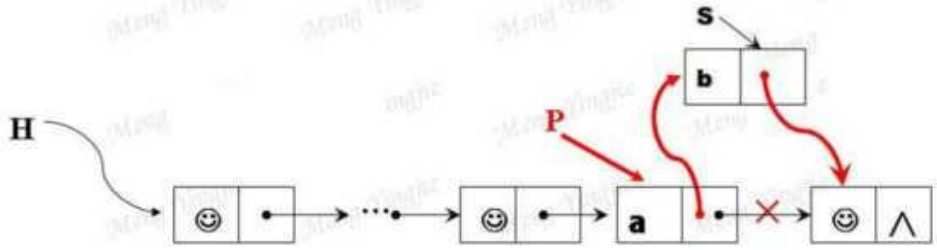
◆操作分析：如果插在a之后:



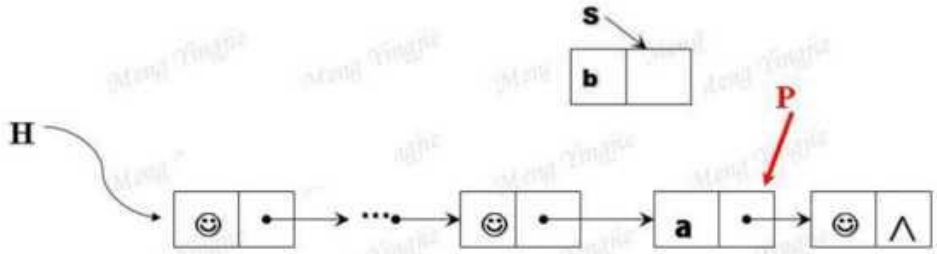
◆操作分析：如果插在a之后:



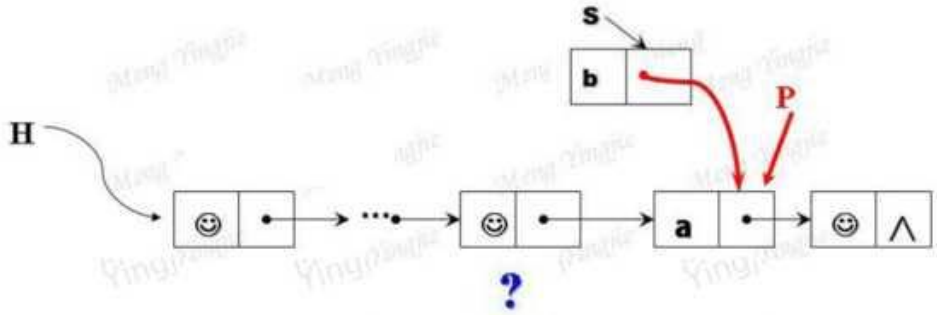
◆操作分析：如果插在a之后：



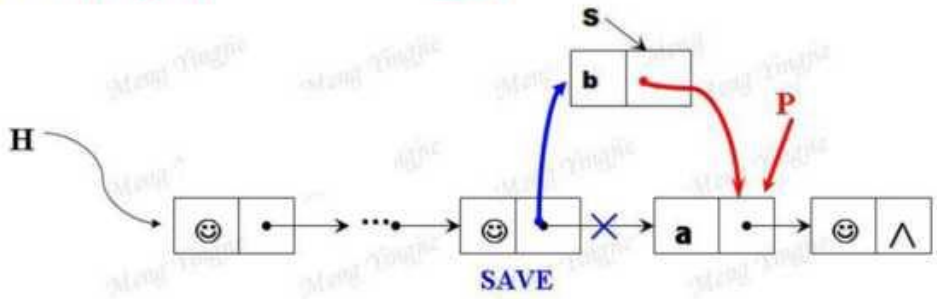
◆操作分析：如果插在a之前：



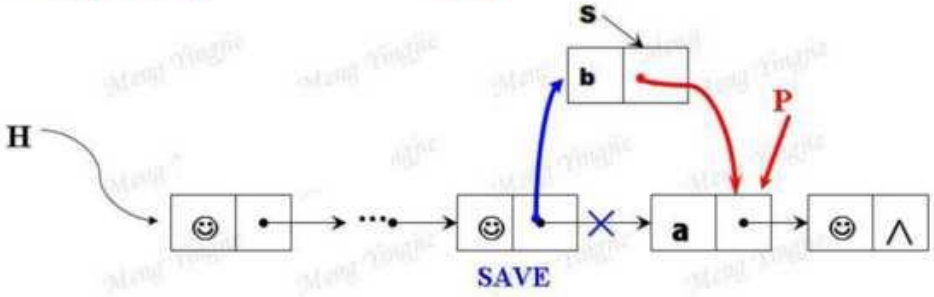
◆操作分析：如果插在a之前：



◆操作分析：如果插在a之前：

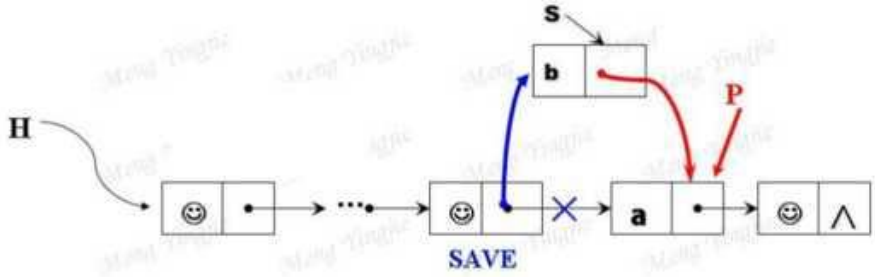


◆操作分析：如果插在a之前：



关键是如何形成两个地址

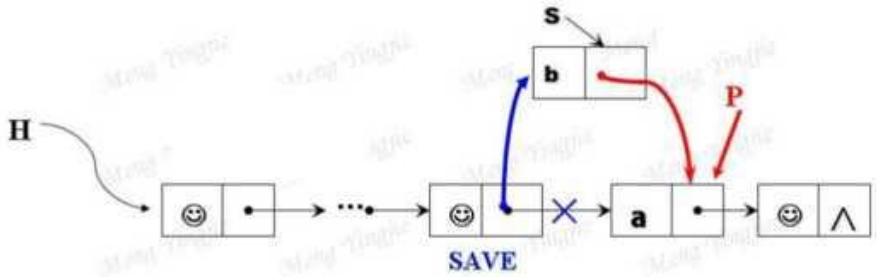
◆操作分析：如果插在a之前：



常规情况：

搜索授权 P←H

◆操作分析：如果插在a之前：



常规情况：

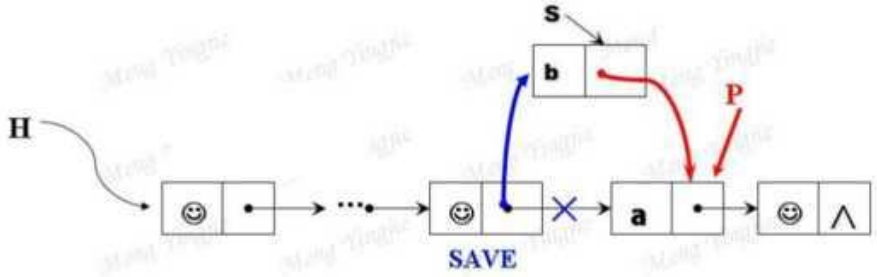
搜索授权 P←H

搜索 WHILE (P↑.data≠a) and (P↑.next≠nil) DO

【 SAVE←P; P←P↑.next 】;

§3 .2 线性表的存储表示 Data Structures

◆操作分析：如果插在a之前：



常规情况：

搜索授权 P←H

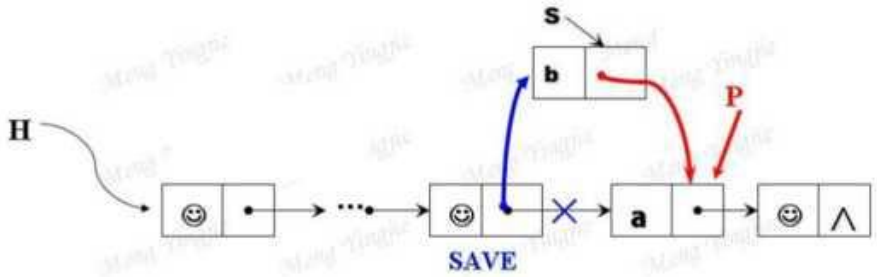
搜索 WHILE (P↑.data≠a) and (P↑.next≠nil) DO

【 SAVE←P; P←P↑.next 】;

插入 IF P↑.data=a THEN 【 S↑.next←P ; SAVE↑.next←S 】

ELSE 【 P↑.next←S ; S↑.next←nil】

◆操作分析：如果插在a之前：



常规情况：

搜索授权 P←H

搜索 WHILE (P↑.data≠a) and (P↑.next≠nil) DO

【 SAVE←P; P←P↑.next 】;

插入 IF P↑.data=a THEN 【 S↑.next←P ; SAVE↑.next←S 】

ELSE 【 P↑.next←S ; S↑.next←nil】

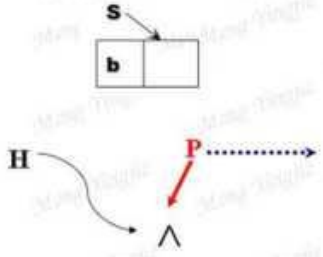
插在a之前的边界情况分析：

(1)若 H=nil 时

指针为空是链表最常见的边界条件之一，

基于正常处理，由于找a初始时 H => P，故P=nil，此时，无法形成搜索判断。

处理策略：直接将S作为链表第一个节点即可



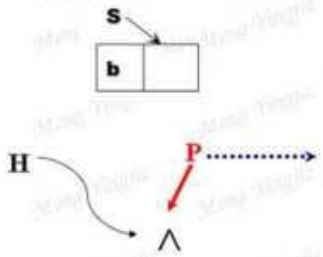
插在a之前的边界情况分析：

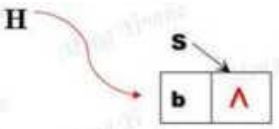
(1)若 H=nil 时

指针为空是链表最常见的边界条件之一，

基于正常处理，由于找a初始时 H => P，故P=nil，此时，无法形成搜索判断。

处理策略：直接将S作为链表第一个节点即可





nil ⟹S↑.next

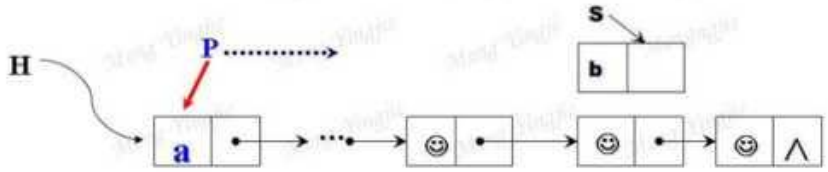
S ⟹H

插在a之前的边界情况分析：

(2) 若 H↑.data=a 时，

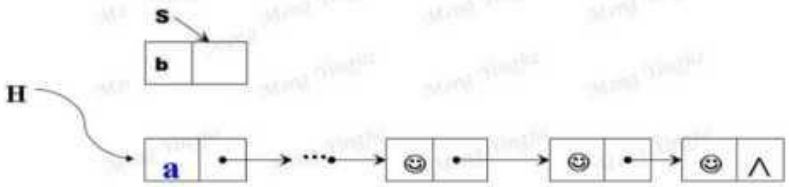
由于平凡情况插入需要使用到指针SAVE，但只有P移动搜索时才会得到SAVE

但第一个就是a时不需要搜索，故SAVE无法形成。

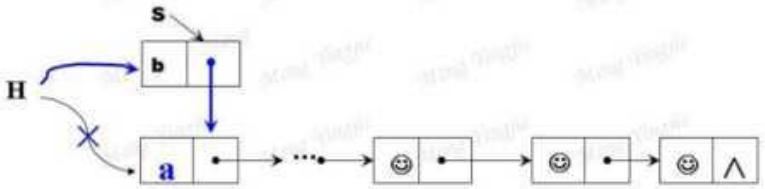


插在a之前的边界情况分析：

(2) 若 H↑.data=a 时，

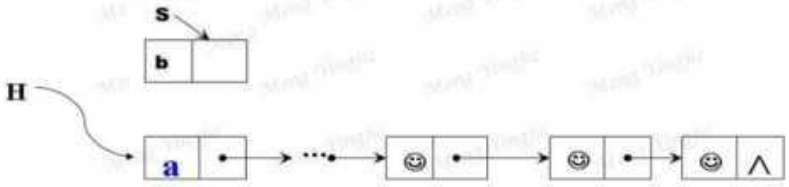


处理策略：直接将S作为链表的第一个节点：

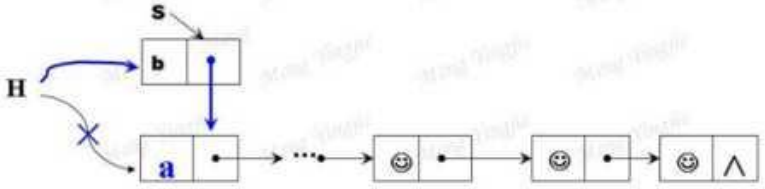


插在a之前的边界情况分析：

(2) 若 H↑.data=a 时，



处理策略：直接将S作为链表的第一个节点：



H==>S↑.next; S ==>H

◆**操作分析**：如果插在a**之前**：

对于前面的分析情况归纳运算处理情况如下：

①H=nil （无法形成搜索循环）

② H↑.data=a （无先导指针save）

③常规(平凡)情况 (非①、②)

将全面过程归纳可得到运算处理的框架：

```
PROC  InsLinkList(H,a,b);
BEGIN
    申请空间，并完成存储;
    CASE
     H=nil 时的处理方法;
     H↑.data=a时的处理方法;
     ELSE 【常规情况】
     ENDCASE
END;
```

将其进行细化、补充填写可得完整的处理过程

基于单链表的插入算法处理过程：

```
PROC InsLinkList(VAR H:link;a,b:datatype);
  {将数据元素b插入到单链表H中第一个数据元素为a的结点之前.}
BEGIN  NEW(S); S↑.data←b;
  CASE
  H=nil : 【 S↑.next←nil;  H←S 】;
  H↑.data=a : 【 S↑.next←H;  H←S 】;
  ELSE 【 P←H;
          WHILE (P↑.data≠a) and (P↑.next≠nil) DO
            【 SAVE←P;  P←P↑.next 】;
          IF P↑.data=a THEN 【 S↑.next←P ; SAVE↑.next←S 】
                       ELSE 【 P↑.next←S ; S↑.next←nil】
       】
  ENDCASE
END;
```


(2).链表中删除数据元素

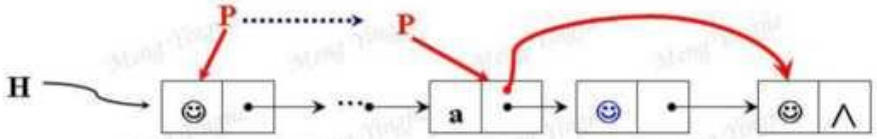
◆操作定义：在单链表H中删除第一个数据元素为a的结点。

（对于删除运算，有些地方其定义是删除第i个，或者删除P的下一个结点，即数据元素a的下一个。）

问题(即需求)的抽象：

DelLinkList(H, a)

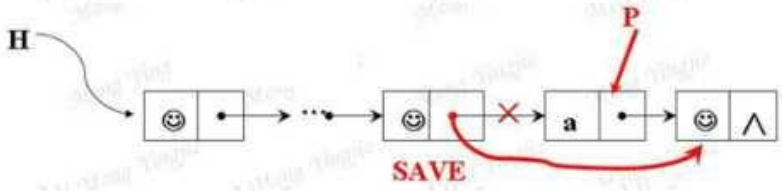
对于删除第 i 个，或者删除p的下一结点，(即a的下一个)的需求分析：



P↑.next<== (P↑.next)↑.next

算法略(注意结点回收)。

对于单链表H中删除第一个数据元素为a的结点：



相对于插入运算来说，寻找a雷同(须设置后拖变量save)，搜索结束后是删除动作。边界条件及处理原理相似。

故，该算法与插入算法原理雷同，具体算法略(注意结点回收)。

2.带表头的单链表表示

◆表头结点(head node):

增加一个附加结点，放置于链表的最前面,也简称头结点。

◆作用和目的:

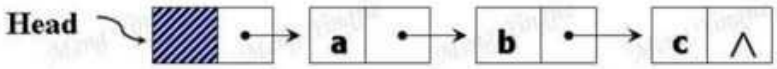
◆头指针与头结点差异

头指针：指向链表起始结点的指针。

头结点：附加在第一数据结点前的结点。

采用带表头结点的单链表示的线性表存储示例:

例1，若表头结点的指针为Head，线性表A=(a,b,c)的存储示意图。



例2，若表头结点的指针为Head，对于空线性表的存储示意图。

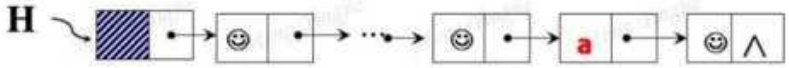
下面通过插入运算，来看一下表头结点的作用，为了比对分析，需求不变，参数符号不变。

◆操作定义：将数据元素b插入到带表头的单链表H中第一个数据元素为a的结点之前。

操作形式化表达：

InsLinkList（H； a; b）

相对于无表头结点的运算差异性分析：



◆无表头结点情况：

①H=nil

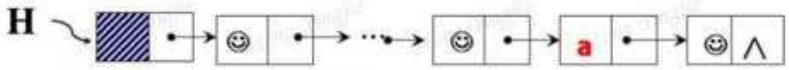
②H↑.data=a

③平凡情况

◆带表头结点情况：

①H↑.next =nil

相对于无表头结点的运算差异性分析：



◆无表头结点情况：

①H=nil

② H↑.data=a

③平凡情况

◆带表头结点情况：

①H↑.next =nil

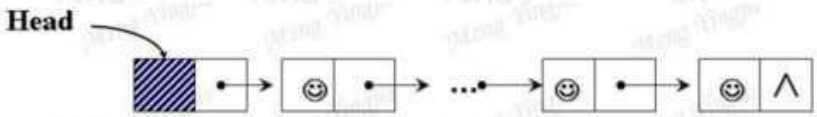
② 平凡情况

比无表头减少了一种情况

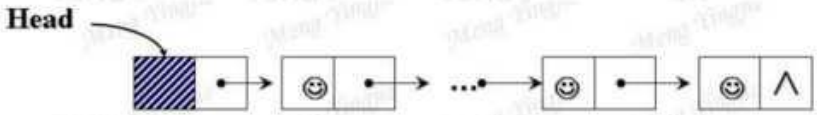
基于带表头结点的单链表插入算法处理过程：

```
PROC  InsLinkList(VAR H:link;a,b:datatype);
  {将数据元素b插入到带表头结点的单链表H中第一个数据元素为a的结点之前。}
BEGIN   NEW(S);  S↑.data←b;
   CASE
   H↑.next =nil : 【 S↑.next←nil;  H↑.next ←S 】;
   ELSE 【 SAVE←H;   P←H↑.next ;
           WHILE (P↑.data≠a) and (P↑.next≠nil) DO
                 【 SAVE←P;   P←P↑.next 】;
           IF P↑.data=a THEN 【 S↑.next←P ;  SAVE↑.next←S 】
                        ELSE 【 P↑.next←S ;  S↑.next←nil】
        】
   ENDCASE
END;
```

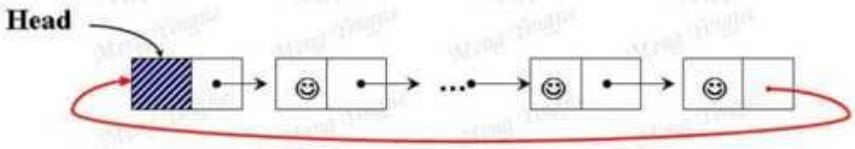
3.带表头结点的循环单链表表示



3.带表头结点的循环单链表表示

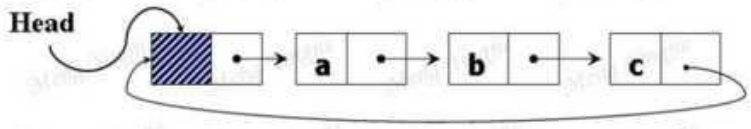


将链表的尾指针空，改为指向表头结点，形成一个逻辑环



存储示例：

例1，若表头结点的指针为Head，线性表A=(a,b,c)的存储示意图。



例2，若表头结点的指针为Head，对于空线性表的存储示意图。

下面通过插入运算，来看一下增加了循环后对运算带来的变化，为了比对分析，需求不变，参数符号不变。

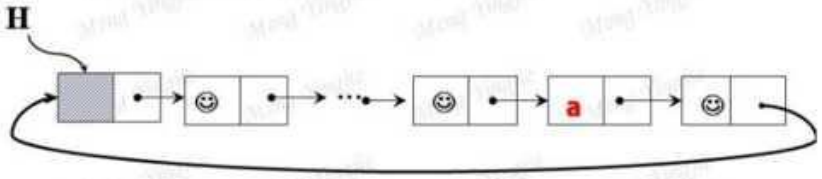
◆操作定义：将数据元素b插入到循环单链表H中的第一个数据元素为a的结点之前。

问题(即需求)的抽象：

概念及表示：InsLinkList (H ; a; b)

过程本质：一般过程

相对于带表头结点的运算差异性分析：



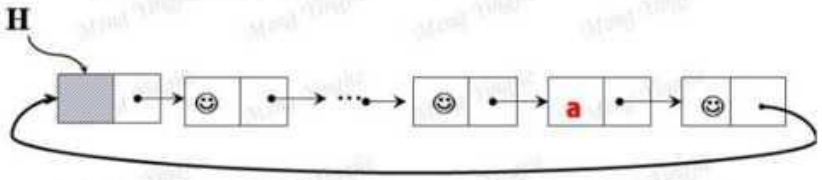
◆带表头结点单链表情况：

① H↑.next =nil ……→ 平凡情况

② 平凡情况

◆带表头结点循环链表情况：

相对于带表头结点的运算差异性分析：



◆带表头结点单链表情况：

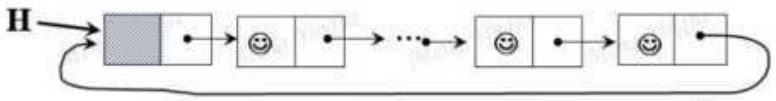
① H↑.next =nil ……→ 平凡情况

② 平凡情况

◆带表头结点循环链表情况：

都成为了平凡情况，但也会带来一些问题，下面简单分析一下：

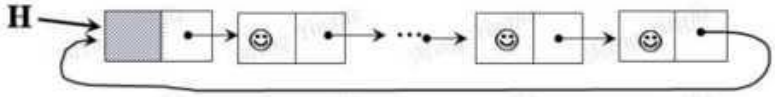
若链表中没有数据元素a该如何处理?



带表头的单链表搜索: **WHILE** (P↑.data≠a) **and** (P↑.next≠nil) **DO**

带表头的循环单链表搜索 **WHILE** (P↑.data≠a) **and** (P↑.next ≠ H) **DO**

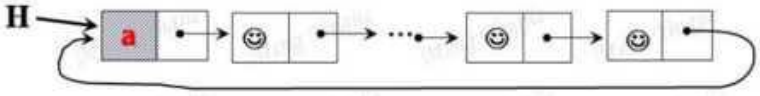
优化



带表头的循环单链表搜索 **WHILE** (P↑.data≠a) **and** (P↑.next ≠ H) **DO**

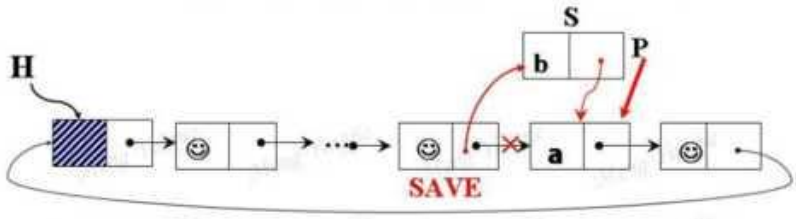
优化

若表头结点中临时存一个a的话，循环会简化，与尾部插入一个结点效果一样。



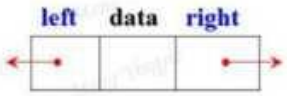
搜索 **WHILE** (P↑.data≠a) **DO**

基于循环单链表的插入算法处理过程

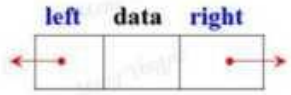



```
PROC  InsCiLinkList(VAR H:link;a,b:datatype);
  {将数据元素b插入到循环单链表H中第一个数据元素为a的结点之前.}
BEGIN    NEW(S);  S↑.data←b;
         H↑.data←a ;  SAVE←H ;   P←H↑.next ;
         WHILE  P↑.data≠a  DO
                【 SAVE←P;  P←P↑.next 】
         S↑.next←P ;   SAVE↑.next←S
END;
```


4.线性表的双向循环链表表示



4.带表头结点的双向循环链表表示

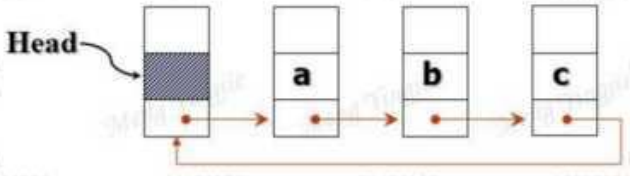


形式化描述：

```
TYPE  pointer=↑node;
      node=RECORD
             left  : pointer;
             data: datatype;
             right : pointer
           END;
      dblink=pointer;
```

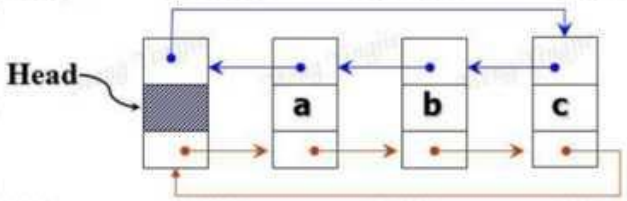

存储示例：

线性表A=(a,b,c)的存储示意图。



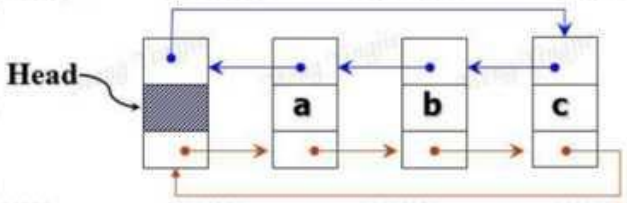
存储示例：

线性表A=(a,b,c)的存储示意图。

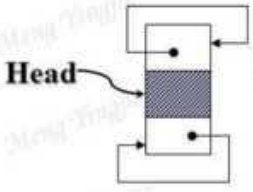


存储示例：

线性表A=(a,b,c)的存储示意图。



线性表为空：

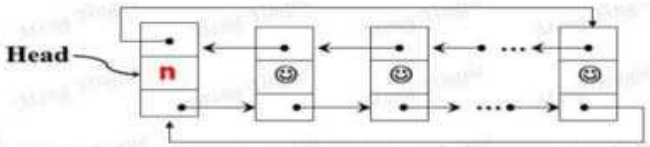


双向链表在检索上的优点：

可实现逆向查找(通过空间代价换取的)

另外，实际使用中双向链表的一些其它变化，例如：

例1. 表头结点的数据项的利用：



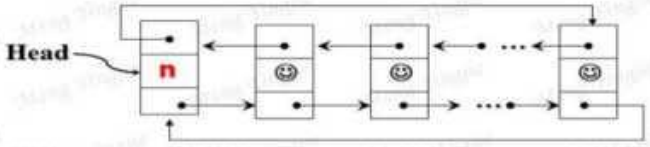
§3 .2 线性表的存储表示 Data Structures

双向链表在检索上的优点：

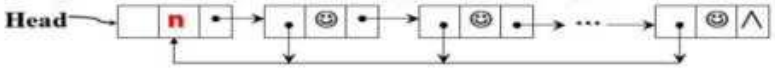
可实现逆向查找(通过空间代价换取的)

另外，实际使用中双向链表的一些其它变化，例如：

例1. 表头结点的数据项的利用：



例2. 指针数据项的变化：

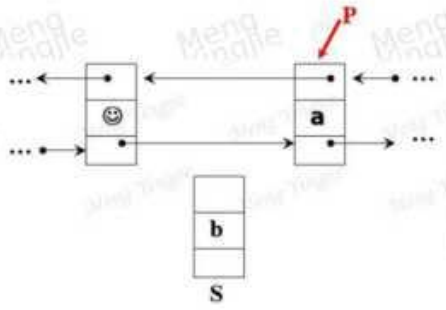


运算过程的设计(即处理分析)

寻找a的过程与前面链表中的过程雷同。

下面重点讨论找到a后的插入动作：

设P为搜索指针，找到数据元素a时的状态如下：

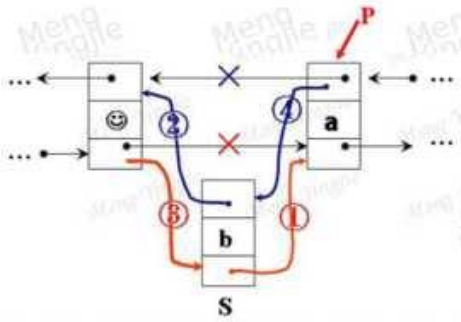


运算过程的设计(即处理分析)

寻找a的过程与前面链表中的过程雷同。

下面重点讨论找到a后的插入动作：

设P为搜索指针，找到数据元素a时的状态如下：



① S↑.right<=P ;

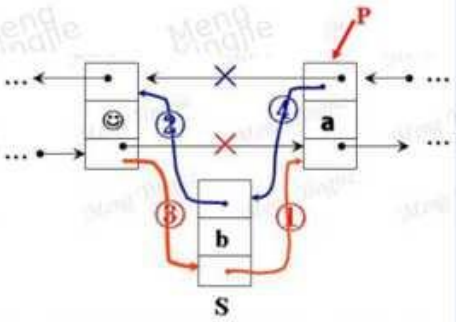
② S↑.left<=P↑.left;

③ P↑.left↑.right<=S;

④ P↑.left<=S;

基于双向循环链表的插入算法处理过程：

```
PROC INSCILINKLIST(VAR H:dblink;a,b:datatype);
{将数据元素b插入到双向循环链表H中第一个数据元素为a的结点之前。}
BEGIN
    NEW(S); S↑.data←b;
    H↑.data←a ; P←H↑.right ;
    WHILE P↑.data≠a DO
        P←P↑.right;
    S↑.right←P ;
    S↑.left←P↑.left;
    P↑.left↑.right←S;
    P↑.left←S
END;
```



5.静态链表表示

静态链表：以整型变量的值作为存储链接指针值(即地址) 联系起来的结点的整体。(指针本质：整数型的数值)

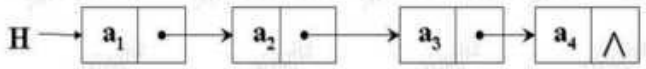
形式化描述：

```
TYPE
LINKLIST=RECORD
            data: ARRAY [1..max] OF datatype;
            next: 0..max
          END;
VAR A:SQLIST;
```

存储示例：

例，线性表A=(a_1,a_2,a_3,a_4) 的存储示意图。

逻辑上呈现以下关系：



	A	
1	a_2	6
2		
3		
4	a_4	0
5		
6	a_3	4
7		
8	a_1	1

6.链接存储的小结

链表结构存储密度低，不能实现随机访问，存储管理复杂。适宜频繁进行插入、删除操作；适合数据规模动态变化情况(属于动态分配)。

空间复杂度：O(1)

算法的时间复杂度主要由查找构成。

时间复杂度：平均：等概率（$\frac{1}{n+1}$）情况下比较次数：

$$\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)\approx\frac{n}{2}$$

平均复杂度：$O(\frac{n}{2})$