

数据结构与算法

人工智能与大数据学院

本节课主要内容

- 顺序表例题讲解
- 线性表的链式存储结构

第2章 线性表——顺序表的例题

- ① 顺序表插入元素问题
- ② 顺序表删除元素问题
- ③ 顺序删除重复元素问题

顺序表插入元素 (插入位置[0,len-1])

1	4	2	7	6	
1	4	new	2	7	6

顺序表插入元素 (插入位置[0,len])

在顺序表中第i个位置处插入元素

```
for (j=len-1; j>= i-1 ; j--)  
    L->data[ j+1] = L->data[ j];  
L->data[ j-1] = x;  
L->last = L->last+1;
```

1	4	2	7	6
---	---	---	---	---

1	4	new	2	7	6
---	---	-----	---	---	---

顺序表插入元素 (插入位置[0,len])

在顺序表中第i个位置处插入元素

```
for (j=len-1; j>= i-1 ; j--)  
    L->data[ j+1] = L->data[ j];  
L->data[ j-1] = x;  
L->last = L->last+1;
```

3

顺序表的插入算法，
最好的情况下的时间复杂度为___；
最坏的情况下的时间复杂度为___

顺序表插入元素 (插入位置[0,len])

在顺序表中第i个位置处插入元素

```
for (j=len-1; j>= i-1 ; j--)  
    L->data[ j+1] = L->data[ j];  
L->data[ i-1] = x;  
L->last = L->last+1;
```

2 下段代码是顺序表插入算法(插入数据元素为data, 插入位置为pos)的主要片段，请将代码补全：

```
int len = L->length,i;  
for(i=len; i>=pos; i--){  
    L->elem[ ] = L->elem[ ];  
}  
L->elem[ ] = data;  
_____  
status=success;
```

顺序表删除元素 (删除位置[0,len-1])

1	4	2	7	6
---	---	---	---	---

1	4	7	6
---	---	---	---

顺序表插入元素（删除位置[0,len-1]）

删除顺序表中第i个位置的元素
for (j=i; j<= len+1 ;j++) {
 L->data[j-1] = L->data[j];
}
L->last = L->last-1;

1	4	2	7	6
---	---	---	---	---

1	4	new	2	7	6
---	---	-----	---	---	---

顺序表插入元素（删除位置[0,len-1]）

删除顺序表中第i个位置的元素
for (j=i; j<= len+1 ;j++) {
 L->data[j-1] = L->data[j];
}
L->last = L->last-1;

5 将下列程序补充完整（顺序表的删除算法）

```
int len = L->length,i;  
if(1<=pos && pos <= ____){  
    for(i=pos; i< ____;i++){  
        L->elem[i]=L->elem[____]  
        ____;  
    }
```

顺序表删除重复元素（有序）

1	2	2	5	7	7
---	---	---	---	---	---

↑ ↑

1	2	5	7	
---	---	---	---	--

顺序表删除重复元素（有序）

1	2	2	5	7	7
---	---	---	---	---	---

1	2	5	7		
---	---	---	---	--	--

```
if (a[src] == a[src-1])  
    src ++;  
else {  
    a[dst] = a[src-1];  
    src ++;  
    dst ++;  
}
```

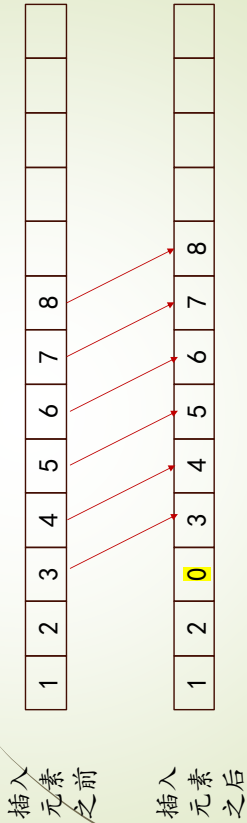
第2章 线性结构——线性表的链式存储

线性表的顺序存储结构是指用一组地址连续的内存单元依次存储线性表中各个数据元素，数据元素之间的线性关系通过存储单元的相邻关系来体现，用这种存储形式存储的线性表称为顺序表。

线性表的链式存储结构是指用一组任意的存储单元（可以连续，也可以不连续）存储线性表中的数据元素。数据元素在存储空间中表示时通常称为结点。

第2章 线性结构——线性表的链式存储

顺序表的缺点：进行插入或删除操作时，平均移动约表中一半的数据元素，对于较长的顺序表而言效率低。



第2章 线性结构——线性表的链式存储

单链表

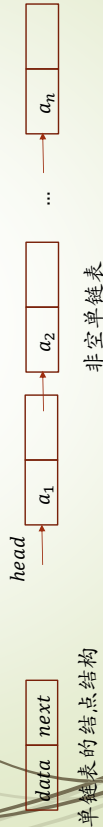
数据元素在存储空间中表示时通常称为结点。

为了能够反映数据元素之间的相邻逻辑关系，每个结点不仅要存放数据元素本身，还需要一些额外的存储空间，用于存放和它有关系的数据元素的地址，即需要存放指向其他元素的指针。

指向第一个结点的指针为头指针，一旦知道头指针，就可以沿着指针依次访问其他数据元素。

单链表中每个结点由两部分组成：数据域和指针域。数据域用于存放数据元素，指针域用于存放数据元素之间的关系，通常用于存放直接后继的地址。

由于每个结点中只有一个指向直接后继的指针，所以称其为单链表



说明：非空单链表中的箭头仅仅表示结点之间的逻辑关系，并不是实际的存储位置

第2章 线性结构——线性表的链式存储

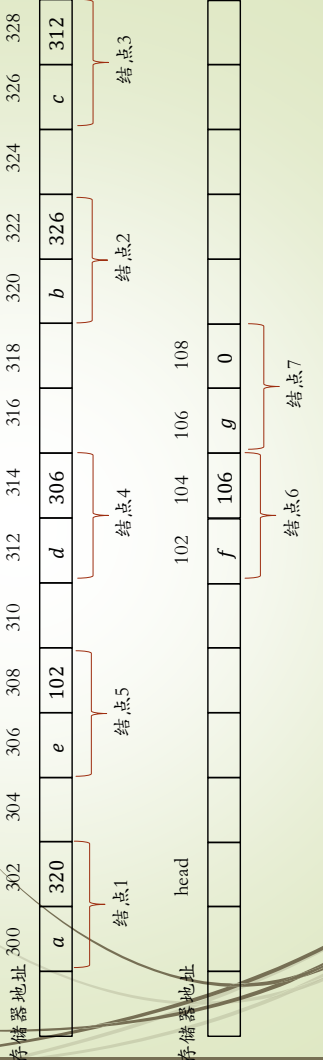
链表的分类

- a. 根据结点中指针数量的多少，可以将链表分为单链表、双链表和多重链表
- b. 根据是否在链表的第一个元素前附加额外的结点，可以将链表分为带头结点的链表和不带头结点的链表
- c. 根据头指针是指向第一个结点，还是指向最后一个结点，可以将链表分为带头指针的链表和带尾指针的链表等

第2章 线性结构——线性表的链式存储

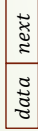
单链表的存储

对于线性表（“a” “b” “c” “d” “e” “f” “g”），假设每个字符占用2个字节，每个指针占用2个字节，存储器按照字节编址，则单链表在计算机存储器中的一种可能情况如下图所示。结点在存储空间中的地址可以相邻，如结点6和结点7，也可以不相邻如其他结点。



第2章 线性结构——线性表的链式存储

单链表



单链表的结点结构

单链表结点结构的类型在C语言中可定义如下：

```
typedef struct node {
    ElemType data;
    struct node *next;
} Listnode, *ListNodePtr;
typedef ListNodePtr List, *ListPtr;
```

在上述类型定义中，我们定义了3个数据类型：struct node, ListNode, *ListNodePtr, 定义不同数据类型的主要目的是提高算法的可读性，同时也使得某些表达较简单。

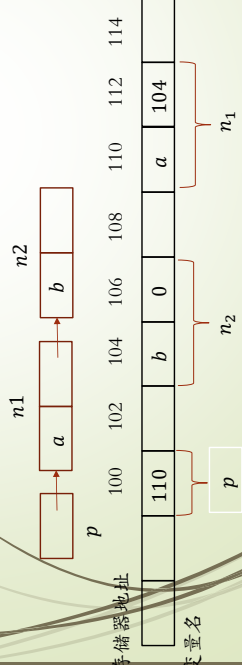
第2章 线性结构——线性表的链式存储

单链表

下面的代码定义了3个变量：

```
ListNode n1, n2; /* 定义2个结点变量 */
ListNodePtr p=&n1; /* 定义一个指向结点n1的指针变量p */
n1.next = &n2; /* 结点n1的指针域存放结点n2的地址 */
```

下图表示了指针变量和结点变量之间的关系，以及在存储器中一种可能的存储方式



第2章 线性结构——线性表的链式存储

顺序表的缺点：进行插入或删除操作时，平均移动约表中一半的数据元素，对于较长的顺序表而言效率低

链表的优点：进行插入或删除操作时，无须移动数据元素

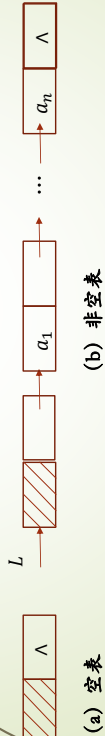
顺序表的优点：可以按照序号随机存取表中的元素

链表的缺点：失去了随机存取数据元素的功能

第2章 线性结构——单链表的基本操作

为了方便，对单链表进行操作之前，通常在第一个结点前增加一个称为头结点的结点。该头结点具有和其他结点相同的数据类型，其中的数据域通常不用，指针域用来存放第一个结点的地址。

在**带头结点的单链表**(a_1, a_2, \dots, a_n)中，头指针指向头结点，如果线性表为空，则头结点的后继为空，可表示为 $L \rightarrow \text{next} = \text{NULL}$ ；否则，头结点的后继为第一个结点



第2章 线性结构——单链表的基本操作

(1) 单链表的查找操作

一种方式是**按位置查找**，即在给定的单链表 L 中，查找指定位置的数据元素，如果存在，则返回 success ，同时取回相应结点的数据。
和顺序表不同，链表的操作只能从**头指针出发，顺着指针域 next 逐个结点比较**，直到搜索到指定位置的结点为止。

```
1 Status List_Retrieve(ListPtr L, int pos, ElemType *elem){
2     Status status = range_error;
3     ListNodePtr p = (*L) -> next;
4     int i = 1;
5     while(p && i < pos){
6         i++;
7         p = p->next;
8     }
9     if(p && i == pos){
10        *elem = p->data;
11        status = success;
12    }
13    return status;
14 }
```

查找数据元素/结点

顺序表：



单链表：



第2章 线性结构——单链表的基本操作

(1) 单链表的查找操作

查找操作从第一个结点开始，依次访问单链表的结点，因此，查找时间最长的情况是指定位置为最后一个结点，或者指定的位置超过线性表长度，指针须遍历整个单链表中的所有结点，故单链表定位查找的最坏时间复杂度为 $O(n)$ 。而顺序表的定位查找效率更高

第2章 线性结构——单链表的基本操作

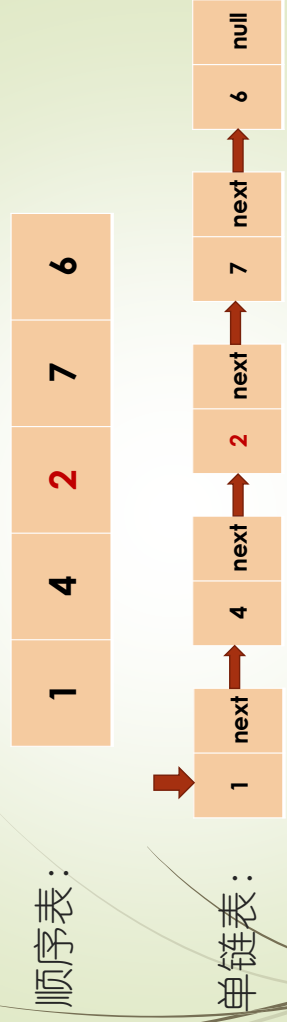
(1) 单链表的查找操作

另一种方式是按值查找，在查找时，也是从单链表的头指针出发，将单链表中的结点逐个和给定值进行比较，直到找到所需数据元素（查找成功），或到达单链表尾部（查找失败）。

```
1 Status List_Locate(ListPtr L, ElemType elem, int *pos){
2     Status status = range_error;
3     ListNodePtr p = *L -> next; /* p指向第一个元素结点*/
4     int i = 1; /* 计数器 */
5     while(p != NULL){
6         if(p -> data == elem) break; /* 当p指向的结点存在时，才能向下比较 */
7         i++; /* 如果找到指定数据元素，则跳出while循环 */
8         p = p -> next; /* 指针后移，同时计数器加1 */
9     }
10    if(p){
11        *pos = i;
12        status = success;
13    }
14    return status;
15 }
```

上述算法的时间复杂度也是 $O(n)$ ，和顺序表按值查找算法的时间复杂度相同

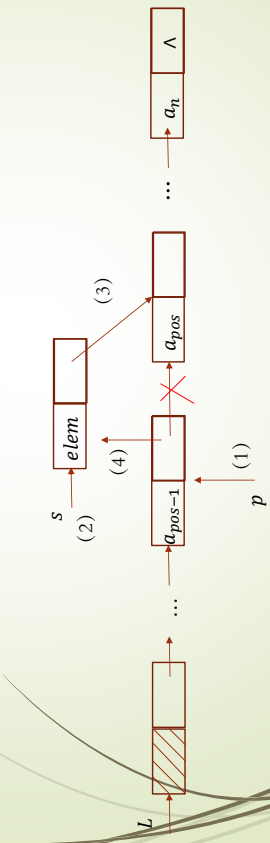
查找数据元素/结点



第2章 线性结构——单链表的基本操作

(2) 单链表的插入操作

插入操作是指将值为elem的新结点插入到单链表中第pos个结点的位置上，即 a_{pos-1} 和 a_{pos} 之间。为了实现这个操作，必须找到 a_{pos-1} 的位置，然后构造一个数据域为elem的新结点，将其挂在单链表上，操作过程如下图所示。



插入数据元素/结点

顺序表:



单链表:



new next

new

```
1 Status List_SetPosition(ListPtr L, int pos, ListNodePtr *ptr){
2     /* 单链表的指针定位算法 (返回指向第pos个结点的指针) */
3     Status status;
4     ListNodePtr p = *L;
5     int i = 0;
6     while(p && i < pos){
7         i++;
8         p = p->next;
9     }
10    if(p && i == pos){
11        *ptr = p;
12        status = success;
13    }
14    return status;
15 }
16
17 Status List_Insert(ListPtr L, int pos, ElemType elem){
18     Status status;
19     ListNodePtr pre, s;
20     status = List_SetPosition(L, pos-1, &pre); /* 找插入位置的前驱 */
21     if(status == success){
22         s = (ListNodePtr)malloc(sizeof(ListNode)); /* 给新结点分配空间 */
23         if(s){
24             s->data = elem;
25             s->next = pre->next;
26             pre->next = s;
27             status = fatal;
28         }
29         return status;
30     }
```

第2章 线性结构——单链表的基本操作

(2) 单链表的插入操作

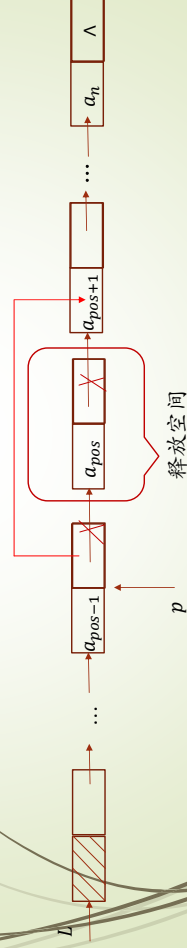
插入算法的时间复杂度为 $O(n)$ 。基本语句是指针移动，而不是数据元素移动。一般来说，数据元素移动的时间消耗要比指针移动的时间消耗大的多，上述算法仅仅移动指针而没有移动数据元素，因此实际运行速度是很快的。

插入数据元素而不移动数据元素也是链表相对于顺序表的一大特点。

第2章 线性结构——单链表的基本操作

(3) 单链表的删除操作

如果要删除单链表中的第 pos 个结点，只需修改第 $pos-1$ 个结点的后继为第 $pos+1$ 结点的地址。因此必须首先求得第 $pos-1$ 结点的地址并用指针 p 指向该地址，然后再释放第 pos 个结点所占的存储空间。删除过程指针变化情况如下图所示。



删除数据元素/结点

顺序表：



单链表：



(3) 单链表的删除操作

```
1 Status List_Remove(ListPtr L, int pos){
2     Status status;
3     ListNodePtr ptr, q;
4     status = List_SetPosition(L, pos-1, &ptr); /* 找插入位置的前驱 */
5     if(status == success){
6         q = ptr->next;
7         ptr->next = q->next;
8         free(q);
9         q=NULL;
10    }
11    return status;
12 }
```

上面算法的时间主要耗费在位置的查找上，如果以指针移动作为基本语句，其时间复杂度为 $O(n)$ 。同单链表的结点插入算法一样，单链表的结点删除操作也不需要移动数据元素，运行起来是很高效的。

(4) 单链表的创建操作

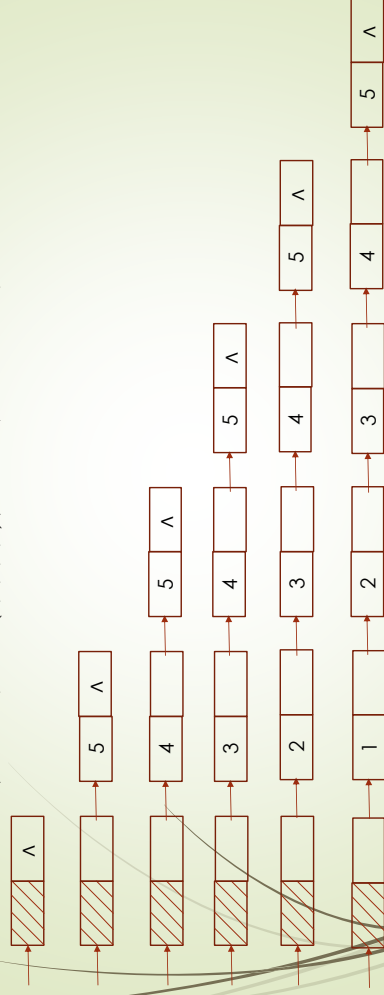
创建单链表的过程域创建顺序表的过程不同，单链表结点所占空间不是一次分配或预先划定的，而是根据结点个数不同即时生成的，并按需分配空间。

单链表的创建可以通过其他操作来实现，例如，首先创建一个空的单链表，然后依次动态生成元素结点，并逐个插入链表而得。数据的插入有两种方法，从链表头部开始插入和从链表尾部开始插入。由于从链表尾部开始插入需要跟踪尾部结点的位置。

因此使用，从头部开始插入结点的方法建立单链表，此时要求数据元素以相反的顺序读入，每读入一个数据，就为其创建一个结点，并将其插入链表的头部。

(4) 单链表的创建操作

例如，创建单链表(1,2,3,4,5)，则头部插入创建单链表过程实现：



(4) 单链表的创建操作

例如，创建单链表(1,2,3,4,5)，则头部插入创建单链表过程实现：

```
1 Status List_Creat(ListPtr L, ElemType elem[], int n){
2     Status status = success;
3     ListNodePtr p, q;
4     int i = n-1;
5     q = (ListNodePtr)malloc(sizeof(ListNode));
6     q->next = NULL;
7     while(i >= 0){
8         p = (ListNodePtr)malloc(sizeof(ListNode));
9         if(!p){status = fatal; break;}
10        p->data=elem[i];
11        p->next=q->next;
12        q->next=p;
13        i = i-1;
14    }
15    *L = q;
16    return status;
17 }
```

(5) 单链表的其他操作

(初始化、销毁、清空、判空)

```
1 Status List_Init(ListPtr L){
2     Status status = fatal;
3     *L = (ListNodePtr)malloc(sizeof(ListNode));
4     if(*L){(*L)->next=NULL; status=success;}
5     return status;
6 }
7
8 void List_Destroy(ListPtr L){
9     List_Clear(L);
10    free(*L);
11 }
12
13 void List_Clear(ListPtr L){
14     ListNodePtr p = *L, q = p->next;
15     while(q){
16         p->next=q->next;
17         free(q);
18         q=p->next;
19     }
20 }
21
22 bool List_Empty(ListPtr L){
23     return(*L->next == NULL);
24 }
25 }
```

单链表的基本操作

与顺序表不同，除初始化 (List_Init) 和判空 (List_Empty) 外，几乎所有单链表的操作都涉及指针的大量移动 (p=p->next)，如果把指针移动作为基本语句，则它们的时间复杂度都是O(n)。

需要注意的是，不同情况下时间复杂度可能是不同的。对于插入和删除操作，如果已知被操作结点的前驱指针，则它们的操作仅仅需要修改有限的几个指针即可，时间复杂度是O(1)，而不像顺序表那样需要移动大量的数据元素。即使考虑指针移动所花费的时间，因为指针移动仅仅是简单变量（相当于整型变量）的赋值操作，和移动数据元素相比，也能节省很多时间，这正是**链式存储结构的一个重要优势**

(6) 单链表的其他操作 (求长度、求结点的前驱、 求结点的后继)

```
26 int List_Size(ListPtr L){
27     int length = 0;
28     ListNodePtr p = (*L) ->next;
29     while(p){
30         length ++;
31         p = p->next;
32     }
33     return length;
34 }
35 Status List_Prior(ListPtr L, int pos, ElemType *elem){
36     Status status;
37     ListNodePtr ptr;
38     status = List_SetPosition(L, pos-1, &ptr);
39     if(status == success){
40         *elem = ptr->data;
41     }
42     return status;
43 }
44 Status List_Next(ListPtr L, int pos, ElemType *elem){
45     Status status;
46     ListNodePtr ptr;
47     status = List_SetPosition(L, pos+1, &ptr);
48     if(status == success){
49         *elem = ptr->data;
50     }
51     return status;
52 }
```