2.3 线性表的链式表示和实现

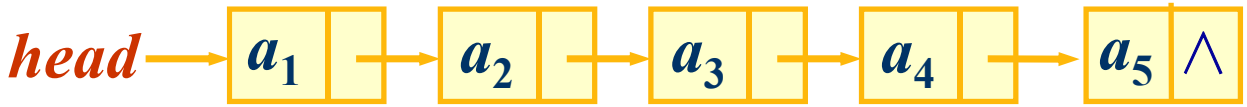
2.3.1 线性链表

线性表的单链表存储方法：用一个指针表示结点间的逻辑关系。因此单链表的一个存储结点包含两个部分，结点的形式如下：

- data：为数据域，用于存储线性表的一个数据元素，也就是说在单链表中一个结点存放一个数据元素。
- next：为指针域或链域，用于存放一个指针，该指针指向后继元素对应的结点，也就是说单链表中结点的指针用于表示后继关系。

- 特点
  - 每个数据元素由结点(Node)构成。

  - 线性结构



  - 结点可以连续，可以不连续存储。
  - 结点的逻辑顺序与物理顺序可以不一致。
  - 表可扩充。
  - 链表中第一个元素结点称为首结点；最后一个元素称为尾结点，其指针域(next)为空(NULL)。

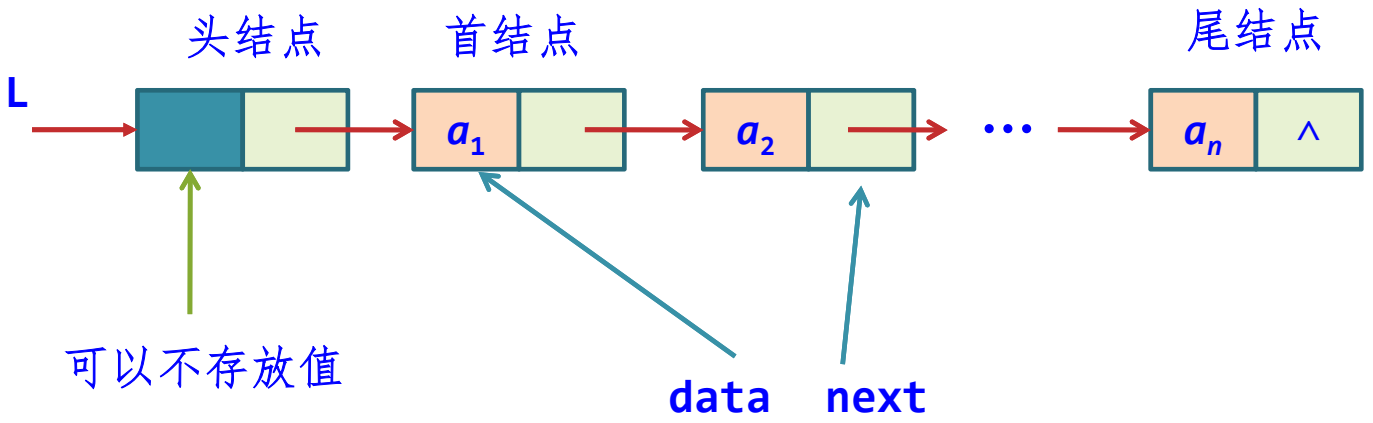
- 单链表分为带头结点和不带头结点两种类型。在许多情况下，带头结点的单链表能够简化运算的实现过程。
- 因此在后续讨论的单链表，除特别指出外均指带头结点的单链表。

◆ 假设数据元素的类型为ElemType。单链表的结点类型声明如下：

```
typedef int ElemType;
typedef int Status;
#define OK  1
#define ERROR 0
// ------ 结点的C语言描述 --------
typedef struct node {
    ElemType   data;
    struct node  *next;
} LNode, *LinkList;
```

2.3.1.1 线性表基本运算在单链表上的实现

带头结点的单链表示意图



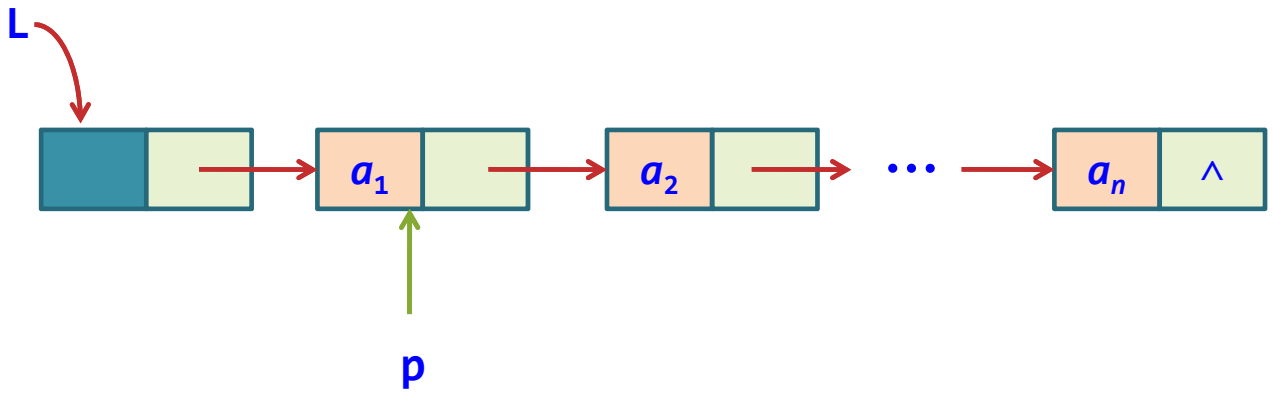
（1）初始化单链表算法

创建一个空的单链表，它只有一个头结点，由L指向它。该结点的next域为空，data域未设定任何值。对应的算法如下：

```
Status InitList(LinkList &L)
{ // 操作结果：构造一个空的线性表L
    // 产生头结点,并使L指向此头结点
    L = (LinkList)malloc(sizeof(LNode));
    if(!L)                          // 存储分配失败
        return ERROR;
    L->next = NULL;                 // 指针域为空
    return OK;
}
```

（2）销毁单链表算法

一个单链表中的所有结点空间都是通过malloc函数分配的，在不再需要时需通过free函数释放所有结点的空间。



```
Status DestroyList(LinkList &L)
{
    // 初始条件：线性表L已存在。操作结果：销毁线性表L
    LinkList p;
    while(L) {
        p = L->next;
        free(L);
        L = p;
    }
    return OK;
}
```

（3）求单链表长度算法

设置一个整型变量i作为计数器，i初值为0，p初始时指向第一个数据结点。然后沿next域逐个往后查找，每移动一次，i值增1。当p所指结点为空时，结束这个过程，i之值即为表长。

```
int ListLength(LinkList L)
{
    // 初始条件：线性表L已存在。操作结果：返回L中数据元素个数
    int i = 0;
    LinkList  p = L->next;      // p指向第一个结点
    while(p) {                  // 没到表尾
        i++;
        p = p->next;
    }
    return i;
}
```

（4）求单链表中值为e的元素算法

- 用*p*从头开始遍历单链表*L*中的结点，用计数器*i*统计遍历过的结点，其初值为0。
- 在遍历时，若*p*不为空，则*p*所指结点即为要找的结点，查找成功，算法返回位序i。
- 否则算法返回0表示未找到这样的结点。

```
int LocateElem(LinkList L, ElemType e)
{   // 操作结果: 返回L中第1个与e相等的数据元素的位序。
    // 若这样的数据元素不存在,则返回值为0
    int i = 0;
    LinkList p = L->next;
    while(p) {
        i++;
        if( p->data == e))  // 找到这样的数据元素
            return i;
        p = p->next;
    }
    return 0;
}
```

（5）单链表的插入算法

◆ 在单链表L中第i个位置，插入值为x的结点。

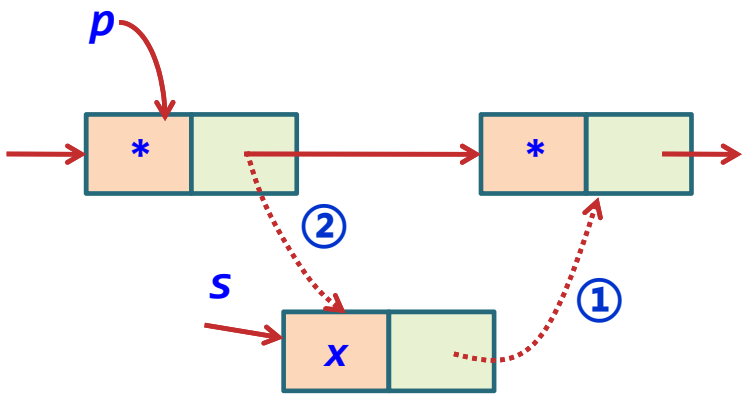
- 先在单链表L中查找第i-1个结点，若未找到返回0；
- 找到后由p指向该结点，创建一个以x为值的新结点s，将其插入到p指结点之后。

插入操作

在*p*结点之后插入*s*结点的操作如下：

① 将结点*s*的next域指向*p*的下一个结点（*s*->next=*p*->next）。

② 将结点*p*的next域改为指向新结点*s*（*p*->next=*s*）。



注意： 插入操作的①和②执行顺序不能颠倒。

```
Status ListInsert_L(LinkList &L, int i, ElemType e)
{
    // 在带头结点的单链线性表L的第i个元素之前插入元素e
    LinkList p, s;
    p = L;
    int j = 0;
    while(p && j < i - 1) {   // 寻找第i-1个结点
        p = p->next;
        ++j;
    }
    if(!p || j > i - 1)
        return ERROR;           // i小于1或者大于表长
    s = (LinkList)malloc(sizeof(LNode));  // 生成新结点
    s->data = e;
    s->next = p->next;          // 插入L中
    p->next = s;
    return OK;
}
```

（6）单链表的删除算法

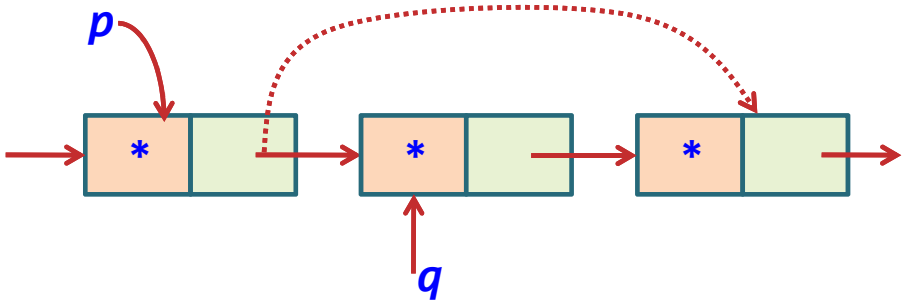
◆ 在单链表L中删除第i个结点。

- 先在单链表L中查找第i-1个结点，若未找到返回0。
- 找到后由p指向该结点，然后让q指向后继结点（即要删除的结点）。
- 若q所指结点为空则返回0，否则删除q结点并释放其占用的空间。

删除操作：

✓ 删除p指结点的后继结点的过程：

```
p->next=q->next;
free(q);
```



```
Status ListDelete_L(LinkList &L, int i, ElemType &e)
{
    // 在带头结点的单链线性表L中，删除第i个元素，并由e返回其值
    LinkList p, q;
    p = L;
    int j = 0;
    // 寻找第i个结点，并令p指向其前趋
    while(p->next && j < i - 1) {
        p = p->next;
        ++j;
    }
    if(!(p->next) || j > i - 1)
        return ERROR;                    // 删除位置不合理
    q = p->next;
    p->next = q->next;                   // 删除并释放结点
    e = q->data;
    free(q);
    return OK;
}
```

END