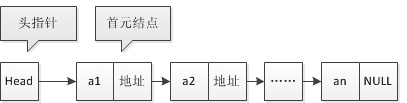


* 头指针具有标识作用，故常用头指针冠以链表的名字。
* 无论链表是否为空，头指针均不为空。[头指针](http://www.nowamagic.net/librarys/veda/tag/%E5%A4%B4%E6%8C%87%E9%92%88)是链表的必要元素。

单链表也可以没有头结点。如果没有头结点的话，那么单链表就会变成这样：

* 

1. //定义结点的结构体
2. //typedef struct LNode{
3. //  int data;
4. //  struct LNode \*next;
5. //}Node,\*LinkList;//则定义LinkList L;时，L为链表的头指针。
6. //
7. //L=(LinkList) malloc (sizeof(LNode)); //创建一个结点，此处返回给L的是一个指针，并且赋给了头指针。
8. //L->next=null; //这里说明我创建了一个头结点，即同时运用了头指针和头结点。
9. //这么方便只要加上->next就说明创建了头节点。不过想想头指针是没有next的，只有头结点才有，所以就不难理解了

12. //带头结点初始化
13. //Node \*head;  //声明头结点
14. //首先第一眼看到(\*head)->next=NULL;和我们刚才所说一样，只要头指针一旦运用了next操作就自动创建了头结点
15. //但是我们今天的重点不在于这个，更多在于Node \*\*head，对于两个指针的操作理解
16. //一级指针是\*head，表明了二级指针head，head这个指针变量存储的是另外一个指向结构体Node的指针，即是一级指针，即\*head作为一个整体
17. //\*head是一级指针，其指向的内容就是结构体Node。经过这么一理解，对于头指针，头结点，首元节点的关系就非常明朗了
18. //  void InitList(Node \*\*head){
19. //      \*head=(Node \*)malloc( sizeof(Node));//这里需要明确的第一点，申请内存返回的都是地址
20. //      //第二点就是（Node \*）表明了其返回的指针指向最后的结果是Node的结构体，如果是指向int，那么我们就写(int \*)
21. //      (\*head)->next=NULL;
22. //}

25. //带头结点方便了首元节点和其他节点一样，统一了操作
26. //方式一：
27. //  void CreatList(Node \*\*head){
28. //      Node \*r=\*head,\*s;//因为前面已经对头结点，头指针初始化过了，因此可以直接使用\*head
29. //      int a;
30. //      while(scanf("%d",&a)){
31. //          if(a!=0){
32. //              s=(Node \*)malloc(sizeof(Node));
33. //              s->value=a;
34. //              r->next=s;//这里没有着急设置s->next，原因在于后续还要插入数据。因此将s赋值给r
35. //              r=s;
36. //          }
37. //          else{
38. //              r->next=NULL;//如果后续输入的数据为空，则就将其设置为null
39. //              break;
40. //          }
41. //      }
42. //}
43. //调用CreatList(&head);//这句话表明了形参Node \*\*head，只有第一个\*才是起作用了，第二个\*号是和head联系在一起，作为整体使用的

46. //方式二：
47. //  void CreatList(Node \*head){
48. //      Node \*r=head,\*s;
49. //      ... //下面的都一样
50. //}
51. //调用CreatList(head)；
52. //
54. //不带头结点初始化
55. //方式一：
56. //void InitList(Node \*\*head){
57. //      \*head=NULL;//这里直接就是指向的首元节点，还有之前自己一个误解，看到head就觉得它就是头指针了，其实它就是随便一个指针变量
58. //并不是像自己之前想的那样的

61. //从这里才发现，真正有无头节点的区别。
62. //\*head=(Node \*)malloc( sizeof(Node));//这里需要明确的第一点，申请内存返回的都是地址
63. //第二点就是（Node \*）表明了其返回的指针指向最后的结果是Node的结构体，如果是指向int，那么我们就写(int \*)
64. //(\*head)->next=NULL;
65. //}
66. //调用InitList(&head)；
67. //
68. //方式二：
69. //void InitList(Node \*head){
70. //      head=NULL;
71. //}
72. //调用InitList(head)；
74. //不带头结点尾插入，第一个节点与其他节点分开操作
75. //void CreatList(Node  \*\*head){
76. //      Node \*p,\*t;         /\*p工作指针,t临时指针\*/
77. //      int a,i=1;
78. //      while(scanf("%d",&a)){
79. //          if(a!=0){
80. //              t=(Node \*)malloc(sizeof(Node));
81. //              t->value=a;
82. //              if(i==1){
83. //                  \*head=t;
84. //              }
85. //              else{
86. //                  p->next=t;
87. //              }
88. //              p=t;
89. //          }
90. //          else{
91. //              p->next=NULL;
92. //              break;
93. //          }
94. //          i++;
95. //      }
96. //}
97. //调用CreatList(&head)；
98. //其实从上面就可以知道，其实有头结点对于我们来说是一种更加明智更加方便的操作

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

//这里插入关于链表中有无头节点进行初始化链表知识

//首先明白头指针与头结点的关系：http://www.nowamagic.net/librarys/veda/detail/1805

//定义结点的结构体

//typedef struct LNode{

// int data;

// struct LNode \*next;

//}LNode,\*LinkList;//则定义LinkList L;时，L为链表的头指针。

//

//L=(LinkList) malloc (sizeof(LNode)); //创建一个结点，此处返回给L的是一个指针，并且赋给了头指针。

//L->next=null; //这里说明我创建了一个头结点，即同时运用了头指针和头结点。

//这么方便只要加上->next就说明创建了头节点。不过你想想头指针是没有next的，只有头结点才有，所以就不难理解了

//带头结点初始化

//Node \*head; //声明头结点

//首先第一眼看到(\*head)->next=NULL;和我们刚才所说是不是一样，只要头指针一旦运用了next操作就自动创建了头结点

//但是我们今天的重点不在于这个，更多在于Node \*\*head，对于两个指针的操作理解

//第一个指针\*head，表明了head这个指针变量存储的是另外一个指向结构体NODE的指针，第二个指针，即\*head作为一个整体

//其结果是一个指针，其指向的内容就是结构体NODE。经过这么一理解，对于头指针，头结点，首元节点的关系就非常明朗了

// void InitList(Node \*\*head){

// \*head=(Node \*)malloc( sizeof(Node));//这里需要明确的第一点，申请内存返回的都是地址

// //第二点就是（Node \*）表明了其返回的指针指向最后的结果是NODE的结构体，如果是指向int，那么我们就写(int \*)

// (\*head)->next=NULL;

//}

//带头结点尾方便了首元节点和其他节点一样，统一了操作

//方式一：

// void CreatList(Node \*\*head){

// Node \*r=\*head,\*s;//因为前面已经对头结点，头指针初始化过了，因此可以直接使用\*head

// int a;

// while(scanf("%d",&a)){

// if(a!=0){

// s=(Node \*)malloc(sizeof(Node));

// s->value=a;

// r->next=s;//这里没有着急设置s->next，原因在于后续还要插入数据。因此将s赋值给r

// r=s;

// }

// else{

// r->next=NULL;//如果后续输入的数据为空，则就将其设置为null

// break;

// }

// }

//}

//调用CreatList(&head);//这句话表明了形参Node \*\*head，只有第一个\*才是起作用了，第个\*号是和head联系在一起，作为整体使用的

//方式二：

// void CreatList(Node \*head){

// Node \*r=head,\*s;

// ... //下面的都一样

//}

//调用CreatList(head)；

//

//不带头结点初始化

//方式一：

//void InitList(Node \*\*head){

// \*head=NULL;//这里直接就是指向的首元节点，还有之前自己一个误解，看到head就觉得它就是头指针了，其实它就是随便一个指针变量

//并不是像自己之前想的那样的

//从这里才发现，真正有无头节点的区别。

//\*head=(Node \*)malloc( sizeof(Node));//这里需要明确的第一点，申请内存返回的都是地址

//第二点就是（Node \*）表明了其返回的指针指向最后的结果是NODE的结构体，如果是指向int，那么我们就写(int \*)

//(\*head)->next=NULL;

//}

//调用InitList(&head)；

//

//方式二：

//void InitList(Node \*head){

// head=NULL;

//}

//调用InitList(head)；

//不带头结点尾插入，第一个节点与其他节点分开操作

//void CreatList(Node \*\*head){

// Node \*p,\*t; /\*p工作指针,t临时指针\*/

// int a,i=1;

// while(scanf("%d",&a)){

// if(a!=0){

// t=(Node \*)malloc(sizeof(Node));

// t->value=a;

// if(i==1){

// \*head=t;

// }

// else{

// p->next=t;

// }

// p=t;

// }

// else{

// p->next=NULL;

// break;

// }

// i++;

// }

//}

//调用CreatList(&head)；

//其实从上面就可以知道，其实有头结点对于我们来说是一种更加明智更加方便的操作

一、两者区别：

     1、不带头结点的单链表对于第一个节点的操作与其他节点不一样，需要特殊处理，这增加了程序的复杂性和出现bug的机会，因此，**通常在单链表的开始结点之前附设一个头结点。**

**2、带头结点的单链表,初始化时一定返回的是指向头结点的地址，所以一定要用二维指针，否则将导致内存访问失败或异常。**

     3、带头结点与不带头结点初始化、插入、删除、输出操作都不样，在遍历输出链表数据时，带头结点的判断条件是while(head->next!=NULL)，而不带头结点是while(head!=NULL)，虽然头指针可以在初始时设定，但是如1所述，对于特殊情况如只有一个节点会出现问题。

二、为什么不带头结点初始化有2种方式，而带头结点只有1种方式呢？

**因为不带头结点声明Node \*head 时；C编译器将其自动初始化为NULL，于是根本不需要调用InitList(head)；也即不带头结点的初始化是个伪操作。而带头结点的初始化在堆开辟了一段内存，需要修改head指针变量指向的地址(即head的值)，所以要修改head的值，必须传保存head变量的地址(即二维指针)。而直接调用CreatList(head);相当于传head变量的值，函数修改的是head的副本，无法真正改变head的值。**

**其实这个就是上面所说的第二点内容**

三、其实本质上还是传值，传址的问题，只不过指针本身保存的地址，让这个过程变得有点纠结。在函数调用需要修改指针变量的指向(value)时，应该传递指针变量的地址(address)。

      另外，对于函数的形参是指针时，只要该参数不在左边（即都是右值操作），二维指针(形参)就可以简化为一维指针。如上面带头结点的尾插简化版本。