注意，以下对链表的操作实现均建立在已创建好链表的基础上，创建链表的代码如下所示：

1. //声明节点结构
2. **typedef** **struct** Link{
3. int elem;//存储整形元素
4. **struct** Link \*next;//指向直接后继元素的指针
5. }link;
6. //创建链表的函数
7. link \* initLink(){
8. link \* p=(link\*)malloc(**sizeof**(link));//创建一个头结点
9. link \* temp=p;//声明一个指针指向头结点，用于遍历链表
10. //生成链表
11. **for** (int i=1; i<5; i++) {
12. //创建节点并初始化
13. link \*a=(link\*)malloc(**sizeof**(link));
14. a->elem=i;
15. a->next=NULL;
16. //建立新节点与直接前驱节点的逻辑关系
17. temp->next=a;
18. temp=temp->next;
19. }
20. **return** p;
21. }

从实现代码中可以看到，该链表是一个具有头节点的链表。由于头节点本身不用于存储数据，因此在实现对链表中数据的"增删查改"时要引起注意。

**链表插入元素**

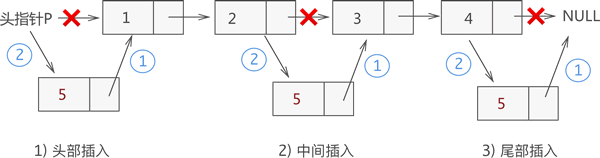
同[顺序表](http://c.biancheng.net/view/3334.html" \t "_blank)一样，向链表中增添元素，根据添加位置不同，可分为以下 3 种情况：

* 插入到链表的头部（头节点之后），作为首元节点；
* 插入到链表中间的某个位置；
* 插入到链表的最末端，作为链表中最后一个数据元素；

虽然新元素的插入位置不固定，但是链表插入元素的思想是固定的，只需做以下两步操作，即可将新元素插入到指定的位置：

1. 将新结点的 next 指针指向插入位置后的结点；
2. 将插入位置前结点的 next 指针指向插入结点；

例如，我们在链表 {1,2,3,4} 的基础上分别实现在头部、中间部位、尾部插入新元素 5，其实现过程如图 1 所示：

  
图 1 链表中插入元素的 3 种情况示意图

从图中可以看出，虽然新元素的插入位置不同，但实现插入操作的方法是一致的，都是先执行步骤 1 ，再执行步骤 2。

注意：链表插入元素的操作必须是先步骤 1，再步骤 2；反之，若先执行步骤 2，除非再添加一个指针，作为插入位置后续链表的头指针，否则会导致插入位置后的这部分链表丢失，无法再实现步骤 1。

通过以上的讲解，我们可以尝试编写 C 语言代码来实现链表插入元素的操作：

1. //p为原链表，elem表示新数据元素，add表示新元素要插入的位置
2. link \* insertElem(link \* p, int elem, int add) {
3. link \* temp = p;//创建临时结点temp
4. //首先找到要插入位置的上一个结点
5. **for** (int i = 1; i < add; i++) {
6. temp = temp->next;
7. **if** (temp == NULL) {
8. printf("插入位置无效\n");
9. **return** p;
10. }
11. }
12. //创建插入结点c
13. link \* c = (link\*)malloc(**sizeof**(link));
14. c->elem = elem;
15. //向链表中插入结点
16. c->next = temp->next;
17. temp->next = c;
18. **return** p;
19. }

**链表删除元素**

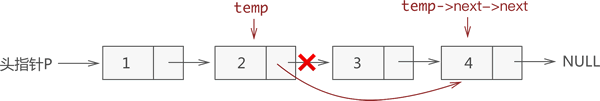
从链表中删除指定数据元素时，实则就是将存有该数据元素的节点从链表中摘除，但作为一名合格的程序员，要对存储空间负责，对不再利用的存储空间要及时释放。因此，从链表中删除数据元素需要进行以下 2 步操作：

1. 将结点从链表中摘下来;
2. 手动释放掉结点，回收被结点占用的存储空间;

其中，从链表上摘除某节点的实现非常简单，只需找到该节点的直接前驱节点 temp，执行一行程序：

temp->next=temp->next->next;

例如，从存有 {1,2,3,4} 的链表中删除元素 3，则此代码的执行效果如图 2 所示：

  
图 2 链表删除元素示意图

因此，链表删除元素的 C 语言实现如下所示：

1. //p为原链表，add为要删除元素的值
2. link \* delElem(link \* p, int add) {
3. link \* temp = p;
4. //遍历到被删除结点的上一个结点
5. **for** (int i = 1; i < add; i++) {
6. temp = temp->next;
7. **if** (temp->next == NULL) {
8. printf("没有该结点\n");
9. **return** p;
10. }
11. }
12. link \* del = temp->next;//单独设置一个指针指向被删除结点，以防丢失
13. temp->next = temp->next->next;//删除某个结点的方法就是更改前一个结点的指针域
14. free(del);//手动释放该结点，防止内存泄漏
15. **return** p;
16. }

我们可以看到，从链表上摘下的节点 del 最终通过 free 函数进行了手动释放。

**链表查找元素**

在链表中查找指定数据元素，最常用的方法是：从表头依次遍历表中节点，用被查找元素与各节点数据域中存储的数据元素进行比对，直至比对成功或遍历至链表最末端的 NULL（比对失败的标志）。  
  
因此，链表中查找特定数据元素的 C 语言实现代码为：

1. //p为原链表，elem表示被查找元素、
2. int selectElem(link \* p,int elem){
3. //新建一个指针t，初始化为头指针 p
4. link \* t=p;
5. int i=1;
6. //由于头节点的存在，因此while中的判断为t->next
7. **while** (t->next) {
8. t=t->next;
9. **if** (t->elem==elem) {
10. **return** i;
11. }
12. i++;
13. }
14. //程序执行至此处，表示查找失败
15. **return** -1;
16. }

注意，遍历有头节点的链表时，需避免头节点对测试数据的影响，因此在遍历链表时，建立使用上面代码中的遍历方法，直接越过头节点对链表进行有效遍历。

**链表更新元素**

更新链表中的元素，只需通过遍历找到存储此元素的节点，对节点中的数据域做更改操作即可。  
  
直接给出链表中更新数据元素的 C 语言实现代码：

1. //更新函数，其中，add 表示更改结点在链表中的位置，newElem 为新的数据域的值
2. link \*amendElem(link \* p,int add,int newElem){
3. link \* temp=p;
4. temp=temp->next;//在遍历之前，temp指向首元结点
5. //遍历到待更新结点
6. **for** (int i=1; i<add; i++) {
7. temp=temp->next;
8. }
9. temp->elem=newElem;
10. **return** p;
11. }

**总结**

以上内容详细介绍了对链表中数据元素做"增删查改"的实现过程及 C 语言代码，在此给出本节的完整可运行代码：

1. #include <stdio.h>
2. #include <stdlib.h>
3. **typedef** **struct** Link {
4. int elem;
5. **struct** Link \*next;
6. }link;
7. link \* initLink();
8. //链表插入的函数，p是链表，elem是插入的结点的数据域，add是插入的位置
9. link \* insertElem(link \* p, int elem, int add);
10. //删除结点的函数，p代表操作链表，add代表删除节点的位置
11. link \* delElem(link \* p, int add);
12. //查找结点的函数，elem为目标结点的数据域的值
13. int selectElem(link \* p, int elem);
14. //更新结点的函数，newElem为新的数据域的值
15. link \*amendElem(link \* p, int add, int newElem);
16. void display(link \*p);
17. int main() {
18. //初始化链表（1，2，3，4）
19. printf("初始化链表为：\n");
20. link \*p = initLink();
21. display(p);
22. printf("在第4的位置插入元素5：\n");
23. p = insertElem(p, 5, 4);
24. display(p);
25. printf("删除元素3:\n");
26. p = delElem(p, 3);
27. display(p);
28. printf("查找元素2的位置为：\n");
29. int address = selectElem(p, 2);
30. **if** (address == -1) {
31. printf("没有该元素");
32. }
33. **else** {
34. printf("元素2的位置为：%d\n", address);
35. }
36. printf("更改第3的位置上的数据为7:\n");
37. p = amendElem(p, 3, 7);
38. display(p);
39. **return** 0;
40. }
41. link \* initLink() {
42. link \* p = (link\*)malloc(**sizeof**(link));//创建一个头结点
43. link \* temp = p;//声明一个指针指向头结点，用于遍历链表
44. //生成链表
45. **for** (int i = 1; i < 5; i++) {
46. link \*a = (link\*)malloc(**sizeof**(link));
47. a->elem = i;
48. a->next = NULL;
49. temp->next = a;
50. temp = temp->next;
51. }
52. **return** p;
53. }
54. link \* insertElem(link \* p, int elem, int add) {
55. link \* temp = p;//创建临时结点temp
56. //首先找到要插入位置的上一个结点
57. **for** (int i = 1; i < add; i++) {
58. temp = temp->next;
59. **if** (temp == NULL) {
60. printf("插入位置无效\n");
61. **return** p;
62. }
63. }
64. //创建插入结点c
65. link \* c = (link\*)malloc(**sizeof**(link));
66. c->elem = elem;
67. //向链表中插入结点
68. c->next = temp->next;
69. temp->next = c;
70. **return** p;
71. }
72. link \* delElem(link \* p, int add) {
73. link \* temp = p;
74. //遍历到被删除结点的上一个结点
75. **for** (int i = 1; i < add; i++) {
76. temp = temp->next;
77. **if** (temp->next == NULL) {
78. printf("没有该结点\n");
79. **return** p;
80. }
81. }
82. link \* del = temp->next;//单独设置一个指针指向被删除结点，以防丢失
83. temp->next = temp->next->next;//删除某个结点的方法就是更改前一个结点的指针域
84. free(del);//手动释放该结点，防止内存泄漏
85. **return** p;
86. }
87. int selectElem(link \* p, int elem) {
88. link \* t = p;
89. int i = 1;
90. **while** (t->next) {
91. t = t->next;
92. **if** (t->elem == elem) {
93. **return** i;
94. }
95. i++;
96. }
97. **return** -1;
98. }
99. link \*amendElem(link \* p, int add, int newElem) {
100. link \* temp = p;
101. temp = temp->next;//tamp指向首元结点
102. //temp指向被删除结点
103. **for** (int i = 1; i < add; i++) {
104. temp = temp->next;
105. }
106. temp->elem = newElem;
107. **return** p;
108. }
109. void display(link \*p) {
110. link\* temp = p;//将temp指针重新指向头结点
111. //只要temp指针指向的结点的next不是Null，就执行输出语句。
112. **while** (temp->next) {
113. temp = temp->next;
114. printf("%d ", temp->elem);
115. }
116. printf("\n");
117. }

代码运行结果：

初始化链表为：  
1 2 3 4  
在第4的位置插入元素5：  
1 2 3 5 4  
删除元素3:  
1 2 5 4  
查找元素2的位置为：  
元素2的位置为：2  
更改第3的位置上的数据为7:  
1 2 7 4