之前接触到的[链表](http://data.biancheng.net/view/5.html" \t "_blank)都只有一个指针，指向直接后继，整个[链表](http://data.biancheng.net/view/160.html)只能单方向从表头访问到表尾，这种结构的链表统称为 “单向链表”或“单链表”。  
  
如果算法中需要频繁地找某结点的前趋结点，单链表的解决方式是遍历整个链表，增加算法的[时间复杂度](http://data.biancheng.net/view/2.html" \t "_blank)，影响整体效率。  
  
为了快速便捷地解决这类问题，在单向链表的基础上，给各个结点额外配备一个指针变量，用于指向每个结点的直接前趋元素。这样的链表被称为“[双向链表](http://data.biancheng.net/view/166.html)”或者“双链表”。

**双链表中的结点**

双向链表中的结点有两个指针域，一个指向直接前趋，一个指向直接后继。（链表中第一个结点的前趋结点为NULL，最后一个结点的后继结点为NULL）

http://data.biancheng.net/uploads/allimg/170718/2-1FGQ62RJT.png

[图](http://data.biancheng.net/view/200.html)1 双向链表中的结点

结点的具体构成：

1. **typedef** **struct** line{
2. **struct** line \* prior; //指向直接前趋
3. int data;
4. **struct** line \* next; //指向直接后继
5. }line;

**创建双向链表并初始化**

双向链表创建的过程中，每一个结点需要初始化数据域和两个指针域，一个指向直接前趋结点，另一个指向直接后继结点。  
  
例如，创建一个双向链表line（1，2，3）：

http://data.biancheng.net/uploads/allimg/170718/2-1FGQ62932M8.png

图2 双向表（1，2，3）

实现代码：

1. line\* initLine(line \* head){
2. head=(line\*)malloc(**sizeof**(line));//创建链表第一个结点（首元结点）
3. head->prior=NULL;
4. head->next=NULL;
5. head->data=1;
6. line \* list=head;
7. **for** (int i=2; i<=3; i++) {
8. //创建并初始化一个新结点
9. line \* body=(line\*)malloc(**sizeof**(line));
10. body->prior=NULL;
11. body->next=NULL;
12. body->data=i;
14. list->next=body;//直接前趋结点的next指针指向新结点
15. body->prior=list;//新结点指向直接前趋结点
16. list=list->next;
17. }
18. **return** head;
19. }

**双向链表中插入结点**

比如在（1，2，3）中插入一个结点 4，变成（1，4，2，3）。  
  
实现效果图：

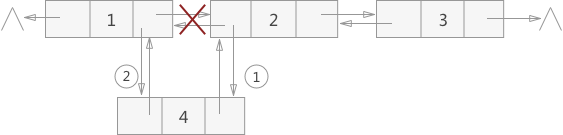


图3 插入结点4

在双向链表中插入数据时，首先完成图 3 中标注为 1 的两步操作，然后完成标注为 2 的两步操作；反之，如果先完成 2，就无法通过头指针访问结点 2，需要额外增设指针，虽然能实现，但较前一种麻烦。

实现代码：

1. line \* insertLine(line \* head,int data,int add){
2. //新建数据域为data的结点
3. line \* temp=(line\*)malloc(**sizeof**(line));
4. temp->data=data;
5. temp->prior=NULL;
6. temp->next=NULL;
7. //插入到链表头，要特殊考虑
8. **if** (add==1) {
9. temp->next=head;
10. head->prior=temp;
11. head=temp;
12. }**else**{
13. line \* body=head;
14. //找到要插入位置的前一个结点
15. **for** (int i=1; i<add-1; i++) {
16. body=body->next;
17. }
18. //判断条件为真，说明插入位置为链表尾
19. **if** (body->next==NULL) {
20. body->next=temp;
21. temp->prior=body;
22. }**else**{
23. body->next->prior=temp;
24. temp->next=body->next;
25. body->next=temp;
26. temp->prior=body;
27. }
28. }
29. **return** head;
30. }

**双向链表中删除节点**

双链表删除结点时，直接遍历链表，找到要删除的结点，然后利用该结点的两个指针域完成删除操作。  
  
例如，在（1，4，2，3）中删除结点 2：

1. //删除结点的函数，data为要删除结点的数据域的值
2. line \* delLine(line \* head,int data){
3. line \* temp=head;
4. //遍历链表
5. **while** (temp) {
6. //判断当前结点中数据域和data是否相等，若相等，摘除该结点
7. **if** (temp->data==data) {
8. temp->prior->next=temp->next;
9. temp->next->prior=temp->prior;
10. free(temp);
11. **return** head;
12. }
13. temp=temp->next;
14. }
15. printf("链表中无该数据元素");
16. **return** head;
17. }

**双向链表中的查找和更改操作**

双向链表的查找操作和单链表的实现方法完全一样，从链表的头结点或者首元结点开始遍历，这里不做过多解释。  
更改链表中某结点的数据域的操作是在查找的基础上完成的。通过遍历找到存储有该数据元素的结点后，直接更改其数据域就可以。

**本节的完整代码**

1. #include <stdio.h>
2. #include <stdlib.h>
3. **typedef** **struct** line{
4. **struct** line \* prior;
5. int data;
6. **struct** line \* next;
7. }line;
8. line\* initLine(line \* head);
9. line \* insertLine(line \* head,int data,int add);
10. line \* delLine(line \* head,int data);
11. void display(line \* head);
12. int main() {
13. line \* head=NULL;
14. head=initLine(head);
16. head=insertLine(head, 4, 2);
17. display(head);
18. head=delLine(head, 2);
19. display(head);
21. **return** 0;
22. }
23. line\* initLine(line \* head){
24. head=(line\*)malloc(**sizeof**(line));
25. head->prior=NULL;
26. head->next=NULL;
27. head->data=1;
28. line \* list=head;
29. **for** (int i=2; i<=3; i++) {
30. line \* body=(line\*)malloc(**sizeof**(line));
31. body->prior=NULL;
32. body->next=NULL;
33. body->data=i;
35. list->next=body;
36. body->prior=list;
37. list=list->next;
38. }
39. **return** head;
40. }
41. line \* insertLine(line \* head,int data,int add){
42. //新建数据域为data的结点
43. line \* temp=(line\*)malloc(**sizeof**(line));
44. temp->data=data;
45. temp->prior=NULL;
46. temp->next=NULL;
47. //插入到链表头，要特殊考虑
48. **if** (add==1) {
49. temp->next=head;
50. head->prior=temp;
51. head=temp;
52. }**else**{
53. line \* body=head;
54. //找到要插入位置的前一个结点
55. **for** (int i=1; i<add-1; i++) {
56. body=body->next;
57. }
58. //判断条件为真，说明插入位置为链表尾
59. **if** (body->next==NULL) {
60. body->next=temp;
61. temp->prior=body;
62. }**else**{
63. body->next->prior=temp;
64. temp->next=body->next;
65. body->next=temp;
66. temp->prior=body;
67. }
68. }
69. **return** head;
70. }
71. line \* delLine(line \* head,int data){
72. line \* temp=head;
73. //遍历链表
74. **while** (temp) {
75. //判断当前结点中数据域和data是否相等，若相等，摘除该结点
76. **if** (temp->data==data) {
77. temp->prior->next=temp->next;
78. temp->next->prior=temp->prior;
79. free(temp);
80. **return** head;
81. }
82. temp=temp->next;
83. }
84. printf("链表中无该数据元素");
85. **return** head;
86. }
87. //输出链表的功能函数
88. void display(line \* head){
89. line \* temp=head;
90. **while** (temp) {
91. **if** (temp->next==NULL) {
92. printf("%d\n",temp->data);
93. }**else**{
94. printf("%d->",temp->data);
95. }
96. temp=temp->next;
97. }
98. }

**总结**

双向链表和单链表唯一的不同在于结构中多了一个指向直接前趋的指针，其他完全一样。如果问题中需要频繁的调取当前结点的前趋结点，那使用双向链表的数据结构为最佳方案。

**补：双向链表和循环链表的结合体**

约瑟夫环问题其实还可以这样玩：如果顺时针报数，有人出列后，顺时针找出出列位置的下一个人，开始反方向（也就是逆时针）报数，有人出列后，逆时针找出出列位置的下一个人，开始顺时针报数。依次重复，直至最后一个出列。  
例如，还是从编号为 3 的开始数，数到 2 的人出列：

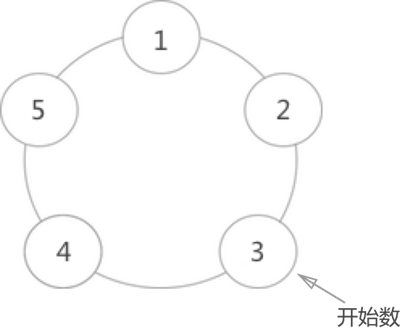


图4 约瑟夫环

新玩法的出列顺序为：  
首先顺时针转，4 数 2，所以 4 出列；  
顺时针找到下一个人为 5，开始逆时针转，3 数 2，所以 3 出列；  
逆时针找到下一个人为 2，开始顺时针转，5 数 2，所以 5 出列；  
顺时针找到下一个人为 1，开始逆时针转，2 数 2，所以 2 出列；  
最后只剩下 1，所以 1 自己出列。      
  
对于新的约瑟夫环问题，需要将循环链表和双向链表结合使用，组成：双向循环链表。