

第二节 线性表-链式存储-链表

回顾数组

时间复杂度：

- 访问：由于可以随机访问， $O(1)$
- 插入： $O(n)$
- 删除： $O(n)$
- 静态操作：常数时间内完成
- 动态操作：线性时间内完成

此外数组还有一个限制：数组大小一经确定不得修改，存在数组满的风险。

可扩容数组实现原理：当数组满后创建一个更大的数组，将原来数组中的内容复制到新的数组，释放掉原来数组所占内存。

究其本质：数组在逻辑结构和物理结构上都是连续的，对数组局部进行的修改可能引起大范围甚至整个数据结构的调整。

改变存储策略

- 保留逻辑上的次序（通过使用指针），每个节点的物理地址不作要求
- 不再具有数组随机访问的特性，静态访问操作时间复杂度 $O(n)$
- 插入删除操作仅在局部进行，动态修改操作时间复杂度 $O(1)$ 。当然确定在哪个节点上操作仍需 $O(n)$ 的时间复杂度。
- 空间利用率更高，按需确定大小，可以动态扩容。但每个节点指针的存储需额外占用一定空间。

单向链表

节点定义

C++写法

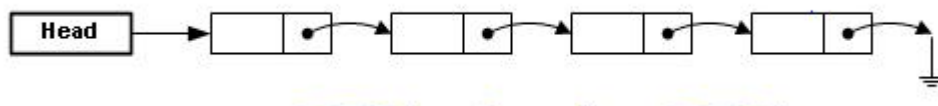
```
1  struct Node {    //C++写法
2      int data;
3      Node* next;
4      //构造函数
5      Node(int x = 0) { data = x; next = NULL; }
6  };
7
8  int main() {
9      Node* temp1 = new Node(1);
10     Node* temp2 = new Node(2);
11     temp1->next = temp2;
12 }
```

C语言写法

```
1  typedef struct Node{    //C语言写法
2      int data;
3      struct Node* next;
4  }Node;
5
6  int main() {
7      Node* temp1 = (Node*)malloc(sizeof(Node));
8      temp1->data = 1;
9      temp1->next = NULL;
10
11     Node* temp2 = (Node*)malloc(sizeof(Node));
12     temp2->data = 2;
13     temp2->next = NULL;
14
15     temp1->next = temp2;
16 }
```

关于头结点

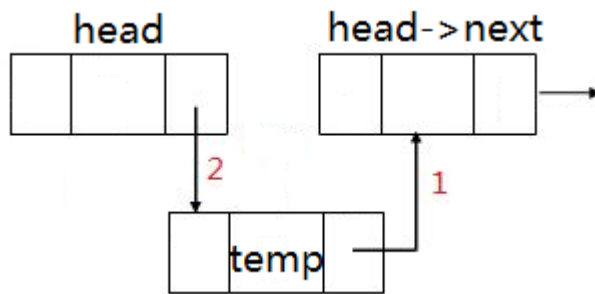
- 又称哨兵节点。将链表空与非空情况进行统一，使得各类算法无需对各种边界退化情况做专门处理。
- 除非题目特别说明，否则默认链表都有头结点。



链表定义

```
1  struct LinkList {
2      Node* head;    //头节点，必须
3      //Node* rear;    //尾指针，指向链表最后一个节点，可选
4      //int size;    //链表节点个数，可选
5
6      LinkList() {    //构造函数，链表创建时的初始化操作
7          head = new Node(); //创建头结点
8          //rear = head;    //尾指针指向头结点
9          //size = 0;
10     }
11 };
12
13 int main() {
14     LinkList* L = new LinkList();
15 }
```

使用头插法建立链表

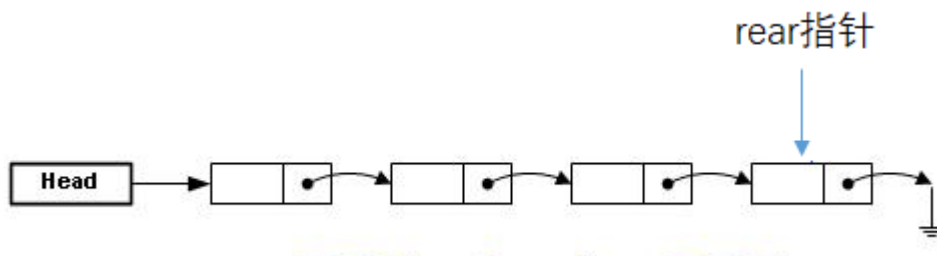


```

1 void initLinkedList_head(LinkedList* L, int A[], int n) { //头插法建立链表
2     for (int i = 0; i < n; i++) {
3         Node* temp = new Node(A[i]);
4         temp->next = L->head->next;
5         L->head->next = temp;
6     }
7 }

```

使用尾插法建立链表



```

1 void initLinkedList_rear(LinkedList* L, int A[], int n) { //尾插法建立链表
2     Node* rear = L->head; //创建尾指针
3     for (int i = 0; i < n; i++) {
4         Node* temp = new Node(A[i]);
5         rear->next = temp;
6         rear = rear->next; //更新尾指针
7     }
8 }

```

统计链表节点个数、输出整个链表

```

1 int getSize(LinkedList* L) {
2     int cnt = 0;
3     for (Node* p = L->head->next; p != NULL; p = p->next) {
4         cnt++;
5     }
6     return cnt;
7 }
8
9 void printLinkedList(LinkedList* L) {
10    for (Node* p = L->head->next; p != NULL; p = p->next) {
11        printf("%d ", p->data);
12    }

```

```

13 | printf("\n");
14 | }

```

查找

查找指定值节点位置

```

1 | Node* findElem(LinkList* L, int x) {
2 |     for (Node* p = L->head->next; p != NULL; p = p->next) {
3 |         if (p->data == x) {
4 |             return p;
5 |         }
6 |     }
7 |     return NULL;
8 | }

```

访问第k个节点

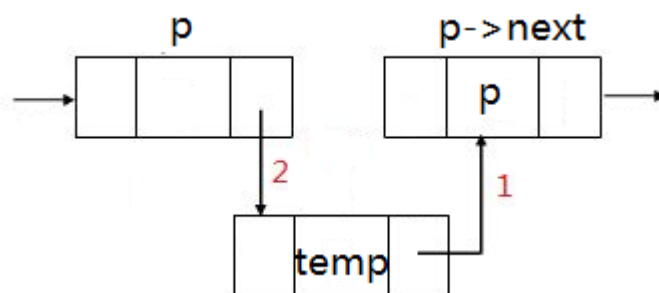
```

1 | int get(LinkList* L, int k) { //访问第k个节点 (k从0开始计数)
2 |     int cnt = 0;
3 |     for (Node* p = L->head->next; p != NULL; p = p->next) {
4 |         if (cnt == k) return p->data;
5 |         cnt++;
6 |     }
7 |     return -1; //越界了
8 | }

```

插入

在指定节点后面插入一个节点



```

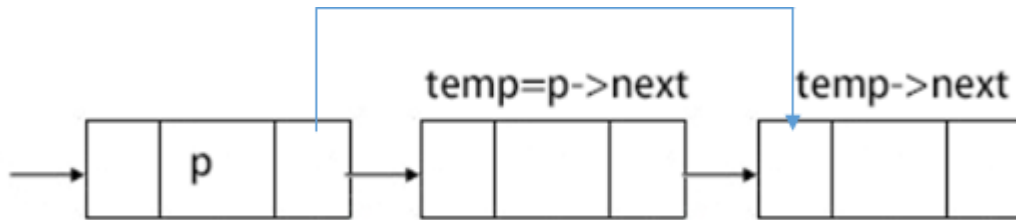
1 | Node* insertNodeAfter(Node* p, int x) { //在指定节点后面插入节点，并返回新插入节点的位置
2 |     Node* temp = new Node(x);
3 |     temp->next = p->next;
4 |     p->next = temp;
5 |     return temp;
6 | }

```

删除

要删除节点p，必须找到节点p的前驱节点。

删除节点p的后继节点



```
1 int deleteNodeAfter(Node* p) { //删除指定节点的后继节点，并返回删除节点的值
2     if (p->next == NULL) return -1;
3     Node* temp = p->next;
4     int ret = temp->data;
5     p->next = temp->next;
6     delete temp; //free(temp);
7     return ret;
8 }
```

删除指定位置节点

由于该单向链表没有尾节点，删除最后一个节点时需要专门处理

```
1 int deleteNode(LinkList* L, Node* t) {
2     Node* p;
3     for (p = L->head; p != NULL; p = p->next) { //找到节点t的前驱节点p
4         if (p->next == t) break;
5     }
6
7     //return deleteNodeAfter(p);
8     int ret = t->data;
9     p->next = t->next;
10    delete t; //free(t);
11    return ret;
12 }
```

完整代码

```
1 #include <stdio>
2
3 struct Node {
4     int data;
5     Node* next;
6
7     //构造函数
```

```

8     Node(int x = 0) { data = x; next = NULL; }
9 };
10
11 struct LinkList {
12     Node* head;      //头节点, 必须
13     //Node* rear;     //尾指针, 指向链表最后一个节点, 可选
14     //int size;       //链表节点个数, 可选
15
16     LinkList() {      //构造函数
17         head = new Node(); //创建头结点
18         //rear = head;    //尾指针指向头结点
19         //size = 0;
20     }
21 };
22
23 void initLinkList_head(LinkList* L, int A[], int n) { //头插法建立链表
24     for (int i = 0; i < n; i++) {
25         Node* temp = new Node(A[i]);
26         temp->next = L->head->next;
27         L->head->next = temp;
28     }
29 }
30
31 void initLinkList_rear(LinkList* L, int A[], int n) { //尾插法建立链表
32     Node* rear = L->head; //创建尾指针
33     for (int i = 0; i < n; i++) {
34         Node* temp = new Node(A[i]);
35         rear->next = temp;
36         rear = rear->next; //更新尾指针
37     }
38 }
39
40 Node* findElem(LinkList* L, int x) {
41     for (Node* p = L->head->next; p != NULL; p = p->next) {
42         if (p->data == x) {
43             return p;
44         }
45     }
46     return NULL;
47 }
48
49 Node* insertNodeAfter(Node* p, int x) { //在指定节点后面插入节点, 并返回新插入节点的位置
50     Node* temp = new Node(x);
51     temp->next = p->next;
52     p->next = temp;
53     return temp;
54 }
55
56 int deleteNodeAfter(Node* p) { //删除指定节点的后继节点, 并返回删除节点的值
57     if (p->next == NULL) return -1;
58     Node* temp = p->next;
59     int ret = temp->data;
60     p->next = temp->next;

```

```

61     delete temp;    //free(temp);
62     return ret;
63 }
64
65 int deleteNode(LinkList* L, Node* t) {
66     Node* p;
67     for (p = L->head; p != NULL; p = p->next) { //找到节点t的前驱节点p
68         if (p->next == t) break;
69     }
70     //return deleteNodeAfter(p);
71     int ret = t->data;
72     p->next = t->next;
73     delete t;    //free(t);
74     return ret;
75 }
76
77 int getLinkListSize(LinkList* L) {
78     int cnt = 0;
79     for (Node* p = L->head->next; p != NULL; p = p->next) {
80         cnt++;
81     }
82     return cnt;
83 }
84
85 void printLinkList(LinkList* L) {
86     for (Node* p = L->head->next; p != NULL; p = p->next) {
87         printf("%d ", p->data);
88     }
89     printf("\n");
90 }
91
92 int main() {
93     int A[] = { 1,2,3,4,5 };
94     int N = sizeof(A) / sizeof(int);
95
96     LinkList* L_1 = new LinkList();
97     initLinkList_head(L_1, A, N);
98     printLinkList(L_1);
99
100    LinkList* L_2 = new LinkList();
101    initLinkList_rear(L_2, A, N);
102    printLinkList(L_2);
103
104    Node* p = findElem(L_1, 3);
105    Node* p1 = insertNodeAfter(p, 10);
106    printLinkList(L_1);
107
108    deleteNode(L_1, p1->next);
109    printLinkList(L_1);
110    deleteNode(L_1, p1->next);
111    printLinkList(L_1);
112
113    return 0;

```

完整代码（头结点裸露、纯C语言实现）

写法1:

```
1 Node* head;    //头结点
```

写法2:

```
1 typedef Node* Linklist;
2 Linklist L;    //跟上面那句本质上其实一样
```

上面两种写法其实本质上没有任何区别，写法2只是给头结点的类型 `Node*` 取了个别名叫 `Linklist`。

以下是完整代码：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct Node {
5     int data;
6     struct Node* next;
7 } Node;
8
9 typedef Node* Linklist;
10
11 Node* head;    //头结点
12 Linklist linklist;    //跟上面那句本质上其实一样
13
14 Linklist initLinklist() {
15     Node* temp = (Node*)malloc(sizeof(Node));
16     temp->next = NULL;
17     return temp;
18 }
19
20 void initLinklist_head(Linklist L, int A[], int n) {    //头插法建立链表
21     //思考：这个函数是要对链表L进行修改的，既然如此为什么不加引用&？
22     //其实这里的L本质上是一个指向链表头结点的指针，通过该指针去修改头结点的内容完全没问题
23     //L指针存放的是链表头结点在内存中的存放地址，我们要修改的只是头结点的内容，
24     //而头结点在内存中的存放地址并没有修改，所以不需要修改指针L，即不需要加引用&
25     //void initLinklist_head(Node* headNodePosi, int A[], int n);
26     for (int i = 0; i < n; i++) {
27         Node* temp = (Node*)malloc(sizeof(Node));
28         temp->data = A[i];
29         temp->next = L->next;
30         L->next = temp;
31     }
32 }
33
```



```

34 void initLinklist_rear(Linklist L, int A[], int n) { //尾插法建立链表
35     Node* rear = L; //创建尾指针
36     for (int i = 0; i < n; i++) {
37         Node* temp = (Node*)malloc(sizeof(Node));
38         temp->data = A[i];
39         temp->next = NULL;
40         rear->next = temp;
41         rear = rear->next; //更新尾指针
42     }
43 }
44
45 Node* findElem(Linklist L, int x) {
46     for (Node* p = L->next; p != NULL; p = p->next) {
47         if (p->data == x) {
48             return p;
49         }
50     }
51     return NULL;
52 }
53
54 Node* insertNodeAfter(Node* p, int x) { //在指定节点后面插入节点，并返回新插入节点的位置
55     Node* temp = (Node*)malloc(sizeof(Node));
56     temp->data = x;
57     temp->next = p->next;
58     p->next = temp;
59     return temp;
60 }
61
62 int deleteNodeAfter(Node* p) { //删除指定节点的后继节点，并返回删除节点的值
63     if (p->next == NULL) return -1;
64     Node* temp = p->next;
65     int ret = temp->data;
66     p->next = temp->next;
67     free(temp);
68     return ret;
69 }
70
71 int deleteNode(Linklist L, Node* t) {
72     Node* p;
73     for (p = L; p != NULL; p = p->next) { //找到节点t的前驱节点p
74         if (p->next == t) break;
75     }
76     //return deleteNodeAfter(p);
77     int ret = t->data;
78     p->next = t->next;
79     free(t);
80     return ret;
81 }
82
83 int getLinklistSize(Linklist L) {
84     int cnt = 0;
85     for (Node* p = L->next; p != NULL; p = p->next) {
86         cnt++;

```

```

87     }
88     return cnt;
89 }
90
91 void printLinklist(Linklist L) {
92     for (Node* p = L->next; p != NULL; p = p->next) {
93         printf("%d ", p->data);
94     }
95     printf("\n");
96 }
97
98 int main() {
99     int A[] = { 1,2,3,4,5 };
100     int N = sizeof(A) / sizeof(int);
101
102     Linklist L_1 = initLinklist();
103     initLinklist_head(L_1, A, N);
104     printLinklist(L_1);
105
106     Node* head = initLinklist();
107     initLinklist_rear(head, A, N);
108     printLinklist(head);
109
110     Node* p = findElem(L_1, 3);
111     Node* p1 = insertNodeAfter(p, 10);
112     printLinklist(L_1);
113
114     deleteNode(L_1, p1->next);
115     printLinklist(L_1);
116     deleteNode(L_1, p1->next);
117     printLinklist(L_1);
118
119     return 0;
120 }

```

双向链表

节点、链表定义

```

1  struct Node {
2      int data;
3      Node *pre, *next;
4
5      Node(int x = 0) { data = x; pre = NULL; next = NULL; }
6  };
7
8  struct LinkList {
9      Node *head, *rear; //头节点、尾节点
10     //int size;
11

```

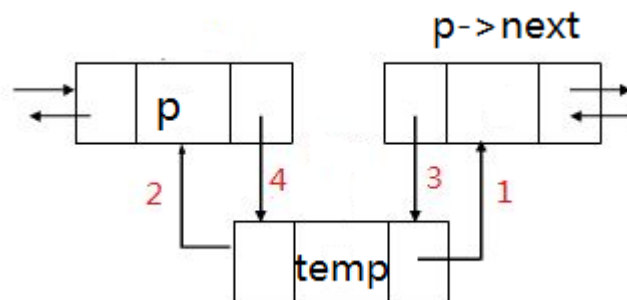
```

12 LinkList() {
13     head = new Node();
14     rear = new Node();
15     head->next = rear;
16     rear->pre = head;
17     //size = 0;
18 }
19 };

```

插入

在p节点后面插入节点

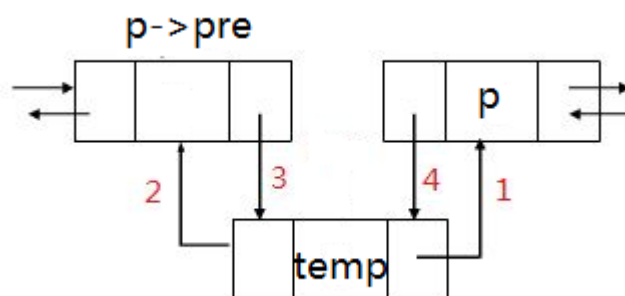


```

1 Node* insertNodeAfter(Node* p, int x) { //在p后面插入
2     Node* temp = new Node(x);
3     temp->next = p->next;
4     temp->pre = p;
5     p->next->pre = temp;
6     p->next = temp;
7     //size++;
8     return temp;
9 }

```

在p节点前面插入节点

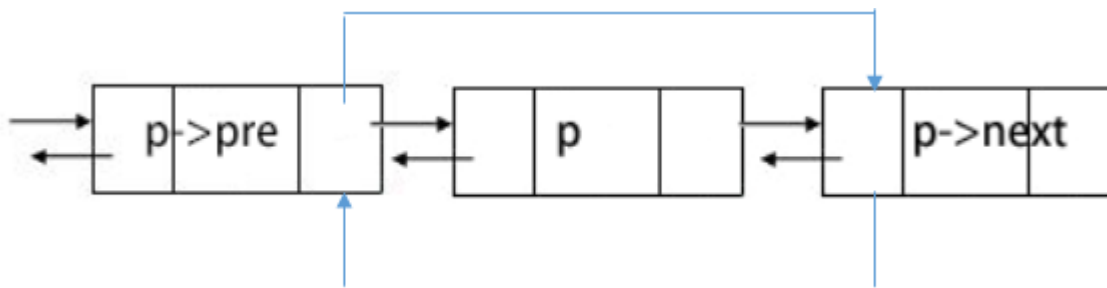


```

1 Node* insertNodeBefore(Node* p, int x) {    //在p前面插入
2     Node* temp = new Node(x);
3     temp->next = p;
4     temp->pre = p->pre;
5     p->pre->next = temp;
6     p->pre = temp;
7     //size++;
8     return temp;
9 }

```

删除



```

1 int deleteNode(Node* p) {
2     int ret = p->data;
3     p->pre->next = p->next;
4     p->next->pre = p->pre;
5     delete p;    //free(p);
6     //size--;
7     return ret;
8 }

```

清空

```

1 void clear(LinkList* L) {
2     for (Node* p = L->head->next->next; p != NULL; p = p->next) {
3         delete p->pre;
4     }
5     L->head->next = rear;
6     L->rear->pre = head;
7     //L->size = 0;
8 }

```

完整代码（使用C++面向对象的方法描述）

```

1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5     int data;
6     Node *pre, *next;
7 }

```

```

8     Node(int x = 0) { data = x; pre = NULL; next = NULL; }
9 };
10
11 class LinkList {
12 private:
13     Node *head, *rear; //头节点、尾节点 (注意区分尾节点与尾指针)
14     int size;
15
16 public:
17     LinkList() { //构造函数, 用于链表的初始化
18         head = new Node();
19         rear = new Node();
20         head->next = rear;
21         rear->pre = head;
22         size = 0;
23     }
24
25     ~LinkList() { //析构函数
26         clear();
27         delete head; delete rear;
28         printf("链表所占内存空间已释放\n");
29     }
30
31     int& get(int k) { //访问第k个节点 (k从0开始计数)
32         //这里函数返回的是引用, 因为不加引用get(k)只能作为右值 int a = get(k);
33         //而使用引用后get(k)则可以作为左值 get(k) = 100;
34         if (k >= size) throw exception("访问下标越界"); //越界了
35         int cnt = 0;
36         for (Node* p = head->next; p != rear; p = p->next) {
37             if (cnt == k) return p->data;
38             cnt++;
39         }
40     }
41
42     Node* findElem(int x) {
43         for (Node* p = head->next; p != rear; p = p->next) {
44             if (p->data == x) return p;
45         }
46         return NULL; //没找到
47     }
48
49     Node* insertNodeAfter(Node* p, int x) {
50         Node* temp = new Node(x);
51         temp->pre = p;
52         temp->next = p->next;
53         p->next->pre = temp;
54         p->next = temp;
55         size++;
56         return temp;
57     }
58
59     Node* insertNodeBefore(Node* p, int x) {
60         Node* temp = new Node(x);

```

```

61     temp->next = p;
62     temp->pre = p->pre;
63     p->pre->next = temp;
64     p->pre = temp;
65     size++;
66     return temp;
67 }
68
69 int deleteNode(Node* p) {
70     int ret = p->data;
71     p->pre->next = p->next;
72     p->next->pre = p->pre;
73     delete p;
74     size--;
75     return ret;
76 }
77
78 void printAll() {
79     for (Node* p = head->next; p != rear; p = p->next) {
80         printf("%d ", p->data);
81     }
82     printf("\n");
83 }
84
85 bool empty() {
86     return size == 0;
87 }
88
89 void clear() {
90     for (Node* p = head->next->next; p != NULL; p = p->next) {
91         delete p->pre;
92     }
93     head->next = rear;
94     rear->pre = head;
95     size = 0;
96 }
97
98 int& operator[] (int k) {    //重载下标[]访问
99     return get(k);
100 }
101
102 //下面这6个操作是为下一章学习队列做准备的
103 int& front() {    //访问第一个节点
104     if (empty()) throw exception("链表为空");
105     return head->next->data;
106 }
107
108 int& back() {    //访问最后一个节点
109     if (empty()) throw exception("链表为空");
110     return rear->pre->data;
111 }
112
113 Node* push_back(int x) {    //在链表末尾插入一个节点

```

```

114         return insertNodeBefore(rear, x);
115     }
116
117     Node* push_front(int x) {    //在链表开始插入一个节点
118         return insertNodeAfter(head, x);
119     }
120
121     int pop_back() {            //删除链表末尾的节点
122         if (empty()) throw exception("链表为空");
123         return deleteNode(rear->pre);
124     }
125
126     int pop_front() {          //删除链表开始的节点
127         if (empty()) throw exception("链表为空");
128         return deleteNode(head->next);
129     }
130 };
131
132 int main() {
133     LinkList L;
134     for (int i = 1; i <= 5; i++) {
135         L.push_back(i);
136     }
137     printf("L[2] = %d\n", L[2]);
138     L[4] = 100;
139     L.printAll();
140
141     Node* p = L.findElem(4);
142     L.insertNodeBefore(p, 10);
143     L.printAll();
144
145     L.deleteNode(p);
146     L.printAll();
147
148     return 0;
149 }

```

静态链表


```

22     NodePosi createNode(int x = 0) {
23         if (index_allocator >= MAX_SIZE) throw exception("静态链表已满");
24         nodes[index_allocator].data = x;
25         nodes[index_allocator].next = -1;    // 相当于节点temp.next = NULL
26         return index_allocator++;
27     }
28
29 public:
30     LinkList() {
31         index_allocator = 0;
32         nodes = new Node[MAX_SIZE];
33         head = createNode();    // 数组0号下标作为头结点
34         printf("静态链表初始化完成\n");
35     }
36
37     ~LinkList() {
38         delete[] nodes;
39     }
40
41     NodePosi insertNodeAfter(NodePosi p, int x) {    // 在指定节点p后面插入节点，并返回新插入节
点的位
置
42         NodePosi temp = createNode(x);
43         nodes[temp].next = nodes[p].next;
44         nodes[p].next = temp;
45         return temp;
46     }
47
48     int deleteNodeAfter(NodePosi p) {    // 删除指定节点p的后继节点，并返回删除节点的值
49         if (nodes[p].next == -1) throw exception("不能删除链表最后一个节点的后继节点");
50         NodePosi temp = nodes[p].next;
51         nodes[p].next = nodes[temp].next;
52         return nodes[temp].data;
53     }
54
55     bool empty() {
56         return nodes[head].next == -1;
57     }
58
59     void print() {
60         for (NodePosi p = nodes[head].next; p != -1; p = nodes[p].next) {
61             printf("%d ", nodes[p].data);
62         }
63         printf("\n");
64     }
65 };
66
67 int main() {
68     LinkList L;
69     printf("L.empty() = %s\n", L.empty() ? "true" : "false");
70
71     for (int i = 1; i <= 5; i++) {
72         L.insertNodeAfter(0, i);    // 在0号节点（头结点）后插入一个节点，相当于头插法建立链表
73     }

```

```

74     L.print();
75     printf("L.empty() = %s\n", L.empty() ? "true" : "false");
76
77     for (int i = 0; i < 5; i++) {
78         L.deleteNodeAfter(0);
79         L.print();
80     }
81
82     printf("L.empty() = %s\n", L.empty() ? "true" : "false");
83
84     return 0;
85 }

```

考试题目

经验：对于单向链表来说，`for (Node* p = head->next; p != NULL; p = p->next)` 遍历链表只适用于静态访问操作，对于遍历链表的同时进行插入删除等动态操作，应想清楚代码逻辑，改用 `while` 循环。

删除链表最小值节点

假设链表各节点的值不重复

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  struct Node {
6      int data;
7      Node* next;
8
9      Node(int x = 0) : data(x), next(NULL) {}
10 };
11
12 void deleteMinNode(Node* head) {
13     Node* curNode_pre = head;
14     Node* minNode_pre = NULL;
15     int minNode_value = 1e9;
16
17     for (Node* p = head->next; p != NULL; p = p->next) {
18         if (p->data < minNode_value) {
19             minNode_value = p->data;
20             minNode_pre = curNode_pre;
21         }
22         curNode_pre = p;
23     }
24
25     Node* temp = minNode_pre->next;
26     minNode_pre->next = temp->next;
27     delete temp;
28 }
29
30 void printLinkList(Node* head) {

```

```

31     for (Node* p = head->next; p != NULL; p = p->next) {
32         printf("%d ", p->data);
33     }
34     printf("\n");
35 }
36
37 int main() {
38     vector<int> num = { 1,2,3,-2 };
39     Node* L = new Node();
40     Node* rear = L;
41     for (int x : num) {
42         rear->next = new Node(x);
43         rear = rear->next;
44     }
45     printLinkList(L);
46
47     deleteMinNode(L);
48     printLinkList(L);
49
50     return 0;
51 }

```

链表逆置

设计一个算法将带头结点的单链表L逆置，要求不能建立新的节点，只能使用表中已有的节点重新组合完成

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  struct Node {
6      int data;
7      Node* next;
8
9      Node(int x = 0) :data(x), next(NULL) {}
10 };
11
12 void reverseLinkList(Node* head) {
13     Node* p = head->next;
14     head->next = NULL;
15     while (p != NULL) {
16         Node* temp = p->next;
17         p->next = head->next;
18         head->next = p;
19         p = temp;
20     }
21 }
22
23 void printLinkList(Node* head) {
24     for (Node* p = head->next; p != NULL; p = p->next) {
25         printf("%d ", p->data);

```

```

26     }
27     printf("\n");
28 }
29
30 int main() {
31     vector<int> num = { 1,2,3,4,5,6 };
32     Node* A = new Node();
33     Node* rear_A = A;
34     for (int x : num) {
35         rear_A->next = new Node(x);
36         rear_A = rear_A->next;
37     }
38     printLinkList(A);
39
40     reverseLinkList(A);
41
42     printLinkList(A);
43
44     return 0;
45 }

```

有序链表去重

```

1 void uniqueLinkList(Node* head) {
2     //错误写法, 绝对不能这么写
3     for (Node* p = head->next; p->next != NULL; p = p->next) {
4         if (p->next->data == p->data) {
5             Node* temp = p->next;
6             p->next = temp->next;
7             delete temp;
8         }
9     }
10 }

```

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 struct Node {
6     int data;
7     Node* next;
8
9     Node(int x = 0) : data(x), next(NULL) {}
10 };
11
12 void uniqueLinkList_1(Node* head) {
13     Node* p = head->next;
14     if (p == NULL) return; //链表为空
15     while (p->next != NULL) {
16         if (p->next->data == p->data) {

```

```

17         Node* temp = p->next;
18         p->next = temp->next;
19         delete temp;
20     } else {
21         p = p->next;
22     }
23 }
24 }
25
26 void printLinkList(Node* head) {
27     for (Node* p = head->next; p != NULL; p = p->next) {
28         printf("%d ", p->data);
29     }
30     printf("\n");
31 }
32
33 int main() {
34     vector<int> num = { 1,1,1,2,3,3,3,5,5 };
35     Node* L = new Node();
36     Node* rear = L;
37     for (int x : num) {
38         rear->next = new Node(x);
39         rear = rear->next;
40     }
41     printLinkList(L);
42
43     uniqueLinkList_1(L);
44     printLinkList(L);
45
46     return 0;
47 }

```

拆分链表

设计一个算法，将一个带头节点的单链表A（数据域为整数）拆分为两个单链表A、B，其中链表A只含原链表data为奇数的节点，链表B只含原链表data为偶数的节点，且均需要保留原链表节点的相对次序。算法应接收两个链表A、B，其中链表B并未初始化。

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  struct Node {
6      int data;
7      Node* next;
8
9      Node(int x = 0) :data(x), next(NULL) {}
10 };
11
12 void splitLinkList(Node* A, Node* &B) {
13     B = new Node();

```

```

14     Node* rear = B;
15     Node* p = A;
16     while ( p->next != NULL) {
17         if (p->next->data % 2 == 0) {
18             Node* temp = p->next;
19             p->next = temp->next;
20             rear->next = temp;
21             temp->next = NULL;
22             rear = temp;    //rear = rear->next;
23         } else {
24             p = p->next;
25         }
26     }
27 }
28
29 void printLinkList(Node* head) {
30     for (Node* p = head->next; p != NULL; p = p->next) {
31         printf("%d ", p->data);
32     }
33     printf("\n");
34 }
35
36 int main() {
37     vector<int> num = { 2,2,1,3,4,4,4,5,6 };
38     Node* A = new Node();
39     Node* rear_A = A;
40     for (int x : num) {
41         rear_A->next = new Node(x);
42         rear_A = rear_A->next;
43     }
44     printLinkList(A);
45     Node* B;
46
47     splitLinkList(A, B);
48     printLinkList(A);
49     printLinkList(B);
50
51     return 0;
52 }

```

合并两个有序单链表

链表A、B是两个带头结点的递增有序单链表，设计一个算法，将链表A、B合并成一个有序的单链表C，其中链表C的节点只能由链表A、B的节点组成，不能创建新的节点。合并结束应销毁链表A、B。

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  struct Node {
6      int data;
7      Node* next;

```

```

8
9     Node(int x = 0) :data(x), next(NULL) {}
10 };
11
12 Node* mergeLinkList(Node* &A, Node* &B) {
13     Node* p = A->next;
14     Node* q = B->next;
15
16     Node* C = A;    //将链表A的头结点给链表C使用
17     C->next = NULL;
18
19     A = NULL;
20     delete B;    //链表B的头结点不需要了，释放掉
21     B = NULL;
22
23     Node* rear = C; //对链表C进行尾插法
24     while (p != NULL && q != NULL) {
25         if (p->data < q->data) {
26             rear->next = p;
27             p = p->next;
28         } else {
29             rear->next = q;
30             q = q->next;
31         }
32         rear = rear->next;
33     }
34     //rear->next = NULL;
35
36     if (p != NULL) {
37         rear->next = p;
38     }
39     if (q != NULL) {
40         rear->next = q;
41     }
42
43     return C;
44 }
45
46
47 void printLinkList(Node* head) {
48     for (Node* p = head->next; p != NULL; p = p->next) {
49         printf("%d ", p->data);
50     }
51     printf("\n");
52 }
53
54 int main() {
55     vector<int> num1 = { 1,3,5,7 };
56     vector<int> num2 = { 2,4,6,20,30 };
57
58     Node* A = new Node();
59     Node* rear_A = A;
60     for (int x : num1) {

```

```
61     rear_A->next = new Node(x);
62     rear_A = rear_A->next;
63 }
64
65 Node* B = new Node();
66 Node* rear_B = B;
67 for (int x : num2) {
68     rear_B->next = new Node(x);
69     rear_B = rear_B->next;
70 }
71
72 Node* C = mergeLinkList(A, B);
73
74 printLinkList(C);
75 printf("%p\n", A); //输出指针A、B的地址
76 printf("%p\n", B);
77
78 return 0;
79 }
```