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

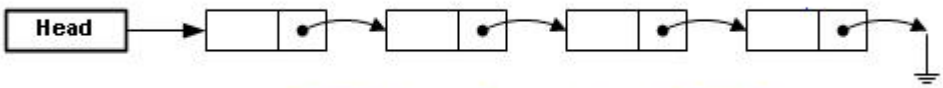
```cpp
struct Node {   //C++写法
    int data;
    Node* next;
    //构造函数
    Node(int x = 0) { data = x; next = NULL; }
};

int main() {
    Node* temp1 = new Node(1);
    Node* temp2 = new Node(2);
    temp1->next = temp2;
}
```

C语言写法

```c
typedef struct Node{    //C语言写法
    int data;
    struct Node* next;
}Node;

int main() {
    Node* temp1 = (Node*)malloc(sizeof(Node));
    temp1->data = 1;
    temp1->next = NULL;

    Node* temp2 = (Node*)malloc(sizeof(Node));
    temp2->data = 2;
    temp2->next = NULL;

    temp1->next = temp2;
}
```

关于头结点

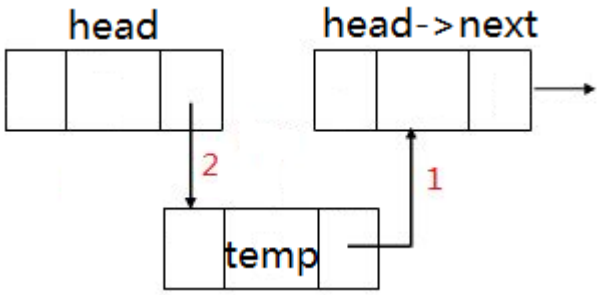
- 又称哨兵节点。将链表**空**与**非空**情况进行统一，使得各类算法无需对各种边界退化情况做专门处理。
- 除非题目特别说明，否则默认链表都有头结点。



链表定义

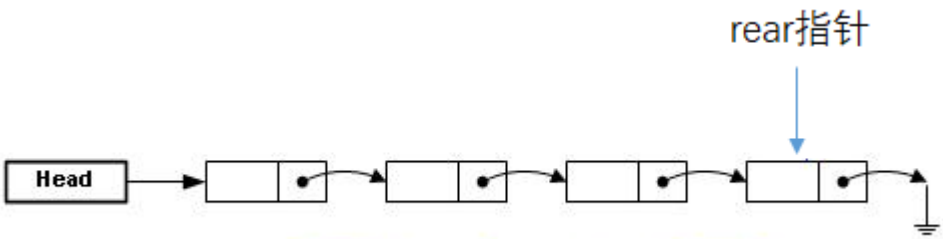
```c
struct LinkList {
    Node* head;     //头节点，必须
    //Node* rear;       //尾指针，指向链表最后一个节点，可选
    //int size;     //链表节点个数，可选

    LinkList() {    //构造函数，链表创建时的初始化操作
        head = new Node();  //创建头结点
        //rear = head;      //尾指针指向头结点
        //size = 0;
    }
};

int main() {
    LinkList* L = new LinkList();
}
```

使用头插法建立链表




```
void initLinkList_head(LinkList* L, int A[], int n) {   //头插法建立链表
    for (int i = 0; i < n; i++) {
        Node* temp = new Node(A[i]);
        temp->next = L->head->next;
        L->head->next = temp;
    }
}
```

使用尾插法建立链表




```
void initLinkList_rear(LinkList* L, int A[], int n) {   //尾插法建立链表
    Node* rear = L->head;       //创建尾指针
    for (int i = 0; i < n; i++) {
        Node* temp = new Node(A[i]);
        rear->next = temp;
        rear = rear->next;      //更新尾指针
    }
}
```

统计链表节点个数、输出整个链表

```
int getSize(LinkList* L) {
    int cnt = 0;
    for (Node* p = L->head->next; p != NULL; p = p->next) {
        cnt++;
    }
    return cnt;
}

void printLinkList(LinkList* L) {
    for (Node* p = L->head->next; p != NULL; p = p->next) {
        printf("%d ", p->data);
    }
```

```
    printf("\n");
}
```

查找

查找指定值节点位置

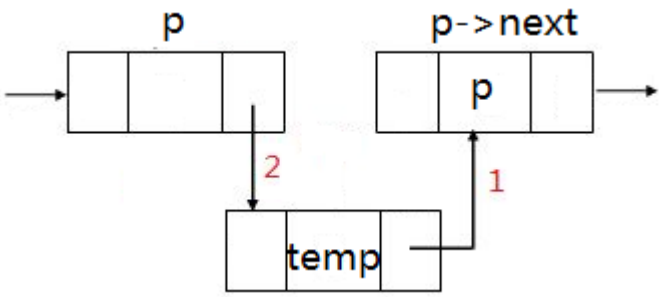
```
Node* findElem(LinkList* L, int x) {
    for (Node* p = L->head->next; p != NULL; p = p->next) {
        if (p->data == x) {
            return p;
        }
    }
    return NULL;
}
```

访问第k个节点

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

插入

在指定节点后面插入一个节点

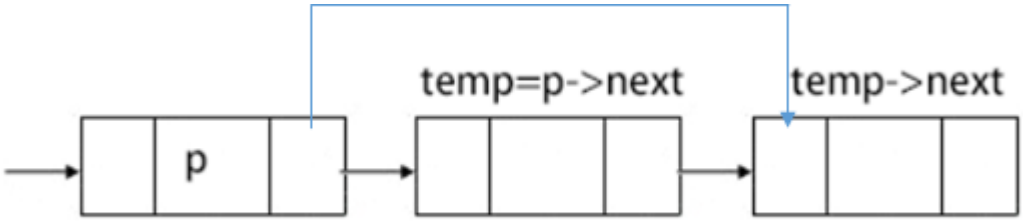



```
Node* insertNodeAfter(Node* p, int x) { //在指定节点后面插入节点，并返回新插入节点的位置
    Node* temp = new Node(x);
    temp->next = p->next;
    p->next = temp;
    return temp;
}
```

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




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

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

```
int deleteNode(LinkList* L, Node* t) {
    Node* p;
    for (p = L->head; p != NULL; p = p->next) { //找到节点t的前驱节点p
        if (p->next == t) break;
    }

    //return deleteNodeAfter(p);
    int ret = t->data;
    p->next = t->next;
    delete t;   //free(t);
    return ret;
}
```

完整代码

```
#include <cstdio>

struct Node {
    int data;
    Node* next;

    //构造函数
```

```
    Node(int x = 0) { data = x; next = NULL; }
};

struct LinkList {
    Node* head;     //头节点，必须
    //Node* rear;       //尾指针，指向链表最后一个节点，可选
    //int size;     //链表节点个数，可选

    LinkList() {    //构造函数
        head = new Node();  //创建头结点
        //rear = head;      //尾指针指向头结点
        //size = 0;
    }
};

void initLinkList_head(LinkList* L, int A[], int n) {   //头插法建立链表
    for (int i = 0; i < n; i++) {
        Node* temp = new Node(A[i]);
        temp->next = L->head->next;
        L->head->next = temp;
    }
}

void initLinkList_rear(LinkList* L, int A[], int n) {   //尾插法建立链表
    Node* rear = L->head;       //创建尾指针
    for (int i = 0; i < n; i++) {
        Node* temp = new Node(A[i]);
        rear->next = temp;
        rear = rear->next;      //更新尾指针
    }
}

Node* findElem(LinkList* L, int x) {
    for (Node* p = L->head->next; p != NULL; p = p->next) {
        if (p->data == x) {
            return p;
        }
    }
    return NULL;
}

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
    Node* temp = new Node(x);
    temp->next = p->next;
    p->next = temp;
    return temp;
}

int deleteNodeAfter(Node* p) {  //删除指定节点的后继节点，并返回删除节点的值
    if (p->next == NULL) return -1;
    Node* temp = p->next;
    int ret = temp->data;
    p->next = temp->next;
```

```
    delete temp;    //free(temp);
    return ret;
}

int deleteNode(LinkList* L, Node* t) {
    Node* p;
    for (p = L->head; p != NULL; p = p->next) { //找到节点t的前驱节点p
        if (p->next == t) break;
    }
    //return deleteNodeAfter(p);
    int ret = t->data;
    p->next = t->next;
    delete t;   //free(t);
    return ret;
}

int getLinkListSize(LinkList* L) {
    int cnt = 0;
    for (Node* p = L->head->next; p != NULL; p = p->next) {
        cnt++;
    }
    return cnt;
}

void printLinkList(LinkList* L) {
    for (Node* p = L->head->next; p != NULL; p = p->next) {
        printf("%d ", p->data);
    }
    printf("\n");
}

int main() {
    int A[] = { 1,2,3,4,5 };
    int N = sizeof(A) / sizeof(int);

    LinkList* L_1 = new LinkList();
    initLinkList_head(L_1, A, N);
    printLinkList(L_1);

    LinkList* L_2 = new LinkList();
    initLinkList_rear(L_2, A, N);
    printLinkList(L_2);

    Node* p = findElem(L_1, 3);
    Node* p1 = insertNodeAfter(p, 10);
    printLinkList(L_1);

    deleteNode(L_1, p1->next);
    printLinkList(L_1);
    deleteNode(L_1, p1->next);
    printLinkList(L_1);

    return 0;
```

```
}
```

完整代码（头结点裸露、纯C语言实现）

写法1：

```
Node* head;     //头结点
```

写法2：

```
typedef Node* Linklist;
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：

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef Node* Linklist;

Node* head;     //头结点
Linklist linklist;      //跟上面那句本质上其实一样

Linklist initLinklist() {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->next = NULL;
    return temp;
}

void initLinklist_head(Linklist L, int A[], int n) {    //头插法建立链表
    //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& ？
    //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
    //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，
    //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&
    //void initLinklist_head(Node* headNodePosi, int A[], int n);
    for (int i = 0; i < n; i++) {
        Node* temp = (Node*)malloc(sizeof(Node));
        temp->data = A[i];
        temp->next = L->next;
        L->next = temp;
    }
}

```

```
void initLinklist_rear(Linklist L, int A[], int n) {    //尾插法建立链表
    Node* rear = L;     //创建尾指针
    for (int i = 0; i < n; i++) {
        Node* temp = (Node*)malloc(sizeof(Node));
        temp->data = A[i];
        temp->next = NULL;
        rear->next = temp;
        rear = rear->next;      //更新尾指针
    }
}

Node* findElem(Linklist L, int x) {
    for (Node* p = L->next; p != NULL; p = p->next) {
        if (p->data == x) {
            return p;
        }
    }
    return NULL;
}

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
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = x;
    temp->next = p->next;
    p->next = temp;
    return temp;
}

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

int deleteNode(Linklist L, Node* t) {
    Node* p;
    for (p = L; p != NULL; p = p->next) {   //找到节点t的前驱节点p
        if (p->next == t) break;
    }
    //return deleteNodeAfter(p);
    int ret = t->data;
    p->next = t->next;
    free(t);
    return ret;
}

int getLinklistSize(Linklist L) {
    int cnt = 0;
    for (Node* p = L->next; p != NULL; p = p->next) {
        cnt++;
```

```
    }
    return cnt;
}

void printLinklist(Linklist L) {
    for (Node* p = L->next; p != NULL; p = p->next) {
        printf("%d ", p->data);
    }
    printf("\n");
}

int main() {
    int A[] = { 1,2,3,4,5 };
    int N = sizeof(A) / sizeof(int);

    Linklist L_1 = initLinklist();
    initLinklist_head(L_1, A, N);
    printLinklist(L_1);

    Node* head = initLinklist();
    initLinklist_rear(head, A, N);
    printLinklist(head);

    Node* p = findElem(L_1, 3);
    Node* p1 = insertNodeAfter(p, 10);
    printLinklist(L_1);

    deleteNode(L_1, p1->next);
    printLinklist(L_1);
    deleteNode(L_1, p1->next);
    printLinklist(L_1);

    return 0;
}
```

双向链表

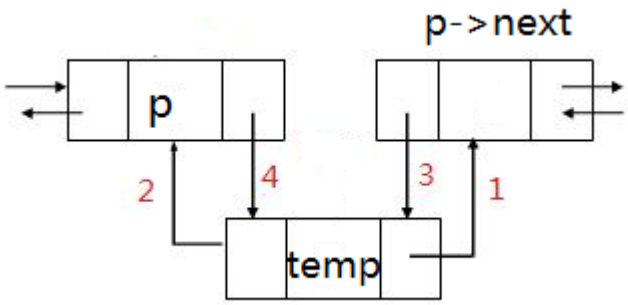
节点、链表定义

```
struct Node {
    int data;
    Node *pre, *next;

    Node(int x = 0) { data = x; pre = NULL; next = NULL; }
};

struct LinkList {
    Node *head, *rear;  //头节点、尾节点
    //int size;

```

```
    LinkList() {
        head = new Node();
        rear = new Node();
        head->next = rear;
        rear->pre = head;
        //size = 0;
    }
};
```

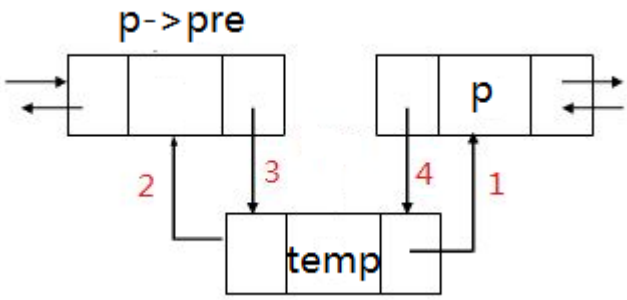
插入

在p节点后面插入节点



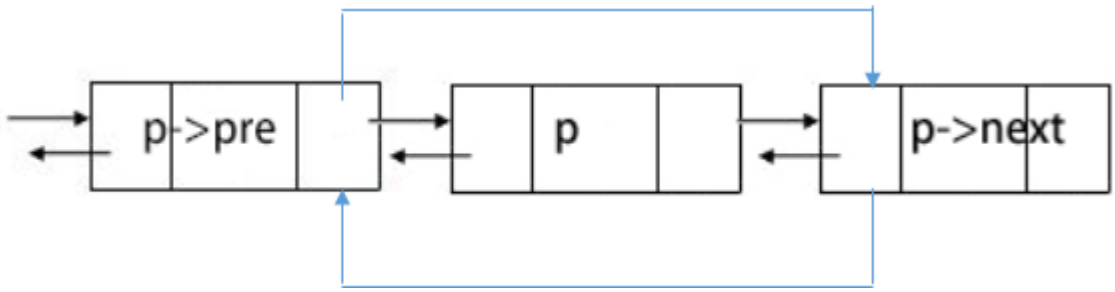

```
Node* insertNodeAfter(Node* p, int x) {      //在p后面插入
    Node* temp = new Node(x);
    temp->next = p->next;
    temp->pre = p;
    p->next->pre = temp;
    p->next = temp;
    //size++;
    return temp;
}
```

在p节点前面插入节点



```
Node* insertNodeBefore(Node* p, int x) {    //在p前面插入
    Node* temp = new Node(x);
    temp->next = p;
    temp->pre = p->pre;
    p->pre->next = temp;
    p->pre = temp;
    //size++;
    return temp;
}
```

删除




```
int deleteNode(Node* p) {
    int ret = p->data;
    p->pre->next = p->next;
    p->next->pre = p->pre;
    delete p;   //free(p);
    //size--;
    return ret;
}
```

清空

```
void clear(LinkList* L) {
    for (Node* p = L->head->next->next; p != NULL; p = p->next) {
        delete p->pre;
    }
    L->head->next = rear;
    L->rear->pre = head;
    //L->size = 0;
}
```

完整代码（使用C++面向对象的方法描述）

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node *pre, *next;

```

```
    Node(int x = 0) { data = x; pre = NULL; next = NULL; }
};

class LinkList {
private:
    Node *head, *rear;  //头节点、尾节点（注意区分尾节点与尾指针）
    int size;

public:
    LinkList() {    //构造函数，用于链表的初始化
        head = new Node();
        rear = new Node();
        head->next = rear;
        rear->pre = head;
        size = 0;
    }

    ~LinkList() {   //析构函数
        clear();
        delete head; delete rear;
        printf("链表所占内存空间已释放\n");
    }

    int& get(int k) {   //访问第k个节点（k从0开始计数）
        //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);
        //而使用引用后get(k)则可以作为左值  get(k) = 100;
        if (k >= size) throw exception("访问下标越界");   //越界了
        int cnt = 0;
        for (Node* p = head->next; p != rear; p = p->next) {
            if (cnt == k) return p->data;
            cnt++;
        }
    }

    Node* findElem(int x) {
        for (Node* p = head->next; p != rear; p = p->next) {
            if (p->data == x) return p;
        }
        return NULL;    //没找到
    }

    Node* insertNodeAfter(Node* p, int x) {
        Node* temp = new Node(x);
        temp->pre = p;
        temp->next = p->next;
        p->next->pre = temp;
        p->next = temp;
        size++;
        return temp;
    }

    Node* insertNodeBefore(Node* p, int x) {
        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

数组下标	struct Node	
	data	next
0	20	4
1	0	6
2	3	1
3		
4	5	2
5		
6	8	`-1(NULL)
7		
8		
20->5->3->0->8->NULL		

节点定义

```
typedef int NodePosi;   // 空指针NULL用-1表示，因此这里不能使用无符号int

struct Node {
    int data;
    NodePosi next;  //模拟指针
};
```

静态链表与普通链表的联系

其实整个内存也可以看成是一个数组，比如4GB的内存，每个字节分配一个数组编号的话，这个“数组”的下标范围就是从 0 ~ 2^32 - 1。这样普通链表也可以看成是静态链表，指针就相当于是内存这个数组的下标。

完整代码*

```
#include <iostream>
using namespace std;

typedef int NodePosi;   // 约定空指针NULL用-1表示，因此这里不能使用无符号int

struct Node {
    int data;
    NodePosi next;  //模拟指针
};

class LinkList {    //使用数组模拟的静态单向链表
private:
    const int MAX_SIZE = 1000;

    Node* nodes;

    NodePosi head;  //头结点，对于静态链表，头结点的数组下标一定是0

    NodePosi index_allocator;    //记录当创建一个新节点时，当前可分配的数组下标。
                                 //这算是一个非常笨的“内存池”，只会分配下标，不会回收下标。

```

```
    NodePosi createNode(int x = 0) {
        if (index_allocator >= MAX_SIZE) throw exception("静态链表已满");
        nodes[index_allocator].data = x;
        nodes[index_allocator].next = -1;   // 相当于节点temp.next = NULL
        return index_allocator++;
    }

public:
    LinkList() {
        index_allocator = 0;
        nodes = new Node[MAX_SIZE];
        head = createNode();    //数组0号下标作为头结点
        printf("静态链表初始化完成\n");
    }

    ~LinkList() {
        delete[] nodes;
    }

    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
        NodePosi temp = createNode(x);
        nodes[temp].next = nodes[p].next;
        nodes[p].next = temp;
        return temp;
    }

    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
        if (nodes[p].next == -1) throw exception("不能删除链表最后一个节点的后继节点");
        NodePosi temp = nodes[p].next;
        nodes[p].next = nodes[temp].next;
        return nodes[temp].data;
    }

    bool empty() {
        return nodes[head].next == -1;
    }

    void print() {
        for (NodePosi p = nodes[head].next; p != -1; p = nodes[p].next) {
            printf("%d ", nodes[p].data);
        }
        printf("\n");
    }
};

int main() {
    LinkList L;
    printf("L.empty() = %s\n", L.empty() ? "true" : "false");

    for (int i = 1; i <= 5; i++) {
        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
    }
```

```
    L.print();
    printf("L.empty() = %s\n", L.empty() ? "true" : "false");

    for (int i = 0; i < 5; i++) {
        L.deleteNodeAfter(0);
        L.print();
    }

    printf("L.empty() = %s\n", L.empty() ? "true" : "false");

    return 0;
}
```

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

```
#include <iostream>
#include <vector>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int x = 0) :data(x), next(NULL) {}
};

void deleteMinNode(Node* head) {
    Node* curNode_pre = head;
    Node* minNode_pre = NULL;
    int minNode_value = 1e9;

    for (Node* p = head->next; p != NULL; p = p->next) {
        if (p->data < minNode_value) {
            minNode_value = p->data;
            minNode_pre = curNode_pre;
        }
        curNode_pre = p;
    }

    Node* temp = minNode_pre->next;
    minNode_pre->next = temp->next;
    delete temp;
}

void printLinkList(Node* head) {
```

```
    for (Node* p = head->next; p != NULL; p = p->next) {
        printf("%d ", p->data);
    }
    printf("\n");
}

int main() {
    vector<int> num = { 1,2,3,-2 };
    Node* L = new Node();
    Node* rear = L;
    for (int x : num) {
        rear->next = new Node(x);
        rear = rear->next;
    }
    printLinkList(L);

    deleteMinNode(L);
    printLinkList(L);

    return 0;
}
```

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

```
#include <iostream>
#include <vector>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int x = 0) :data(x), next(NULL) {}
};

void reverseLinkList(Node* head) {
    Node* p = head->next;
    head->next = NULL;
    while (p != NULL) {
        Node* temp = p->next;
        p->next = head->next;
        head->next = p;
        p = temp;
    }
}

void printLinkList(Node* head) {
    for (Node* p = head->next; p != NULL; p = p->next) {
        printf("%d ", p->data);
```

```
    }
    printf("\n");
}

int main() {
    vector<int> num = { 1,2,3,4,5,6 };
    Node* A = new Node();
    Node* rear_A = A;
    for (int x : num) {
        rear_A->next = new Node(x);
        rear_A = rear_A->next;
    }
    printLinkList(A);

    reverseLinkList(A);

    printLinkList(A);

    return 0;
}
```

有序链表去重

```
void uniqueLinkList(Node* head) {
    //错误写法，绝对不能这么写
    for (Node* p = head->next; p->next != NULL; p = p->next) {
        if (p->next->data == p->data) {
            Node* temp = p->next;
            p->next = temp->next;
            delete temp;
        }
    }
}
```

```
#include <iostream>
#include <vector>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int x = 0) :data(x), next(NULL) {}
};

void uniqueLinkList_1(Node* head) {
    Node* p = head->next;
    if (p == NULL) return;  //链表为空
    while (p->next != NULL) {
        if (p->next->data == p->data) {
```

```
            Node* temp = p->next;
            p->next = temp->next;
            delete temp;
        } else {
            p = p->next;
        }
    }
}

void printLinkList(Node* head) {
    for (Node* p = head->next; p != NULL; p = p->next) {
        printf("%d ", p->data);
    }
    printf("\n");
}

int main() {
    vector<int> num = { 1,1,1,2,3,3,3,5,5 };
    Node* L = new Node();
    Node* rear = L;
    for (int x : num) {
        rear->next = new Node(x);
        rear = rear->next;
    }
    printLinkList(L);

    uniqueLinkList_1(L);
    printLinkList(L);

    return 0;
}
```

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。

```
#include <iostream>
#include <vector>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int x = 0) :data(x), next(NULL) {}
};

void splitLinkList(Node* A, Node* &B) {
    B = new Node();
```

```
    Node* rear = B;
    Node* p = A;
    while ( p->next != NULL) {
        if (p->next->data % 2 == 0) {
            Node* temp = p->next;
            p->next = temp->next;
            rear->next = temp;
            temp->next = NULL;
            rear = temp;    //rear = rear->next;
        } else {
            p = p->next;
        }
    }
}

void printLinkList(Node* head) {
    for (Node* p = head->next; p != NULL; p = p->next) {
        printf("%d ", p->data);
    }
    printf("\n");
}

int main() {
    vector<int> num = { 2,2,1,3,4,4,4,5,6 };
    Node* A = new Node();
    Node* rear_A = A;
    for (int x : num) {
        rear_A->next = new Node(x);
        rear_A = rear_A->next;
    }
    printLinkList(A);
    Node* B;

    splitLinkList(A, B);
    printLinkList(A);
    printLinkList(B);

    return 0;
}
```

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。

```
#include <iostream>
#include <vector>
using namespace std;

struct Node {
    int data;
    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) {
```

```
        rear_A->next = new Node(x);
        rear_A = rear_A->next;
    }

    Node* B = new Node();
    Node* rear_B = B;
    for (int x : num2) {
        rear_B->next = new Node(x);
        rear_B = rear_B->next;
    }

    Node* C = mergeLinkList(A, B);

    printLinkList(C);
    printf("%p\n", A);  //输出指针A、B的地址
    printf("%p\n", B);

    return 0;
}
```