链表

什么是链表

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。 相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而线性表和顺序表相应的时间复杂度分别是O(logn)和O(1)。

我们可以把链表想象成火车，火车头是链表的表头，没节车厢就是链表中的元素，车厢中的人或者物品就是元素的数据域，连接车厢的部件就是元素的指针。

特点

1.元素之间前后依赖，串联而成。

2.想要查找后面的元素必须经过前面的元素

3.元素前后不会出现多个元素连接的情况

优缺点

顺序存储时，相邻数据元素的存放地址也相邻（逻辑与物理统一）；要求内存中可用存储单元的地址必须是连续的。

优点：存储密度大（＝1？），存储空间利用率高。缺点：插入或删除元素时不方便。

链式存储时，相邻数据元素可随意存放，但所占存储空间分两部分，一部分存放结点值，另一部分存放表示结点间关系的指针

优点：插入或删除元素时很方便，使用灵活。缺点：存储密度小（<1），存储空间利用率低。

操作

插入

```
1.找到插入位置
2.先令待插入结点的next指针指向当前结点
3.令插入位置之前的结点的next指针指向插入结点
```

遍历

```
1.进入循环
2.遍历结点，并更新变量为当前结点的下一个结点
```

删除

1.遍历找到删除的位置
2.令前结点的next指针指向后结点
3.删除结点

代码实现

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

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

LinkedList insert(LinkedList head, Node *node, int index) {
    if (head == NULL) {
        if (index != 0) {
            return head;
        }
        head = node;
        return head;
    }
    if (index == 0) {
        node->next = head;
        head = node;
        return head;
    }
    Node *current_node = head;
    int count = 0;
    while (current_node->next != NULL && count < index - 1) {
        current_node = current_node->next;
        count++;
    }
    if (count == index - 1) {
        node->next = current_node->next;
        current_node->next = node;
    }
    return head;
}

void output(LinkedList head) {
    if (head == NULL) {
        return;
    }
    Node *current_node = head;
    while (current_node != NULL) {
        printf("%d ", current_node->data);
        current_node = current_node->next;
    }
    printf("\n");
}
```

```
LinkedList delete_node(LinkedList head, int index) {
    if (head == NULL) {
        return head;
    }
    Node *current_node = head;
    int count = 0;
    if (index == 0) {
        head = head->next;
        free(current_node);
        return head;
    }
    while (current_node->next != NULL && count < index - 1) {
        current_node = current_node->next;
        count++;
    }
    if (count == index - 1 && current_node->next != NULL) {
        Node *delete_node = current_node->next;
        current_node->next = delete_node->next;
        free(delete_node);
    }
    return head;
}

// 请在下面实现链表的反转函数 reverse
LinkedList reverse(LinkedList head) {
    if(head == NULL) {
        return head;
    }
    Node *next_node, *current_node;
    current_node = head->next;
    head->next = NULL;
    while(current_node != NULL) {
        next_node = current_node->next;
        current_node->next = head;
        head = current_node;
        current_node = next_node;
    }
    return head;
}


void clear(LinkedList head) {
    Node *current_node = head;
    while (current_node != NULL) {
        Node *delete_node = current_node;
        current_node = current_node->next;
        free(delete_node);
    }
}

int main() {
    LinkedList linkedlist = NULL;
```

```
    for (int i = 1; i <= 10; i++) {
        Node *node = (Node *)malloc(sizeof(Node));
        node->data = i;
        node->next = NULL;
        linkedlist = insert(linkedlist, node, i - 1);
    }
    output(linkedlist);
    linkedlist = delete_node(linkedlist, 3);
    output(linkedlist);
    linkedlist = reverse(linkedlist);
    output(linkedlist);
    clear(linkedlist);
    return 0;
}
```