法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

关注 小象学院

第八课　结构体与链表

林沐

内容概述

第一部分:结构体

1.结构体的引入
2.结构体的内存
3.结构体的别名
4.结构体的赋值
5.指针与结构体
6.结构体数组
7.结构体数组排序
8.结构体数组排序的回调函数
9.结构体与结构体指针作为函数参数
10.结构体数组作为函数参数，并返回结构体
11.结构体中的结构体
12.结构体中的指针
13.动态内存分配结构体
14.链表引入

第二部分: 链表

1.单链表定义
2.单链表的初始化与添加节点
3.单链表的查找、打印、释放
4.单链表的节点删除
5.单链表整体测试代码
6.链表算法题:链表逆序
7.拆解链表逆序的过程
8.使用循环逆置链表
9.链表逆序方法1:就地逆置法
10.链表逆序方法2:头插法
11.两种逆置链表方法的比较
12.链表逆序测试与leetcode提交结果

第一部分：结构体

通常情况下，我们需要使用**多个变量**描述现实生活中的**一个事物**，例如在描述学生时，需要使用字符串存储姓名，整型存储年龄，整型数组存储成绩，浮点型存储平均成绩。**关键字struct**可以定义各种类型的变量集合，称为**结构体(structure)**，在struct中的这些变量会被看作**一个单元**进行使用。
使用"struct 结构体名 结构体变量名"的方式**声明结构体**，使用"结构体变量名"+"."句点+"结构体成员变量名"的方式**使用**结构体内的成员变量。

```
The student:
name  = LinMu
age  = 28
scores: 80 97 71
average  = 82.67
请按任意键继续. . .
```

```c
#include <stdio.h>
struct student{
    char name[10];
    int age;
    int score[3];
    double average;
};

int main(){
    struct student stu = {
        "LinMu",
        28,
        {80, 97, 71},
        0.0
    };
    double sum = 0;
    int i;
    for (i = 0; i < 3; i++){
        sum += stu.score[i];
    }
    stu.average = sum / 3;
    printf("The student:\n");
    printf("name = %s\n", stu.name);
    printf("age = %d\n", stu.age);
    printf("scores: %d %d %d\n",
           stu.score[0], stu.score[1], stu.score[2]);
    printf("average = %.2lf\n", stu.average);
    return 0;
}
```

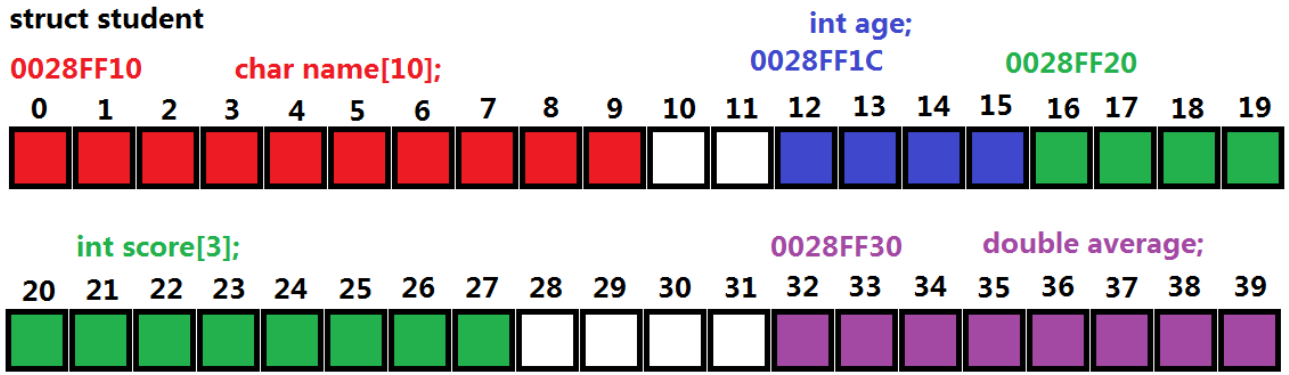
//代表学生的结构体
//该结构体将姓名、年龄、成绩、平均成绩这些变量合并为成一个单元

//使用{}初始化结构体，结构体中的变量按照顺序进行初始化

//结构体中成员变量的使用

结构体的内存

结构体在**声明**时，结构体内可以是**任何类型**的变量，如struct student结构体包括表示姓名的**字符串**，表示年龄的**整型**，表示三门成绩的**整型数组**，表示成绩的**浮点数**；这些都占用了**相应的**内存空间。结构体**总的内存占用**不一定是结构体中各个成员变量所占用内存的**总和**，这是因为内存在分配时存在**边界调整**。具体来讲，除了char型之外，2字节变量**起始地址**通常是2的倍数，4字节变量的**起始地址**通常是4的倍数，8字节变量的**起始地址**通常是8的倍数，这样中间就会有一些**空隙**。




```
sizeof(struct student) = 40
sum = 34
stu address = 0028FF10
stu.name address = 0028FF10
stu.age address = 0028FF1C
stu.score address = 0028FF20
stu.average address = 0028FF30
请按任意键继续. . .
```

```c
#include <stdio.h>
struct student{
    char name[10];
    int age;
    int score[3];
    double average;
};
```

```c
int main(){
    struct student stu;
    printf("sizeof(struct student) = %d\n", sizeof(stu));
    int sum  = sizeof(stu.name) + sizeof(stu.age) +
               sizeof(stu.score) + sizeof(stu.average);
    printf("sum = %d\n", sum);
    printf("stu address = %p\n", &stu);
    printf("stu.name address = %p\n", stu.name);
    printf("stu.age address = %p\n", &stu.age);
    printf("stu.score address = %p\n", stu.score);
    printf("stu.average address = %p\n", &stu.average);
    return 0;
}
```

结构体的别名

实际上，我们可以**省略struct**，直接使用**结构体名**定义结构体变量，但需要使用**typedef定义**，从而产生**新的类型**。该语句的使用方法：typedef struct 结构体名 新名字;
例如: typedef struct student student; 我们就可以使用student直接定义变量了。
typedef是用来为复杂的声明定义**简单的别名**，也可以为基础数据类型**换一个**其他的名字。这样做的好处是，将来对于某些变量可以**灵活快速**的改变其声明类型。

```
#include <stdio.h>
struct student{
    char name[10];
    int age;
    int score[3];
    double average;
};

typedef struct student student;

int main(){
    typedef struct student ABC;
    student stu1 = {
        "LinMu",
        28,
        {80, 97, 71},
        0.0
    };
    ABC stu2 = {
        "bbb",
        0,
        {0},
        0.0
    };
    printf("%s %s\n", stu1.name, stu2.name);

    //typedef int DATA;
    typedef double DATA;
    DATA num = 1.23;
    DATA num2 = 2.3;
    DATA num3 = 3.6;
    printf("num = %.2lf num2 = %.2lf num3 = %.2lf\n"
                                    , num, num2, num3);
    return 0;
}
```

//后续可以直接使用student定义该类型

//后续可以直接使用ABC定义该类型

//定义新的数据类型DATA，该数据类型可以由int或double来实现

结构体的赋值

可以使用**赋值运算符**“=”，将一个结构体的全部内容**拷贝**到另一个结构体变量中。结构体中的，如字符串、整型数组类似的变量也会进行**完全的复制**，即结构体对应的**内存的完整拷贝**。

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

struct student{
    char name[10];
    int age;
    int score[3];
    double average;
};

typedef struct student student;

int main(){
    student s1 = {
        "LinMu",
        28,
        {80, 97, 71},
        0.0
    };
    student s2;
    strcpy(s2.name, "Xiaoming");
    printf("name = %s\n", s2.name);
    s2 = s1;
    printf("name = %s\n", s2.name);
    printf("age = %d\n", s2.age);
    printf("score = %d %d %d\n", s2.score[0], s2.score[1], s2.score[2]);
    return 0;
}
```

//若对结构体中的字符串变量赋值，需使用字符串拷贝运算

//若对结构体整体进行拷贝，直接使用赋值运算符=

```
name = Xiaoming
name = LinMu
age = 28
score = 80 97 71
请按任意键继续. . .
```

指针与结构体

当使用某类型指针**指向**该类型的变量后，就可以通过**指针操作**该变量了。同样，**结构体类型**也可以有**对应的**指针类型存储该结构体变量的**地址**，利用该**结构体指针**就可以**操作**这个结构体变量了。在使用结构体指针**操作**结构体对应的内存时，可以使用**成员选择运算符"."**或使用**按指针选择成员运算符->**。使用**成员选择运算符"."**时，需要先通过**取消引用运算符***将指针转为对应变量，并且要在**加上括号**，因为成员选择运算符"."**优先级高于**取消引用运算符"*"。

```c
#include <stdio.h>
#include <string.h>

struct student{
    char name[20];
    int score;
};
typedef struct student student;

int main(){
    student s;
    student *ptr = &s;  //使用student指针ptr指向变量s

    strcpy(ptr->name, "LinMu");
    ptr->score = 85;
    printf("%s %d\n", ptr->name, ptr->score);

    strcpy((*ptr).name, "Xiaoming");
    (*ptr).score = 93;
    printf("%s %d\n", (*ptr).name, (*ptr).score);

    return 0;
}
```

```
LinMu 85
Xiaoming 93
请按任意键继续. . .
```

//使用按指针选择成员运算符->

//使用成员选择运算符"."，注意要加括号

结构体数组

存储**一组结构体数据**可以使用**结构体数组**，结构体数组的**声明**与普通数组声明没有本质区别。例如，声明一组学生结构体，并为这组学生结构体进行**初始化**。

```
#include <stdio.h>
struct student{
    char name[10];
    int age;
    int score[3];
    double average;
};

typedef struct student student;

int main(){
    student s[3] = {
        {"LinMu", 28, {0}, 0.0},
        {"Xiaoming", 30, {0}, 0.0},
        {"Fangfang", 18, {0}, 0.0}
    };

    int i;
    for (i = 0; i < 3; i++){
        printf("name = %s, age = %d\n", s[i].name, s[i].age);
    }

    return 0;
}
```

//结构体数组赋初值

//遍历结构体数组

```
name = LinMu, age = 28
name = Xiaoming, age = 30
name = Fangfang, age = 18
请按任意键继续. . .
```

结构体数组排序-课堂练习

已知一个**结构体数组**表示学生，该结构体数组中包含**两个元素**，表示姓名的**字符串**与表示成绩的**整型**。对该结构体数组进行排序，**成绩高**的排**在前面**，当**成绩相同**时，按照**姓名字符序**排序结构体。

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

struct student{
    char name[20];
    int score;
};

typedef struct student student;

void swap(student *s1, student *s2){
    student temp = *s1;
    4
    5
}
```

```
int main(){
    student s[20];
    int n;
    int i, j;
    scanf("%d", &n);
    for (i = 0; i < n; i++){
        scanf("%s %d", s[i].name, &s[i].score);
    }
    for (i = 0; i < n; i++){
        for (j = i + 1; j < n; j++){
            if ( 1 ){
                2
            }
            else if(s[i].score == s[j].score){
                if ( 3 ){
                    2
                }
            }
        }
    }
    printf("sort:\n");
    for (i = 0; i < n; i++){
        printf("%s %d\n", s[i].name, s[i].score);
    }
    return 0;
}
```

```
5
Xiaoming 78
LinMu 86
Fangfang 86
Jim 93
Peter 65
sort:
Jim 93
Fangfang 86
LinMu 86
Xiaoming 78
Peter 65
请按任意键继续. . .
```

3分钟，填写代码，有问题提出！

结构体数组排序-实现

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

struct student{
    char name[20];
    int score;
};

typedef struct student student;

void swap(student *s1, student *s2){
    student temp = *s1;
    *s1 = *s2;
    *s2 = temp;
}
```

```
5
Xiaoming 78
LinMu 86
Fangfang 86
Jim 93
Peter 65
sort:
Jim 93
Fangfang 86
LinMu 86
Xiaoming 78
Peter 65
请按任意键继续. . .
```

```
int main(){
    student s[20];
    int n;
    int i, j;
    scanf("%d", &n);
    for (i = 0; i < n; i++){
        scanf("%s %d", s[i].name, &s[i].score);
    }
    for (i = 0; i < n; i++){
        for (j = i + 1; j < n; j++){
            if ( s[i].score < s[j].score ){
                swap(&s[i], &s[j]);
            }
            else if(s[i].score == s[j].score){
                if ( strcmp(s[i].name, s[j].name) > 0 ){
                    swap(&s[i], &s[j]);
                }
            }
        }
    }
    printf("sort:\n");
    for (i = 0; i < n; i++){
        printf("%s %d\n", s[i].name, s[i].score);
    }
    return 0;
}
```

结构体数组排序的回调函数，课堂练习

使用qsort对**结构体数组**进行排序，qsort用法:
函数原型:

void qsort(void *base, int num, int width, int (*compare)(const void *, const void *))

void *base : 指向**待排序**的数组;
int num: 待排序数组的**元素个数**;
int width: 待排序数组的**元素大小**。
int (*compare)(const void *, const void *):
排序时的**回调函数**，用户传进来一个回调函数，在比较两元素的时候，使用这个回调函数对**元素进行比较**。

```
5
Xiaoming 78
LinMu 86
Fangfang 86
Jim 93
Peter 65
sort:
Jim 93
Fangfang 86
LinMu 86
Xiaoming 78
Peter 65
请按任意键继续. . .
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct student{
    char name[20];
    int score;
};
typedef struct student student;
int student_cmp(const void *a, const void *b){
    student *s1 = (student *)a;
    student *s2 = (student *)b;
    if (  1  ){
        return s2->score - s1->score;
    }
    return  2
}

int main(){
    student s[20];
    int n;
    int i, j;
    scanf("%d", &n);
    for (i = 0; i < n; i++){
        scanf("%s %d", s[i].name, &s[i].score);
    }
     3
    printf("sort:\n");
    for (i = 0; i < n; i++){
        printf("%s %d\n", s[i].name, s[i].score);
    }
    return 0;
}
```

课堂练习，实现

```
5
Xiaoming 78
LinMu 86
Fangfang 86
Jim 93
Peter 65
sort:
Jim 93
Fangfang 86
LinMu 86
Xiaoming 78
Peter 65
请按任意键继续. . .
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct student{
    char name[20];
    int score;
};
typedef struct student student;
int student_cmp(const void *a, const void *b){
    student *s1 = (student *)a;
    student *s2 = (student *)b;
    if ( s1->score != s2->score ){
        return s2->score - s1->score;
    }
    return strcmp(s1->name, s2->name);
}
int main(){
    student s[20];
    int n;
    int i, j;
    scanf("%d", &n);
    for (i = 0; i < n; i++){
        scanf("%s %d", s[i].name, &s[i].score);
    }
    qsort(s, n, sizeof(s[0]), student_cmp);
    printf("sort:\n");
    for (i = 0; i < n; i++){
        printf("%s %d\n", s[i].name, s[i].score);
    }
    return 0;
}
```

结构体与结构体指针作为函数参数

将**结构体**作为**函数参数**与一般变量作为参数的传递没有不同，但**直接**将结构体作为参数会对结构体进行**副本**的拷贝，若**结构体较大**，会占用**大量的内存**，复制时也耗费**大量时间**。所以我们一般将**结构体指针**作为函数参数，直接传递结构体的**原始地址**，从而**节省时间与空间**的开销。

```c
#include <stdio.h>
struct student{
    char name[10];
    int age;
    int score[3];
};
typedef struct student student;
```

```c
int is_a_better_than_b(student a, student b){
    int i;
    int sum_a = 0;
    int sum_b = 0;
    for (i = 0; i < 3; i++){
        sum_a += a.score[i];
        sum_b += b.score[i];
    }
    return sum_a > sum_b;
}
```

//直接传递结构体占用大量的栈空间

```c
int is_a_better_than_b_2(student *a, student *b){
    int i;
    int sum_a = 0;
    int sum_b = 0;
    for (i = 0; i < 3; i++){
        sum_a += a->score[i];
        sum_b += a->score[i];
    }
    return sum_a > sum_b;
}
```

//一般写成指针形式传递参数，节省复制的时间

```c
int main(){
    student s1 = {
        "LinMu",
        28,
        {80, 97, 71}
    };
    student s2 = {
        "XiaoMing",
        25,
        {60, 85, 73}
    };
    student s3 = {
        "Peter",
        26,
        {98, 91, 89}
    };
    if (is_a_better_than_b(s1, s2)){
        printf("%s is better.\n", s1.name);
    }
    else{
        printf("%s is better.\n", s2.name);
    }
    if (is_a_better_than_b_2(&s2, &s3)){
        printf("%s is better.\n", s2.name);
    }
    else{
        printf("%s is better.\n", s3.name);
    }
    return 0;
}
```

```
LinMu is better.
Peter is better.
请按任意键继续. . .
```

结构体数组作为函数参数，并返回结构体

结构体数组作为函数参数与普通数组作为函数参数**没有区别**，同样的需要给出结构体数组中**有效元素个数**。当将**结构体**作为函数**返回值**时，若直接返回结构体，需要**内存的拷贝**，函数若要返回结构体，一般**返回结构体指针**的形式，这样可以节省结构体复制时需要的**空间与时间**。在使用结构体时，尽量**避免**对结构体**直接拷贝**操作，尽量**使用指针**操作结构体。

```c
#include <stdio.h>

struct student{
    char name[10];
    int age;
    int score[3];
};

typedef struct student student;

student find_best_student(student s[], int n){
    student best;
    int max_score = -1;
    int i, j;
    for (i = 0; i < n; i++){
        int sum = 0;
        for (j = 0; j < 3; j++){
            sum += s[i].score[j];
        }
        if (max_score < sum){
            max_score = sum;
            best = s[i];
        }
    }
    return best;
}
```

//结构体的拷贝，耗费大量时间

```c
student* find_best_student_2(student s[], int n){
    student *best;
    int max_score = -1;
    int i, j;
    for (i = 0; i < n; i++){
        int sum = 0;
        for (j = 0; j < 3; j++){
            sum += s[i].score[j];
        }
        if (max_score < sum){
            max_score = sum;
            best = &s[i];
        }
    }
    return best;
}
int main(){
    student s[3] = {
        {"LinMu", 28, {80, 97, 71}},
        {"XiaoMing", 25, {60, 85, 73}},
        {"Peter", 26, {98, 91, 89}}
    };
    student best = find_best_student(s, 3);
    student *best_ptr = find_best_student_2(s, 3);
    printf("best name = %s\n", best.name);
    printf("best name = %s\n", best_ptr->name);
    return 0;
}
```

```
best name = Peter
best name = Peter
请按任意键继续. . .
```

//结构体的拷贝，耗费大量时间

结构体中的结构体

所有**基本数据类型**(数组)都可以作为结构体的成员。除此之外，**结构体**也可以**作为**另一个结构体的**成员**。结构体内的结构体可以在**结构体外**定义，也可以在**结构体中**定义；在结构体中定义的结构体**无法在结构体外**使用。一般情况下我们**不在**结构体内定义结构体。

```c
#include <stdio.h>

struct subject{
    int math;
    int english;
    int chemistry;
};
typedef struct subject subject;

struct student{
    char name[10];
    int age;
    subject score;
};
typedef struct student student;

struct student2{
    struct subject2{
        int math;
        int english;
        int chemistry;
    };
    char name[10];
    int age;
    struct subject2 score;
};
```

//结构体外定义结构体

//结构体内声明结构体变量

//结构体内定义结构体

```c
int main(){
    student s1 = {
        "LinMu",
        28,
        {80, 97, 71}
    };
    struct student2 s2 = {
        "XiaoMing",
        25,
        {60, 85, 73}
    };
    printf("name = %s, age = %d, M = %d E = %d C = %d\n",
        s1.name, s1.age, s1.score.math,
        s1.score.english, s1.score.chemistry);

    printf("name = %s, age = %d, M = %d E = %d C = %d\n",
        s2.name, s2.age, s2.score.math,
        s2.score.english, s2.score.chemistry);

    return 0;
}
```

```
name = LinMu, age = 28, M = 80 E = 97 C = 71
name = XiaoMing, age = 25, M = 60 E = 85 C = 73
请按任意键继续. . .
```

结构体中的指针

指针变量是用来存储内存地址的变量，结构体中也可以设置**指针变量成员**。在使用包含指针变量的结构体时，要**特别注意**，复制这样的结构体并**不对**其指针变量指向的**内存进行复制**，只是复制了指针中存储的地址。

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

struct student{
    char *name;
    int age;
};

typedef struct student student;

int main(){
    char buffer[100] = "LinMu";
    student s1;
    student s2;
    s1.name = buffer;
    s1.age = 28;
    s2 = s1;
    printf("s1 : %s %p %d\n", s1.name, s1.name, s1.age);
    printf("s2 : %s %p %d\n", s2.name, s2.name, s2.age);

    strcpy(s1.name, "Peter");
    printf("s2 : %s %p %d\n", s2.name, s2.name, s2.age);
    printf("buffer = %s\n", buffer);

    return 0;
}
```

```
s1 : LinMu 0028FED0 28
s2 : LinMu 0028FED0 28
s2 : Peter 0028FED0 28
buffer = Peter
请按任意键继续. . .
```

//对结构体的复制，复制结构体中的指针指向(存储的地址)，不会复制出一片空间

动态内存分配结构体

使用malloc为结构体**动态申请空间**，使用malloc方式与为普通变量申请空间没有不同。要注意的是，如果结构体中有**指针变量**，该指针变量指向的空间需要进行**设置**或**动态申请**。

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

struct student1{
    char name[20];
    int age;
};

struct student2{
    char *name;
    int age;
};

typedef struct student1 student1;
typedef struct student2 student2;

int main(){
    char *name[3] = {
        "LinMu",
        "XiaoMing",
        "Peter"
    };
    //动态分配一个结构体
    student1 *s = malloc(sizeof(student1));
    strcpy(s->name, "Fangfang");
    s->age = 21;

    int i = 0;
    //动态分配结构体数组
    student1 *s2 = malloc(sizeof(student1) * 3);
    for (i = 0; i < 3; i++){
        strcpy(s2[i].name, name[i]);
        s2[i].age = 25 + i;
    }
    student1 *s3[3] = {0};
    //结构体指针数组，为每个指针动态分配结构体，功能类似结构体数组
    for (i = 0; i < 3; i++){
        s3[i] = malloc(sizeof(student1));
        strcpy(s3[i]->name, name[i]);
        s3[i]->age = 25 + i * 2;
    }
    //注意，name的空间也要动态分配
    student2 *s4 = malloc(sizeof(student2));
    s4->name = malloc(20 * sizeof(char));
    strcpy(s4->name, "Jim");
    s4->age = 20;

    printf("s %s %d\n\n", s->name, s->age);
    for (i = 0; i < 3; i++){
        printf("s2[%d] %s %d\n", i, s2[i].name, s2[i].age);
    }
    printf("\n");
    for (i = 0; i < 3; i++){
        printf("s3[%d] %s %d\n", i, s3[i]->name, s3[i]->age);
    }
    printf("\n");
    printf("s4 %s %d\n\n", s4->name, s4->age);

    free(s);
    free(s2);
    for (i = 0; i < 3; i++){
        free(s3[i]);
    }
    free(s4->name);
    free(s4);

    return 0;
}
```

```
s Fangfang 21

s2[0] LinMu 25
s2[1] XiaoMing 26
s2[2] Peter 27

s3[0] LinMu 25
s3[1] XiaoMing 27
s3[2] Peter 29

s4 Jim 20

请按任意键继续. . .
```

//释放各个指针指向的内存空间

链表的引入

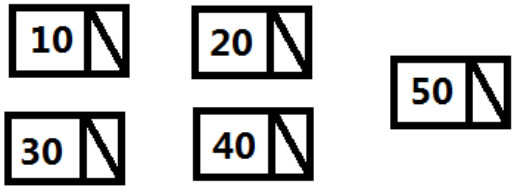
//定义ListNode节点结构体

```
typedef struct ListNode ListNode;

struct ListNode {
    int val;
    ListNode *next;
};
```

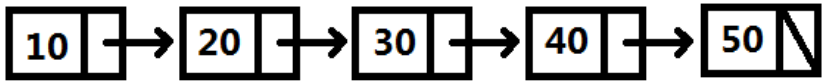
//存储数据元素的 数据域

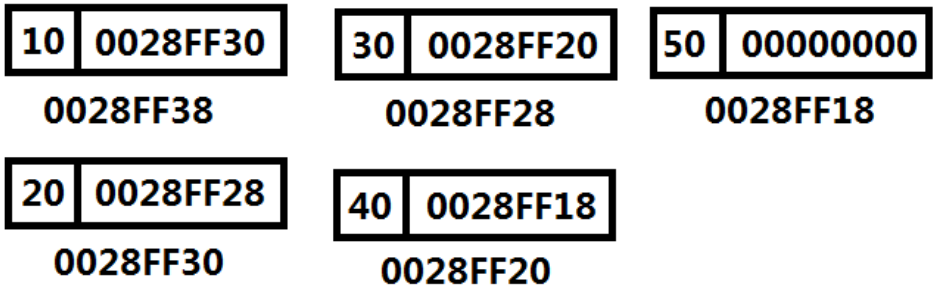
//存储自身地址的 指针域

1.创建5个节点:

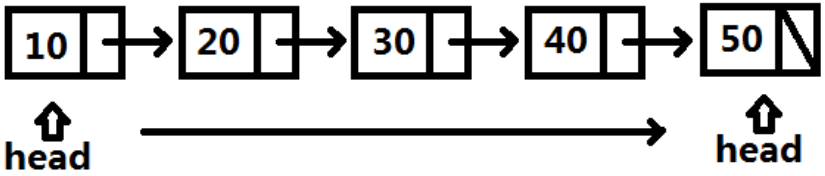


2. 通过next指针 将它们连接在一起:



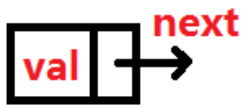


3.遍历它们，并打印节点的值:



链表引入-课堂练习

```
#include <stdio.h>

typedef struct ListNode ListNode;

struct ListNode {
    int val;          //存储数据元素的 数据域
    ListNode *next;   //存储自身地址的 指针域
};

int main(){
    ListNode a;
    ListNode b;
    ListNode c;
    ListNode d;
    ListNode e;
    a.val = 10;
    b.val = 20;
    c.val = 30;
    d.val = 40;
    e.val = 50;
    a.next = &b;
    1
    c.next = &d;
    d.next = &e;
    2
    ListNode *head = &a;
    while(head){
        printf("%d %p %p\n", head->val, head, head->next);
    3
    }
    return 0;
}
```

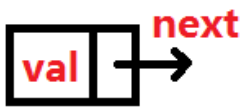



```
10 0028FF38 0028FF30
20 0028FF30 0028FF28
30 0028FF28 0028FF20
40 0028FF20 0028FF18
50 0028FF18 00000000
请按任意键继续. . .
```

3分钟，填写代码，有问题提出！

链表引入-实现

```c
#include <stdio.h>

typedef struct ListNode ListNode;

struct ListNode {
    int val;          //存储数据元素的 数据域
    ListNode *next;   //存储自身地址的 指针域
};

int main(){
    ListNode a;
    ListNode b;
    ListNode c;
    ListNode d;
    ListNode e;
    a.val = 10;
    b.val = 20;
    c.val = 30;
    d.val = 40;
    e.val = 50;
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    e.next = NULL;
    ListNode *head = &a;
    while(head){
        printf("%d %p %p\n", head->val, head, head->next);
        head = head->next;
    }
    return 0;
}
```




```
10 0028FF38 0028FF30
20 0028FF30 0028FF28
30 0028FF28 0028FF20
40 0028FF20 0028FF18
50 0028FF18 00000000
请按任意键继续. . .
```

课间休息10分钟！

有问题提出！

单链表定义

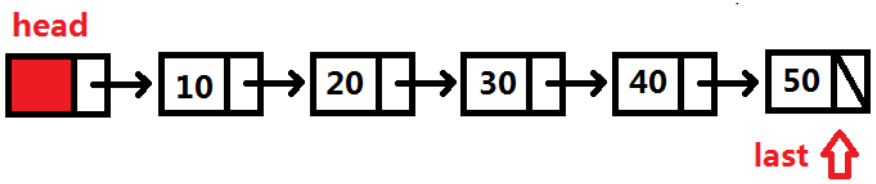
单链表是**非连续、非顺序**的**链式**存储结构，由一些列结点组成，它们的**逻辑顺序**是通过链表中的指针链接顺序表示的。结点中包括两部分，**数据域**与**指针域**，数据域用来存储**相应的数据**，指针域用来存储链表节点的**逻辑关系**。在单链表中，一般分为**带头结点**与**不带头结点**的两种链表。一般带头节点的链表**更容易实现**添加节点、删除节点、遍历节点等操作，因为我们不必对链表的第一个数据节点进行**特殊处理**。

带头结点的单链表:List，**head**：链表的**头结点**，注意这是一个**结构体变量**；

***last**：指向链表**最后一个节点**的指针，注意这是一个**指针**。

```c
#include <stdio.h>

typedef struct ListNode ListNode;
struct ListNode {
    int val;
    ListNode *next;
};
typedef struct List List;
struct List{
    ListNode head;
    ListNode *last;
};
```

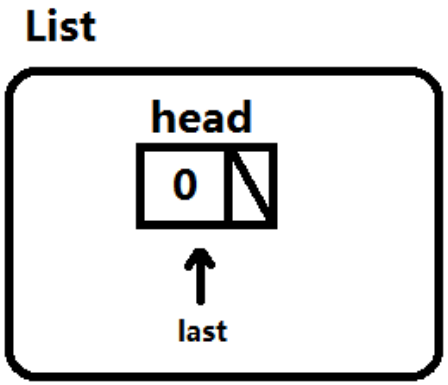
```c
void list_init(List *list);  //链表的初始化
void list_destroy(List *list); //释放链表中的所有节点
void list_insert(List *list, int data); //插入一个值为data的元素
void list_erase(List *list, int data); //删除链表中值为data的节点
ListNode* list_find(List *list, int data); //查找值为data的节点，返回该节点指针
void list_print(List *list);  //打印链表

int main(){
    List list;
    return 0;
}
```



单链表的初始化与添加节点

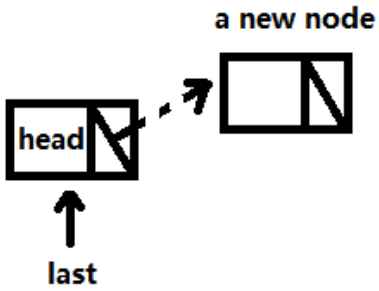
void list_init(List *list)作用是将**链表初始化**，为后续链表相应的操作**做准备**。在初始化时，将指向**链表尾部**的指针*last指向链表头结点head，这样**当前**链表的**最后一个**节点即为**头节点**。头节点中的**数据域**一般没什么用，可以用来存储链表节点个数等。

void list_insert(List *list, int data)向链表插入数据时，首先要开辟**新的链表节点**空间，再使用**last指针**进行节点的**连接**。

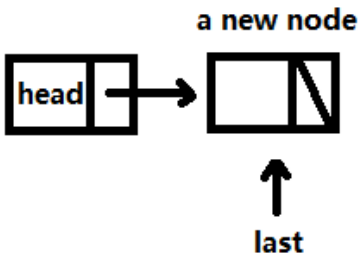


链表中没有节点时:

链接节点:list->last->next = new_node

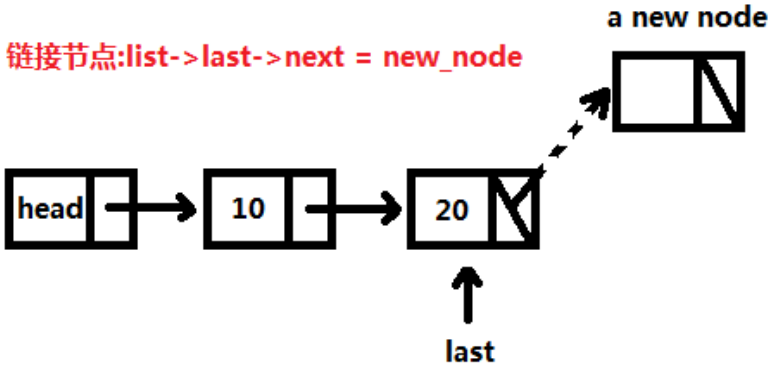


移动指针:list->last = new_node

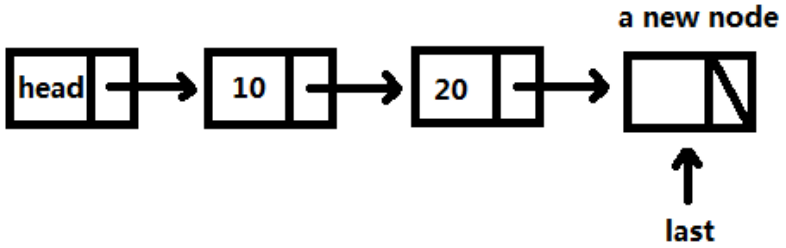


链表中有节点时:

链接节点:list->last->next = new_node

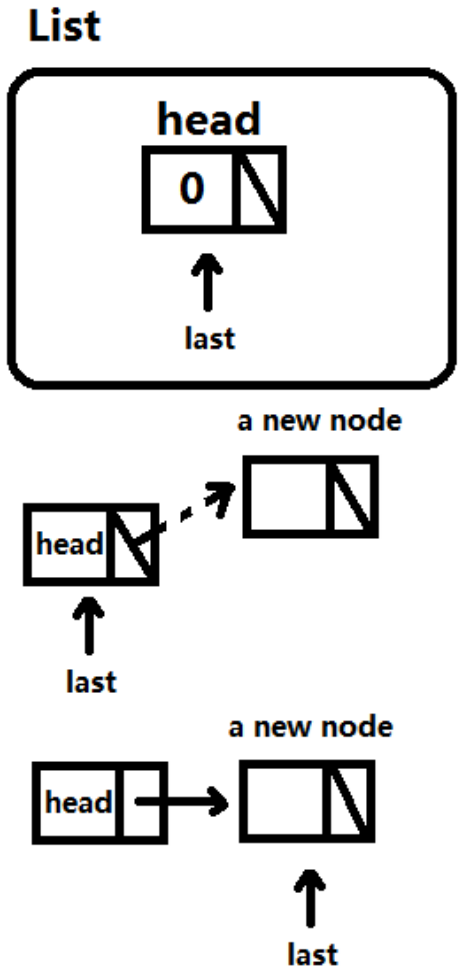


移动指针:list->last = new_node



单链表的初始化与添加节点-课堂练习

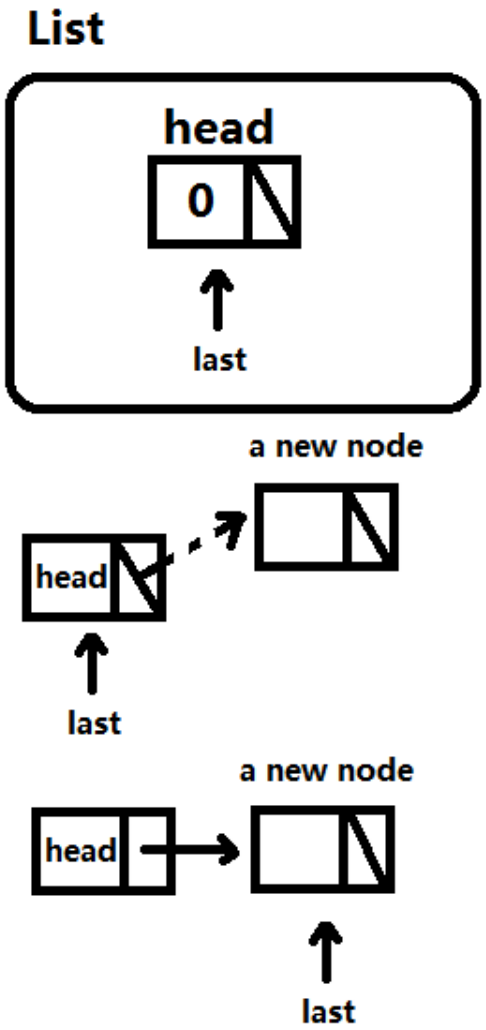
```
void list_init(List *list){
    list->head.val = 0;
    list->head.next = NULL;
    1
}

void list_insert(List *list, int data){
    ListNode *new_node = malloc(sizeof(ListNode));
    new_node->val = data;
    new_node->next = NULL;
    2
    3
}
```



3分钟，填写代码，有问题提出！

单链表的初始化与添加节点-实现

```
void list_init(List *list){
    list->head.val = 0;
    list->head.next = NULL;
    list->last = &list->head;
}

void list_insert(List *list, int data){
    ListNode *new_node = malloc(sizeof(ListNode));
    new_node->val = data;
    new_node->next = NULL;
    list->last->next = new_node;
    list->last = new_node;
}
```



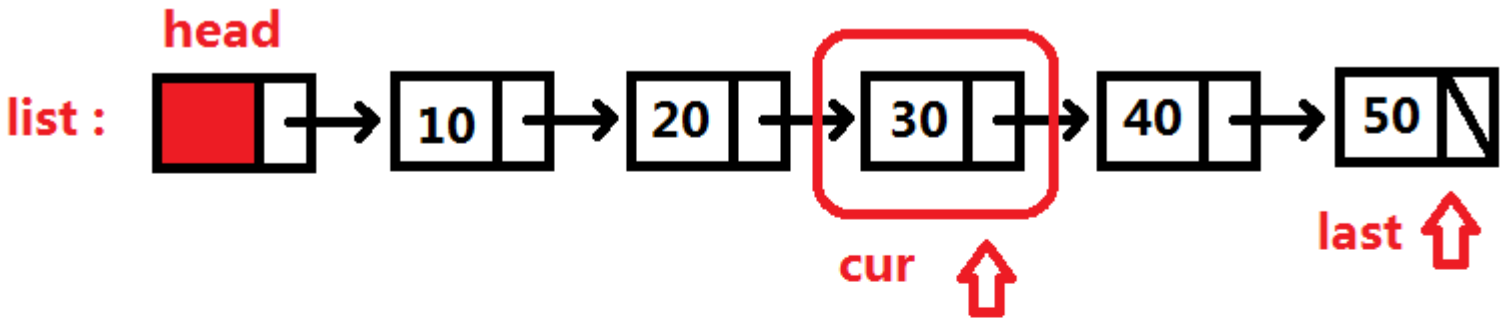
单链表的查找、打印、释放

查找(find)、打印(print)、释放(destroy)链表，实际上都是对于**链表节点的遍历**:

list_find(List *list, int data)：在链表list中，**查找值为data的节点**，如果找到值为data的节点，返回指向该节点的指针(**第一个**值为data的节点)；否则返回空。

list_print(List *list)：**打印链表list中所有元素**，打印效果如，head->[ ]->...->[ ]

list_destroy(List *list)：**释放**链表list中的**所有数据节点的空间**(头节点无需释放)，释放时，暂时存储cur的值，cur指向下一个节点后，再删除该节点。




```
void list_travel(List *list){
    ListNode *cur = list->head.next;
    while(cur){
        //do something
        cur = cur->next;
        //do something
    }
}
```

单链表的查找、打印、释放-课堂练习

```
void list_print(List *list){
    ListNode *cur = 1
    printf("head");
    while(cur){
        printf("->[%d]",cur->val);
        cur = cur->next;
    }
    printf("\n");
}
ListNode* list_find(List *list, int data){
    ListNode *cur = 1
    while(cur){
        if( 2 ){
            3
        }
        cur = cur->next;
    }
    return NULL;
}
```

```
void list_destroy(List *list){
    ListNode *cur = 1
    while(cur){
        ListNode *del = 4
        cur = cur->next;
        5
    }
    list_init(list);    //将链表重新初始化，清空数据
}
```

3分钟，填写代码，有问题提出！

单链表的查找、打印、释放-实现

```
void list_print(List *list){
    ListNode *cur = list->head.next;
    printf("head");
    while(cur){
        printf("->[%d]",cur->val);
        cur = cur->next;
    }
    printf("\n");
}
ListNode* list_find(List *list, int data){
    ListNode *cur = list->head.next;
    while(cur){
        if( cur->val == data ){
            return cur;
        }
        cur = cur->next;
    }
    return NULL;
}
```

```
void list_destroy(List *list){
    ListNode *cur = list->head.next;
    while(cur){
        ListNode *del = cur;
        cur = cur->next;
        free(del);
    }
    list_init(list);   //将链表重新初始化，清空数据
}
```

单链表的节点删除

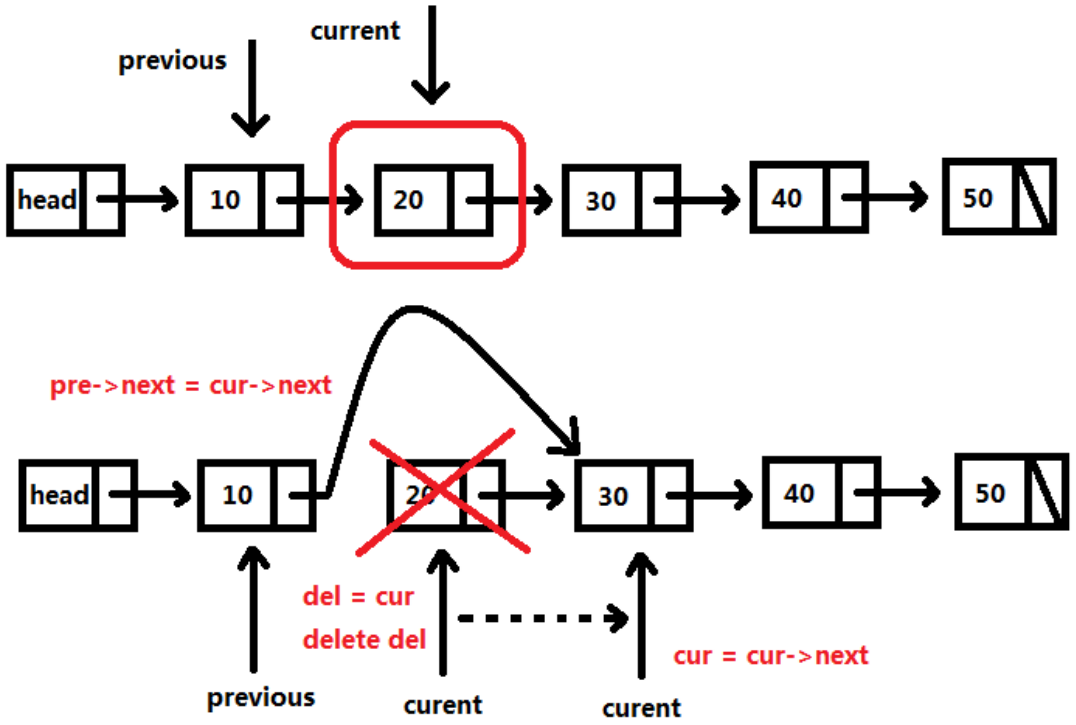
list_erase(List *list, int data)：**删除**链表中**所有值为data**的节点，在这个过程中需要用**两个指针**遍历链表，**previous指针**与**current指针**。previous指针指向的节点是current指针指向的节点的**前一个节点**，同时利用previous指针与current指针遍历链表，当**找到待删除节点**后(值域为data的节点):

1.**记录**待删除节点地址，设置del指针指向current指针指向的节点。

2.**修改**previous指向的节点的next域，使它指向current指针指向节点的下一个。

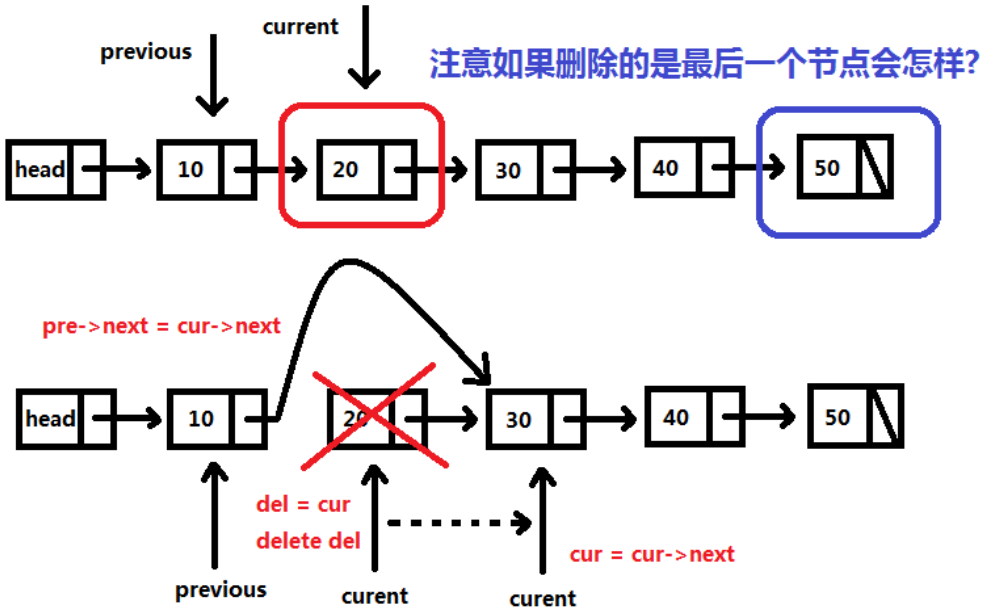
3.current指针向后**移动**。

4.**释放**del指针指向的节点空间。



单链表的节点删除-课堂练习

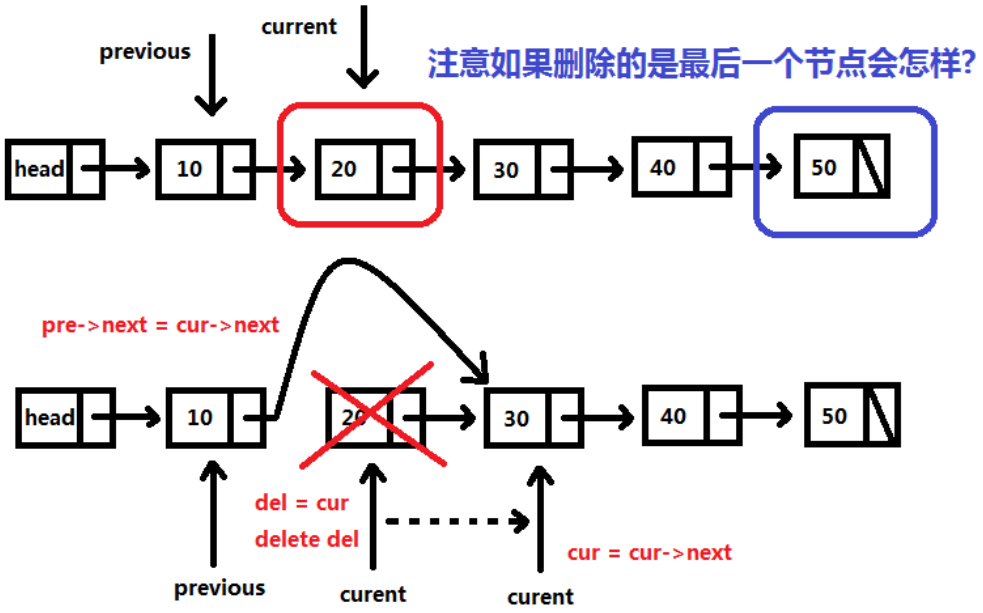
```
void list_erase(List *list, int data){
    ListNode *pre = 1
    ListNode *cur = list->head.next;
    while(cur){
        if( 2 ){
            ListNode *del = cur;
            3
            cur = cur->next;
            free(del);
        }
        else{
            4
            cur = cur->next;
        }
    }
    5
}
```



3分钟，填写代码，有问题提出！

单链表的节点删除-实现

```
void list_erase(List *list, int data){
    ListNode *pre = &list->head;
    ListNode *cur = list->head.next;
    while(cur){
        if( cur->val == data ){
            ListNode *del = cur;
            pre->next = cur->next;
            cur = cur->next;
            free(del);
        }
        else{
            pre = cur;
            cur = cur->next;
        }
    }
    list->last = pre;
}
```



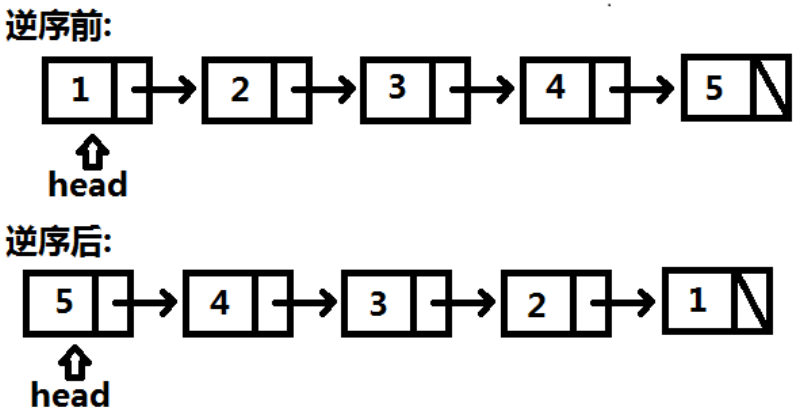
单链表整体测试

```
int main(){
    List list;
    list_init(&list);

    list_insert(&list, 10);
    list_insert(&list, 2);
    list_insert(&list, 10);
    list_insert(&list, 3);
    list_insert(&list, 10);
    list_print(&list);

    list_erase(&list,2);
    list_print(&list);

    list_erase(&list,10);
    list_print(&list);

    list_insert(&list, -99);
    list_insert(&list, 88);
    list_print(&list);

    ListNode *node = list_find(&list, -99);
    printf("node next data = %d\n", node->next->val);
    list_destroy(&list);

    return 0;
}
```

```
head->[10]->[2]->[10]->[3]->[10]
head->[10]->[10]->[3]->[10]
head->[3]
head->[3]->[-99]->[88]
node next data = 88
请按任意键继续. . .
```

链表算法题:链表逆序

已知链表**第一个节点**指针head，将**链表**逆序。(不可申请**额外**空间)



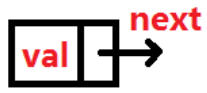

```
#include <stdio.h>

struct ListNode {
    int val;
    struct ListNode *next;
};

struct ListNode* reverseList(struct ListNode* head) {
}
```

//存储元素的数据域

//存储下一个节点地址的指针域

//链表第一个节点的指针，注意这个链表第一个节点是存储数据的

//返回逆置后的链表的第一个节点



选自 **LeetCode 206. Reverse Linked List**

https://leetcode.com/problems/reverse-linked-list/description/

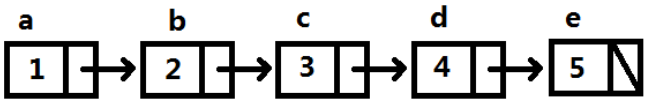
拆解链表逆序的过程

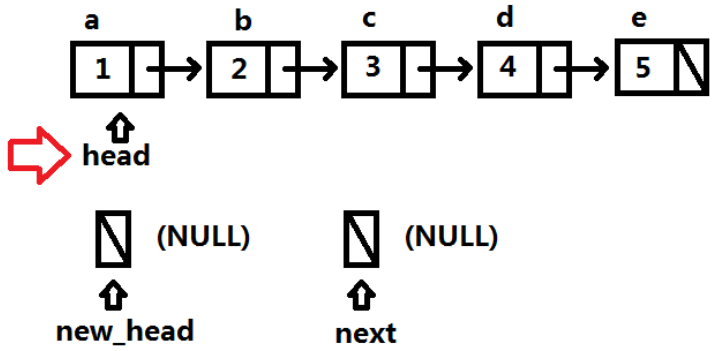
```c
#include <stdio.h>

struct ListNode {
    int val;
    struct ListNode *next;
};

typedef struct ListNode ListNode;

void print_list(ListNode *head,
                const char *list_name){
    printf("%s :", list_name);
    if (!head){
        printf("NULL\n");
        return;
    }
    while(head){
        printf("[%d] ", head->val);
        head = head->next;
    }
    printf("\n");
}
```

```c
int main(){
    ListNode a;
    ListNode b;
    ListNode c;
    ListNode d;
    ListNode e;
    a.val = 1;
    a.next = &b;
    b.val = 2;
    b.next = &c;
    c.val = 3;
    c.next = &d;
    d.val = 4;
    d.next = &e;
    e.val = 5;
    e.next = 0;

    ListNode *head = &a;
    ListNode *new_head = NULL;
    ListNode *next = NULL;
    print_list(head, "old");
    print_list(new_head, "new");
    ●●●
    return 0;
}
```

1.构造5个节点a,b,c,d,e；并对它们的val做初始化。

2.将a,b,c,d,e 5个节点链接在一起。






```
old :[1] [2] [3] [4] [5]
new :NULL
```

```
old :[2] [3] [4] [5]
new :[1]
```

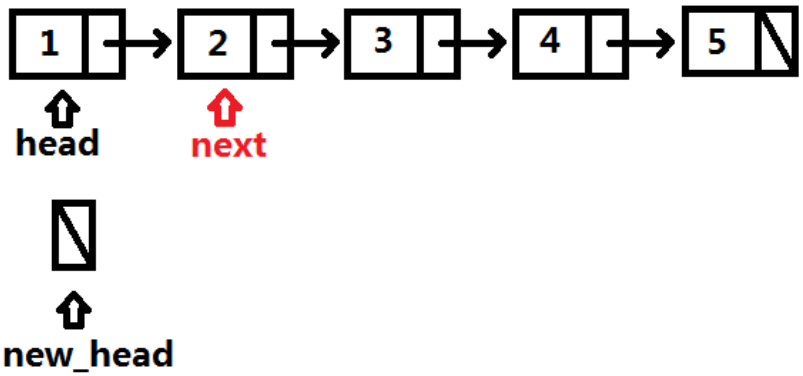
思考：

…中的的代码，如何**一个节点一个节点**的进行逆置？

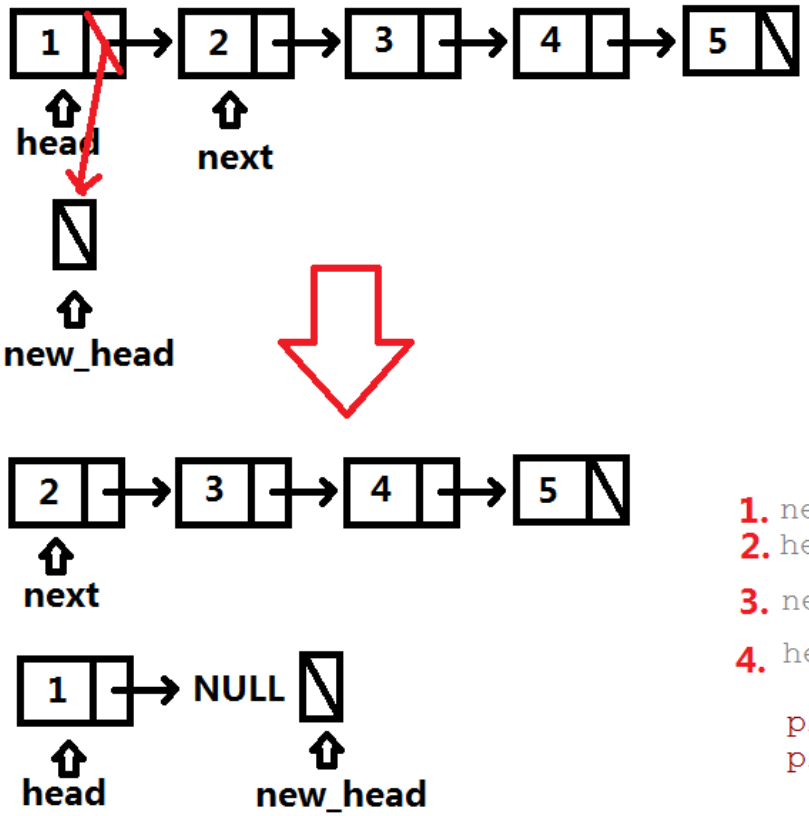
逆置第1个节点

第一组代码:

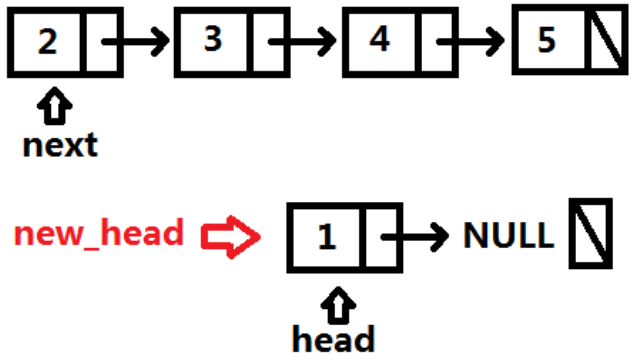
1. next = head->next;



2.head->next = new_head;

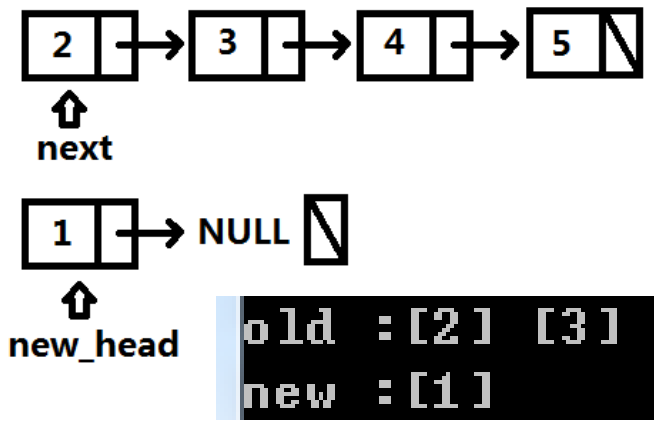


3.new_head = head;



head

4.head = next;



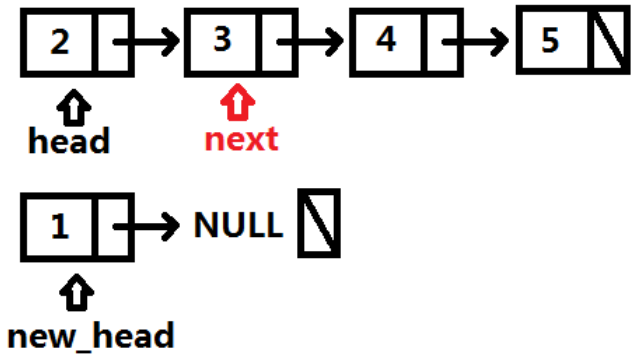

```
old :[2] [3] [4] [5]
new :[1]
```

```
1. next = head->next;
2. head->next = new_head;
3. new_head = head;
4. head = next;

   print_list(head, "old");
   print_list(new_head, "new");
```

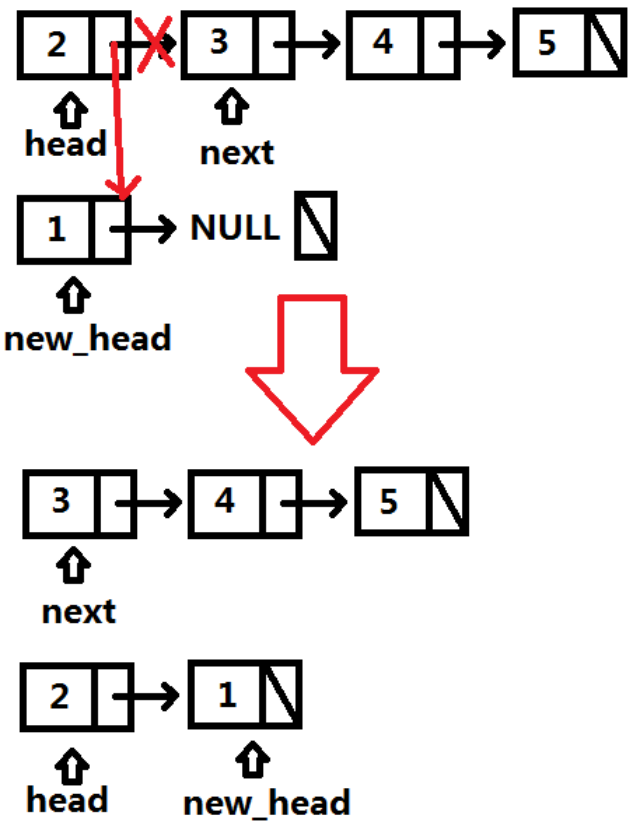
逆置第2个节点

第2组代码:

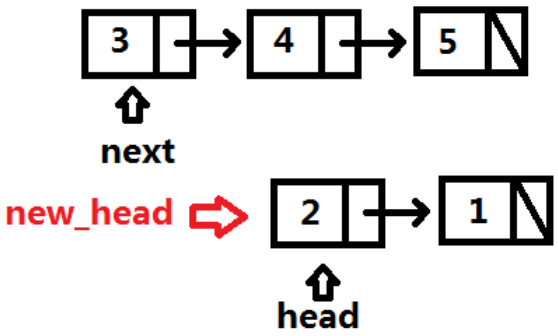
1. next = head->next;



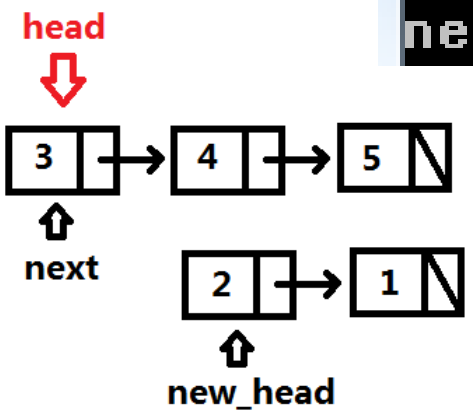
2.head->next = new_head;



3.new_head = head;



4.head = next;

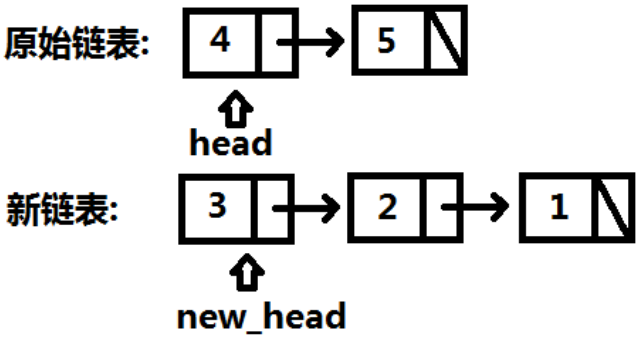



```
old :[3] [4] [5]
new :[2] [1]
```

```
1. next = head->next;
2. head->next = new_head;
3. new_head = head;
4. head = next;

   print_list(head, "old");
   print_list(new_head, "new");
```

逆置第3,4,5节点

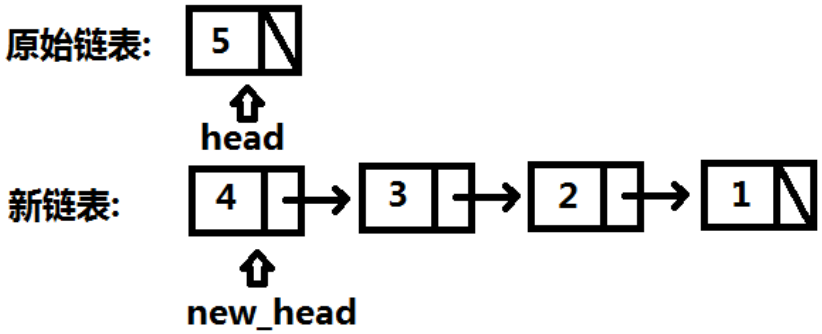
第3组代码:




```
next = head->next;
head->next = new_head;
new_head = head;
head = next;
print_list(head, "old");
print_list(new_head, "new");
```

```
old :[4] [5]
new :[3] [2] [1]
```

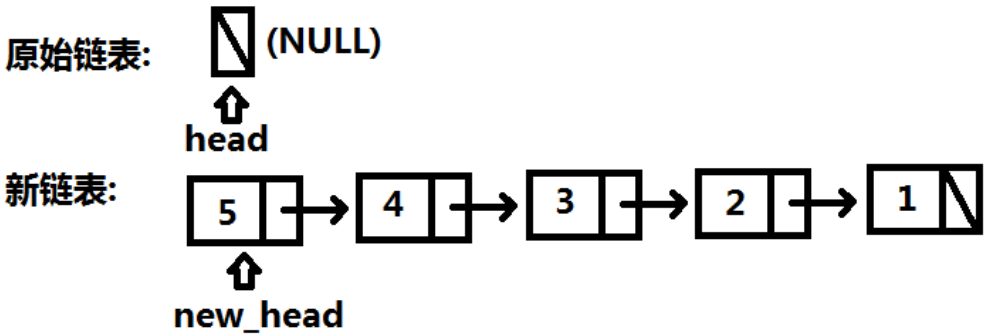
第4组代码:




```
next = head->next;
head->next = new_head;
new_head = head;
head = next;
print_list(head, "old");
print_list(new_head, "new");
```

```
old :[5]
new :[4] [3] [2] [1]
```

第5组代码:




```
next = head->next;
head->next = new_head;
new_head = head;
head = next;
print_list(head, "old");
print_list(new_head, "new");
```

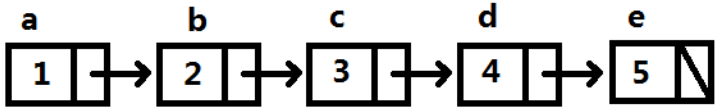
```
old :NULL
new :[5] [4] [3] [2] [1]
```

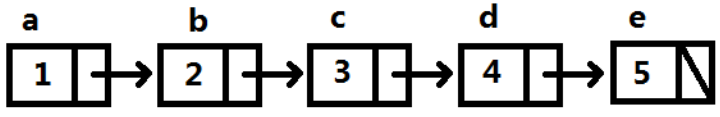
使用循环逆置链表

```
int main(){
    ListNode a;
    ListNode b;
    ListNode c;
    ListNode d;
    ListNode e;
    a.val = 1;
    a.next = &b;
    b.val = 2;
    b.next = &c;
    c.val = 3;
    c.next = &d;
    d.val = 4;
    d.next = &e;
    e.val = 5;
    e.next = 0;
```

1.构造5个节点a,b,c,d,e；并对它们的val做初始化。

2.将a,b,c,d,e 5个节点链接在一起。






```
    ListNode *head = &a;
    ListNode *new_head = NULL;
    ListNode *next = NULL;
    print_list(head, "old");
    print_list(new_head, "new");
```

head

(NULL)

(NULL)

new_head

next

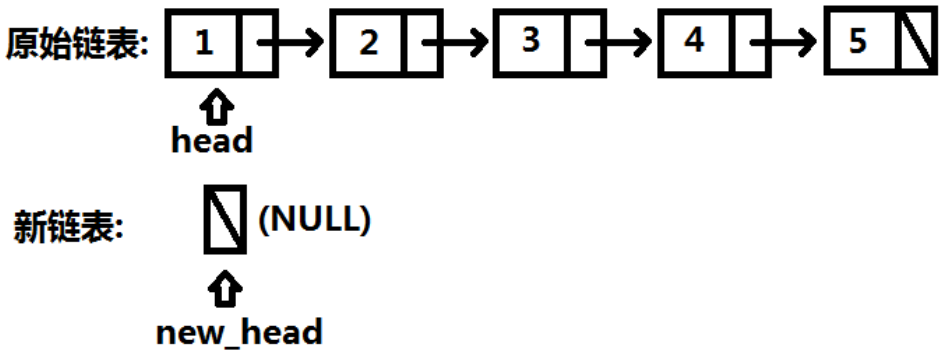
```
    int i;
    for (int i = 0; i < 5; i++){
        next = head->next;
        head->next = new_head;
        new_head = head;
        head = next;
        print_list(head, "old");
        print_list(new_head, "new");
    }
    return 0;
}
```

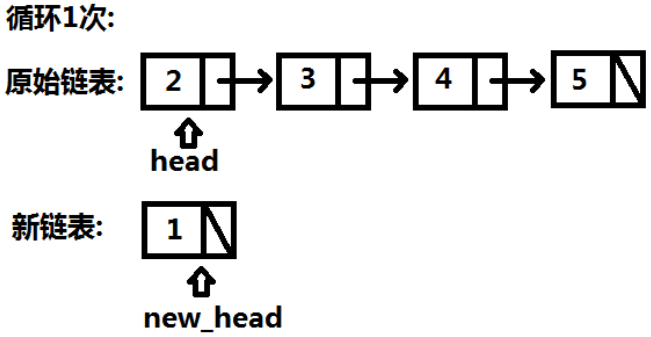
//利用循环逆置5个节点

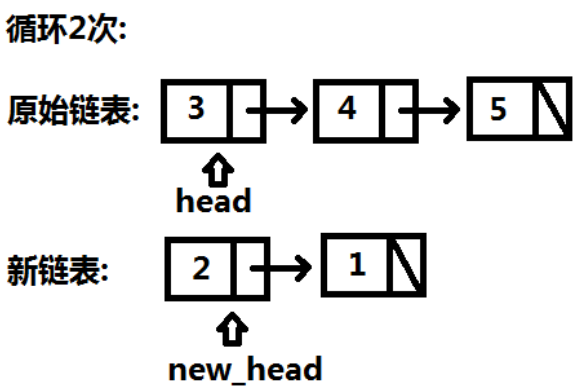
```
old :[1] [2] [3] [4] [5]
new :NULL
old :[2] [3] [4] [5]
new :[1]
old :[3] [4] [5]
new :[2] [1]
old :[4] [5]
new :[3] [2] [1]
old :[5]
new :[4] [3] [2] [1]
old :NULL
new :[5] [4] [3] [2] [1]
请按任意键继续. . .
```

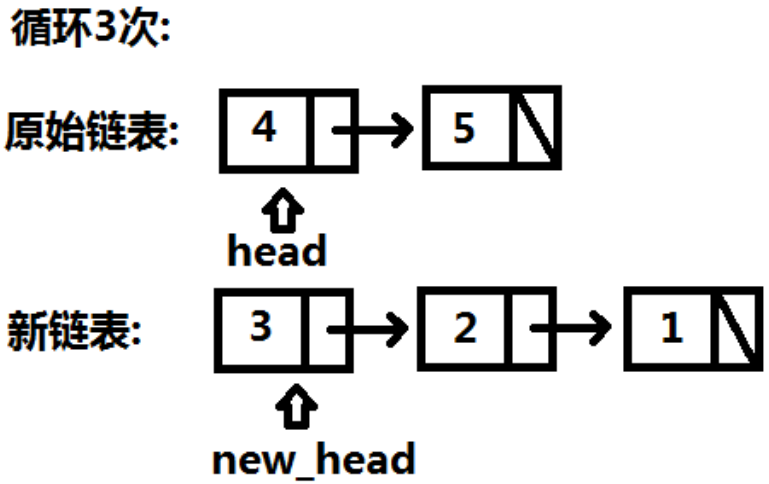
方法1:就地逆置法

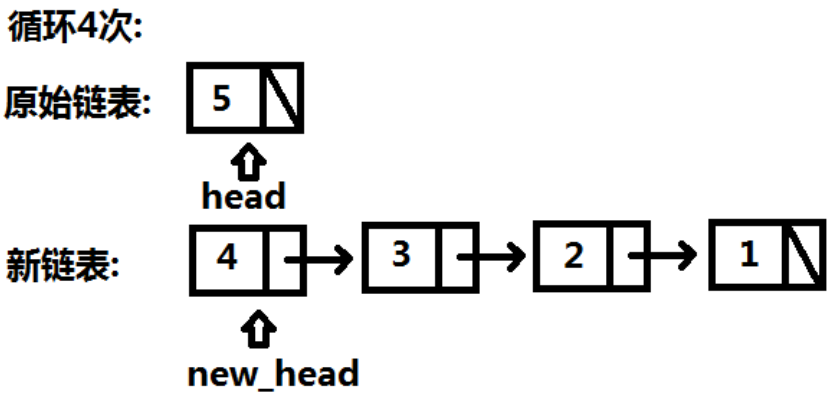
依次遍历链表节点，每遍历一个节点即逆置一个节点

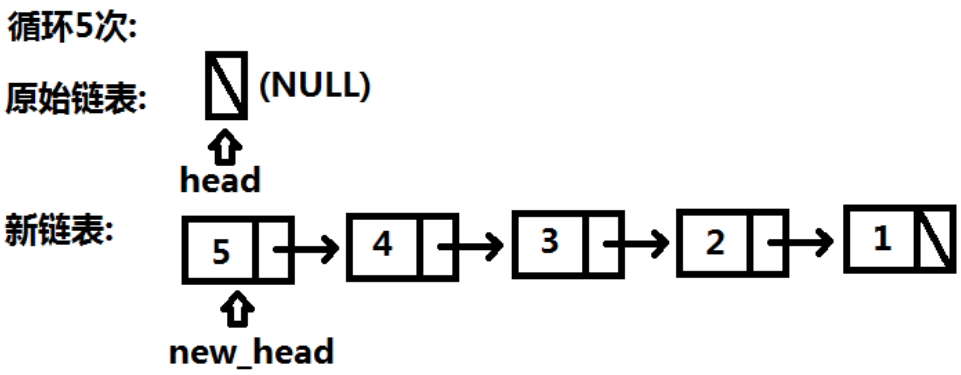




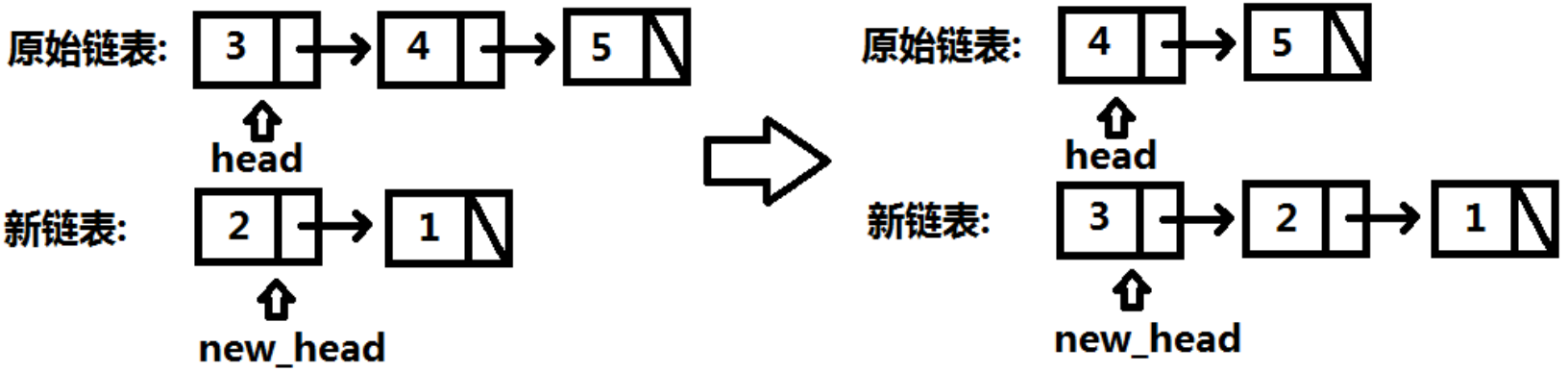


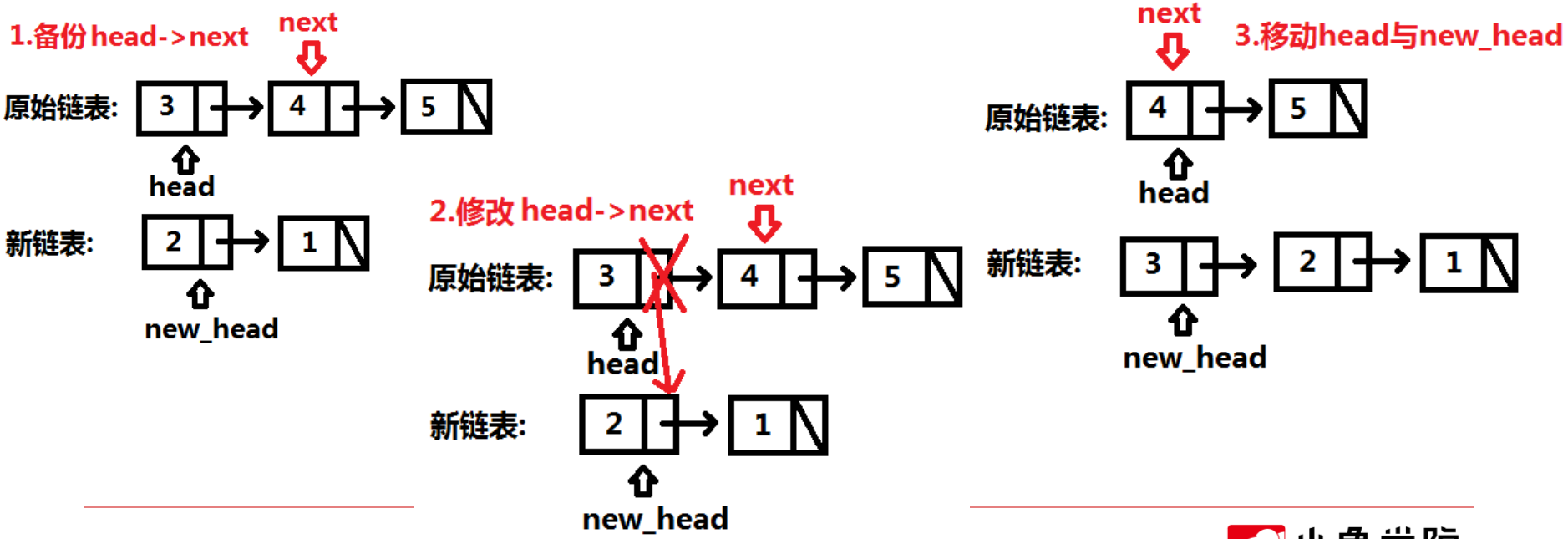






方法1:就地逆置法





就地逆置法，课堂练习

```
struct ListNode {
    int val;
    struct ListNode *next;
};

typedef struct ListNode ListNode;

struct ListNode* reverseList(struct ListNode* head) {
    ListNode *new_head = NULL;
    while( 1 ){
        ListNode *next = 2
        3
        4
        head = next;
    }
    return 5
}
```

3分钟，填写代码，有问题提出！

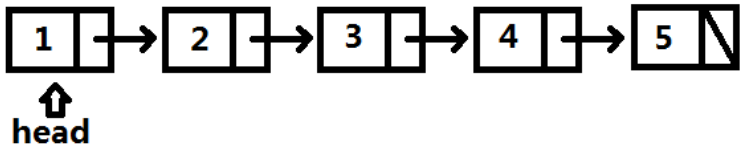
就地逆置法，实现

```
struct ListNode {
    int val;
    struct ListNode *next;
};

typedef struct ListNode ListNode;

struct ListNode* reverseList(struct ListNode* head) {
    ListNode *new_head = NULL;
    while(  head  ){
        ListNode *next =  head->next;
        head->next = new_head;
        new_head = head;
        head = next;
    }
    return  new_head;
}
```

方法2:头插法

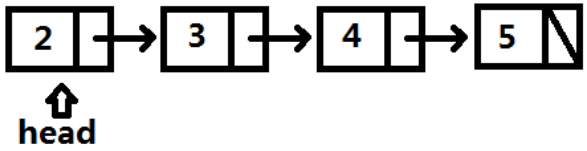
设置一个临时头节点temp_head，利用head指针遍历链表，
每遍历一个节点即将该节点插入到temp_head后。

初始状态，待插入1号节点:



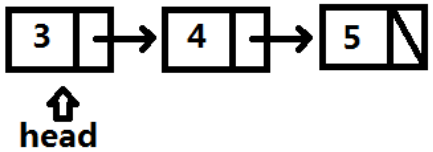
temp_head

完成1号节点插入，待插入2号节点:

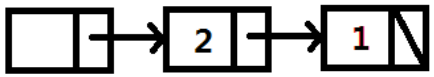


temp_head

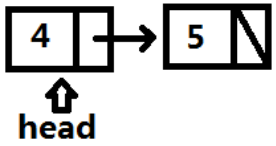
完成2号节点插入，待插入3号节点:



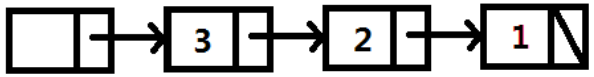
temp_head



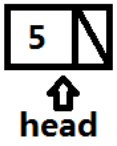
完成3号节点插入，待插入4号节点:



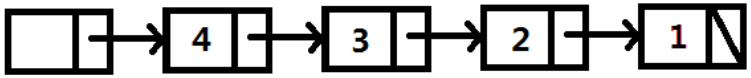
temp_head



完成4号节点插入，待插入5号节点:



temp_head

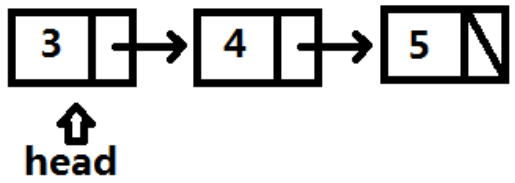


完成5号节点插入，所有节点均完成遍历，head指向了空

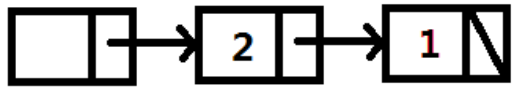
temp_head

方法2:头插法

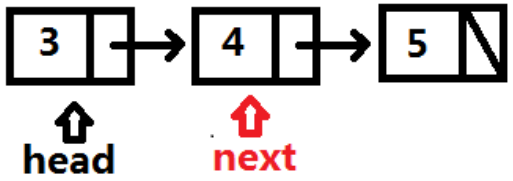
插入head指向的某一节点:



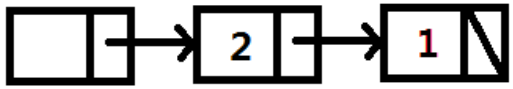
temp_head



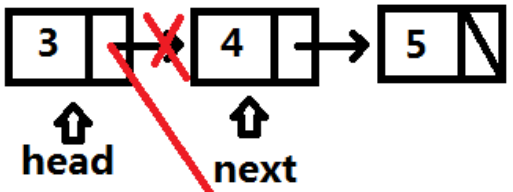
1. 备份 next = head->next;



temp_head

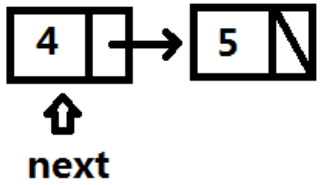


2.修改head->next, head->next = temp_head.next;

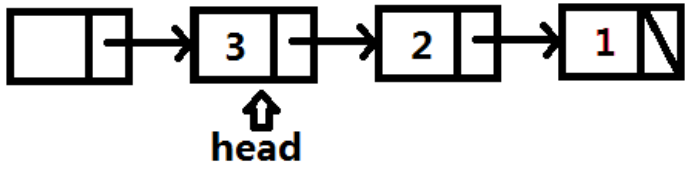


temp_head

3.修改temp_head.next , temp_head.next = head

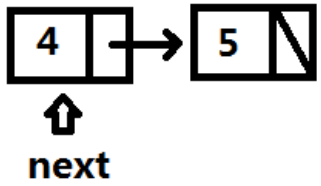


temp_head



4.移动head , head = next;

new_head



temp_head

头插法，课堂练习

```
struct ListNode {
    int val;
    struct ListNode *next;
};

typedef struct ListNode ListNode;

struct ListNode* reverseList(struct ListNode* head) {
    ListNode temp_head;
    temp_head.next = NULL;
    while(head){
        ListNode *next = head->next;
            1
            2
        head = next;
    }
    return
            3
}
```

3分钟，填写代码，有问题提出！

头插法，实现

```
struct ListNode {
    int val;
    struct ListNode *next;
};

typedef struct ListNode ListNode;

struct ListNode* reverseList(struct ListNode* head) {
    ListNode temp_head;
    temp_head.next = NULL;
    while(head){
        ListNode *next = head->next;
        head->next = temp_head.next;
        temp_head.next = head;
        head = next;
    }
    return temp_head.next;
}
```

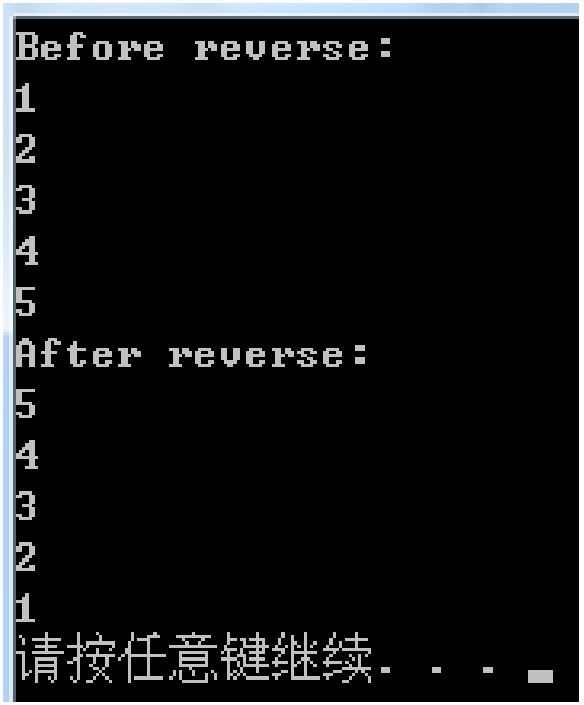
两种逆置链表方法的比较

```
//就地逆置法
struct ListNode* reverseList(struct ListNode* head) {
    ListNode *new_head = NULL;
    while(head){
        ListNode *next = head->next;
        head->next = new_head;
        new_head = head;
        head = next;
    }
    return new_head;
}
```

```
//头插法
struct ListNode* reverseList(struct ListNode* head) {
    ListNode temp_head;
    temp_head.next = NULL;
    while(head){
        ListNode *next = head->next;
        head->next = temp_head.next;
        temp_head.next = head;
        head = next;
    }
    return temp_head.next;
}
```

测试与leetcode提交结果

```
int main(){
    ListNode a;
    ListNode b;
    ListNode c;
    ListNode d;
    ListNode e;
    a.val = 1;
    a.next = &b;
    b.val = 2;
    b.next = &c;
    c.val = 3;
    c.next = &d;
    d.val = 4;
    d.next = &e;
    e.val = 5;
    e.next = 0;
    ListNode *head = &a;
    printf("Before reverse:\n");
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    head = reverseList(&a);
    printf("After reverse:\n");
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    return 0;
}
```



Reverse Linked List

Submission Detail

27 / 27 test cases passed. Status: Accepted

Runtime: **4 ms** Submitted: **0 minutes ago**

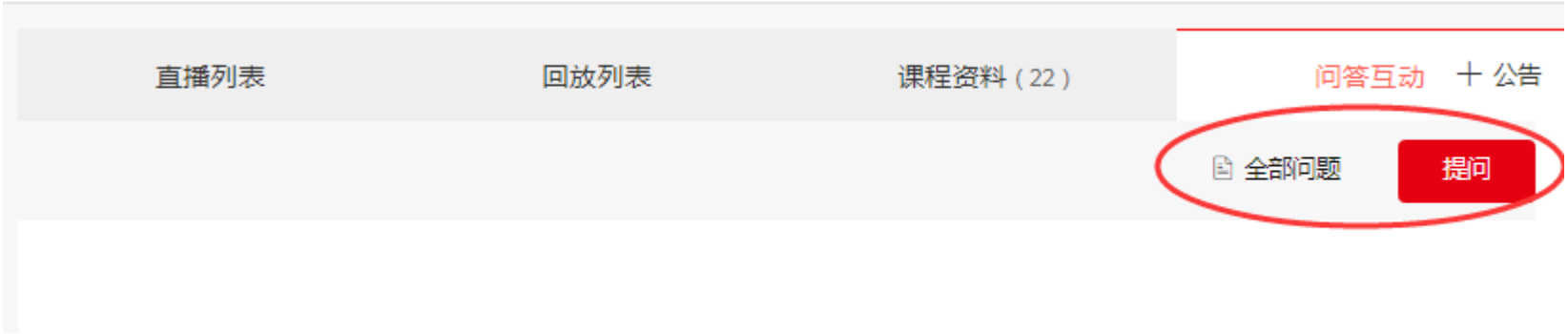
结束

非常感谢大家！

林沐

问答互动

在所报课的课程页面，
1、点击“全部问题”显示本课程所有学员提问的问题。
2、点击“提问”即可向该课程的老师和助教提问问题。



联系我们

小象学院：互联网新技术在线教育领航者

– 微信公众号：**小象学院**