计算机程序设计

Computer Programming

构造数据类型

主讲：吴锋

目录
CONTENTS

◎ 结构体类型

- 考虑一个问题：如何表示一个班级全体学生的各科成绩？
  - 用一个二维数组 float score[50][6];
    - 第一维代表学生序号，第二维代表第几个科目

	科目1	科目2	科目3	科目4	科目5	科目6
学生1	81	85	81	85	81	85
学生2	82	84	82	84	82	84
学生3	83	83	83	83	83	83
学生4	84	82	84	82	84	82
学生5	85	81	85	81	85	81

    - 你还需要一个额外的说明文档，记录学生的顺序，科目的顺序

◎ 结构体类型

- 考虑一个问题：如何表示一个班级全体学生的各科成绩？
  - 用一个二维数组 float score[50][6];
- 如果需要同时记录学生的学号和名字呢？
  - 再增加一个长整型数组 long number[50]; 和一个字符数组 char name[50][16];
- 如果有的科目成绩是A、B、C，而不是百分制呢？
  - 为这些科目再单独设置一个二维数组 char score2[50][3];

- 不同类型的数据不能放在同一个数组中，这使得我们不得不采用多个数组来表达同一个对象的多种不同类型属性数据
  - 程序代码繁琐，可读性差，容易出错，维护困难

◎ 结构体类型

- C语言提供了结构体类型（简称结构类型）
  - 把不同类型变量（成员）放在一个集合中
  - 管理方便，容易维护
  - 结构体内成员变量的名字本身就具备一定的注释功能，可读性好

```
struct Student {
    long num;
    char name[16];
    float math;
    float physics;
    float chemistry;
} score[50];
```

- 结构体类型是一种构造类型，也称用户自定义类型
  - int、double、char等语言已严格定义的，称为基本类型
  - 数组、结构体等由用户自行定义的，称为构造类型

◎ 结构体类型的声明

```
struct 结构类型名 {
    数据类型1 成员1;
    数据类型2 成员2;
    ……
    数据类型n 成员n;
};
```

```
struct Student {
    long num;
    char name[16];
    float math;
    float physics;
};
```

- 包括结构类型名称及其所包含的成员（或称为域）
- struct是关键字，表明其后定义的是结构体类型
- 成员的声明方式同变量定义

◎ 结构体类型的嵌套

- 当成员的类型是已定义过的其它结构体类型时，就形成结构的嵌套

```
struct Date {
    int year;
    int month;
    int day;
};
struct Person {
    long num;
    char name[20];
    float salary;
    struct Date birthday;  //结构的嵌套
};
```

◎ 结构体类型的嵌套

- 结构的成员不能是自身的结构变量

```
struct Student {
    long num;
    float score;
    struct Student st;
};
```

✗

该为它分配多大的存储空间呢？

- 但结构的成员可以是指向自身的结构指针

```
struct Student {
    long num;
    float score;
    struct Student *ps;
};
```

√

结构体变量的声明

- 先声明类型，再用结构类型声明变量

```
struct Student  {
        long num;
        char name[16];
        float math;
        float physics;
};
struct Student stu1, *pstu;
```

- 直接在声明类型的同时声明变量

```
struct Student  {
        long num;
        char name[16];
        float math;
        float physics;
} stu1, *pstu;
```

- 两种声明方式等价
- 可以声明指向结构体类型的指针变量
- 如果在声明结构类型的同时设声明变量，那么可以省略结构类型名
  - 但不建议这么做，因为你可能会在某处再次用到该类型申明

◎ 结构体变量的声明

- 使用类型定义符typedef自定义类型的“别名”
- 一般形式：

  typedef 原类型名 新类型名;

- 如：

```
typedef struct Student {
        long num;
        char name[16];
        float math;
        float physics;
} Stu;
struct Student stu1;
Stu *pstu;
```

- 使用 typedef 定义结构体类，并起别名为 Stu
- 注意struct之后的类型名与typedef定义的别名可以重复
- 可以使用 struct Student 来声明结构体变量
- 也可以使用 Stu 来声明结构体变量（声明时可以省略 struct）
- 用两种方式定义的变量并不完全等价

◎ 结构体成员的引用

- 结构体变量是一组数据的集合，对其中具体成员的访问，称为引用
- 结构成员的引用
  - 结构体普通变量的成员引用格式：
    结构体变量.成员名
    - 例如 stu1.num, stu1.name, stu1.math 等
  - 结构体指针变量的成员引用格式：
    结构体指针名->成员名
    - 例如 pstu->num, pstu->name

```
struct Student  {
        long num;
        char name[16];
        float math;
        float physics;
} stu1, *pstu;
```

◎ 结构体成员的引用

- 嵌套结构的成员引用时要逐层进行访问

  - 例 ps.birthday.year=1990;

  - 例 pps->birthday.year=1990;

```
struct Date {
    int year;
    int month;
    int day;
};

struct Person {
    long num;
    char name[20];
    float salary;
    struct Date birthday;
} ps, *pps;
```

结构体变量的初始化

- 在使用结构体变量前，应对成员赋初值
- 可以在声明结构体变量的同时初始化

```
struct Date {
    int year;
    int month;
    int day;
} d1={2004,11,17}, d2={2014,12,28};
```

- 可以在声明之后，分别给每个成员赋值

```
d1.year=2004;
strcpy(stu1.name, "Li Jun");
```

◎ 结构体变量的初始化

- 可以指定初始化

```
struct Date {
    int year;
    int month;
    int day;
} d1={.month=4, .day=18};
```

  - 没有涉及的成员设为0

结构体变量的初始化

- 若在结构中定义了指针成员，使用该指针变量时要注意其初始化问题

```
struct  Student {
        long no;
        char *name;
        char gender;
        int age;
        float score;
} s3;
```

- 可以s3.name="Cheng Hua";
- 可以struct Student s3={ 20011201, "Cheng hua"... };
- 不能： gets(s3.name);
- 不能： scanf("%s", s3.name);

◎ 结构体变量间的赋值

- 可以将一个有值的结构变量整体赋给另一个同类型的结构变量

```
struct Student d1={1400101,"Liu",'M',19,92}, d2;
d2=d1;
```

- 当结构体作为形参/实参时，采用的是值传递，整个结构体的数据会被复制
  - 值传递效率较低，尽可能传递指针
- 即使结构体中包含数组，也会被整体赋值
  - 数组不能整体赋值，封装在结构体中就可以，可能比自己写循环复制的效率更高
    - DMA高速通道
  - 利用结构体封装，也可以将整个数组作为函数实参进行传递

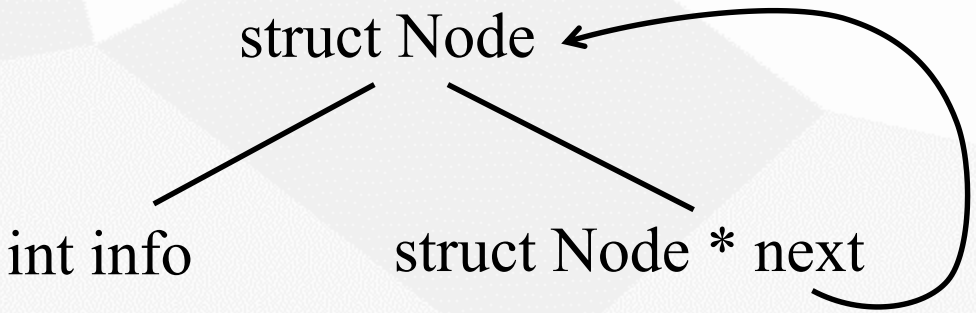
◎ 结构体变量间的赋值

- 什么是同类型？C语言对结构类型采用名字等价，对其他类型采用结构等价

```
int a[10];                      int *fun1(){
struct S1 {int i;};                 return(a);
struct S2 {int i;};             }
struct S1 s;                    struct S2 fun2() {
                                    return(s);
                                }
```

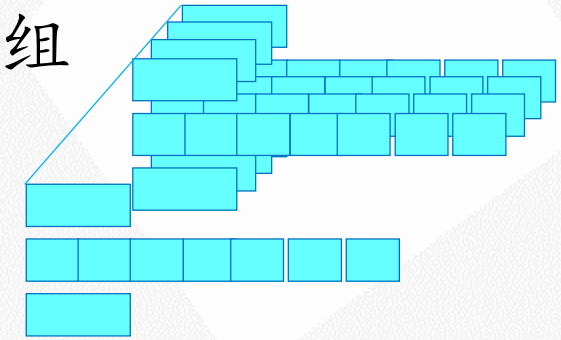
- 为什么要名字等价？

```
struct Node {
    int info;
    struct Node *next;
}
```



◎ 结构体数组

- 结构体类型也可以作为数组元素类型，声明结构体数组



- 例如：

```
struct Person {
    long num;
    char name[20];
    float salary;
};
struct Person ps[50];
```

```
struct Person {
    long num;
    char name[20];
    float salary;
} ps[50];
```

- 常常用于构造复杂表格等数据结构

结构体数组的初始化

- 与一般数组初始化一样
  - 可以在声明数组时就进行初始化

```
struct Person {
      long num;
      char name[20];
      float salary;
} ps[4]={{1003,"Zhang",3536.5},
              {1009,"Li",3874.2},
              {1010,"Chen",4225.6},
              {1021,"Zhou",3392.0}};
```

每个数组元素都是一个结构体，要用{}将它的初值界定起来

- 也可以在声明之后逐个元素进行赋值

```
ps[2].num=1010;
strcpy(ps[2].name,"Chen");
ps[2].salary = 4225.6;
```

◎ 结构体数组成员的引用

- 先访问数组元素，再访问结构体成员
  - 数组元素.结构体成员名

    ps[12].num

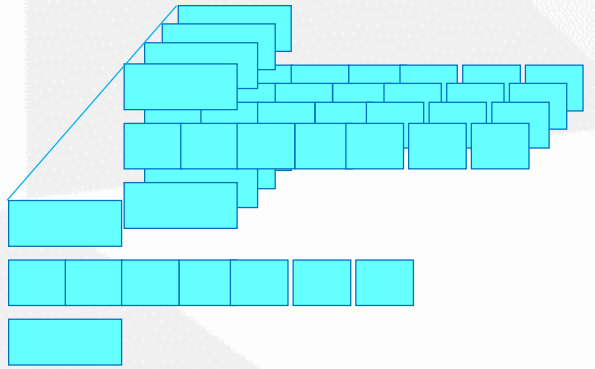
  - (*指针形式访问数组元素).结构体成员名

    (*(ps+23)).salary=3865.9;

  - 数组元素指针->结构体成员名

    (pps+23)->num=1401024;

```
struct Person {
        long num;
        char name[20];
        float salary;
} ps[50], *pps=ps;
```



◎ 结构体数组的大小

- 在不同的系统里，即使各基本类型所占内存大小一致，下面两个结构体的 sizeof 值也很可能不一样，为什么？

```
struct _a{
    char c1;
    long i;
    char c2;
} a;
```

sizeof(a)=12

```
struct _b{
    char c1;
    char c2;
    long i;
} b;
```

sizeof(b)=8

sizeof(char)=1
sizeof(long)=4

- 在分配结构体空间的时候，存在地址对齐的要求

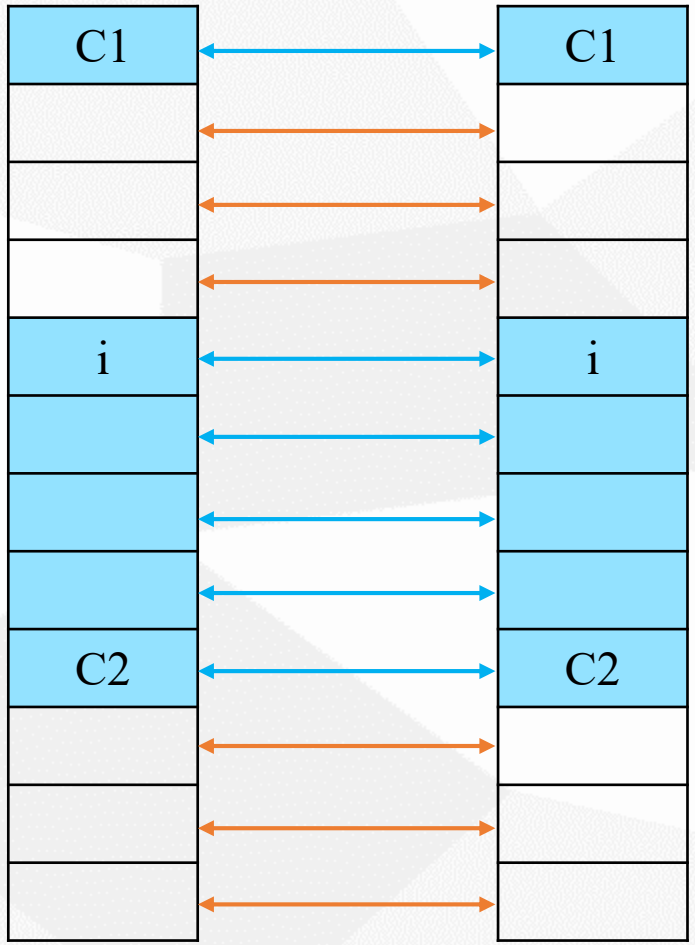
◎ 结构体数组的大小

- 为什么要对齐？
  - 某些平台对特定类型的数据只能从特定地址开始存取
  - 提高内存访问效率，便于CPU的缓存调度
- 结构体字节对齐的细节和具体编译器实现相关（不同的系统可能存在差异），一般而言满足三个准则：
  1. 结构体变量的首地址能够被**实际对齐单位**所整除；
  2. 每个成员相对于结构体首地址的偏移量(offset)都是成员大小或**实际对齐单位**的整数倍，否则会在成员之间加上填充字节(internal adding)；
  3. 结构体的整体大小必须为**实际对齐单位**的整数倍，否则会在最末一个成员之后加上填充字节(trailing padding)。
  - **实际对齐单位**与最宽基本类型成员的大小、CPU参数、预编译指令有关

◎ 结构体数组的大小

- 结构体中“可能的空洞”的存在，决定了结构体变量间不能进行相等或不等的运算
  - 逐个比较成员是极没有效率的
  - 比较全部字节是一个较好的选择，但是空洞使得比较无法进行

```
struct _a{
    char c1;
    long i;
    char c2;
} a;
```



◎ 结构体数组的大小

- 假设对齐要求为 char: 1， long: 4　　sizeof(char)=1, sizeof(long)=4

```
struct _a{
    char c1;
    long i;
    char c2;
} a;
```

0	c1
1	-
2	-
3	-
4	i
5	
6	
7	
8	c2
9	-
10	-
11	-

sizeof(a)=12

```
struct _b{
    char c1;
    char c2;
    long i;
} b;
```

0	c1
1	c2
2	-
3	-
4	i
5	
6	
7	

sizeof(b)=8

◎ 结构体数组的大小

- 对于结构体数组，因地址对齐而导致的内存浪费，值得重视

```
struct Student_info1  {
      char name[10];
      long num;
      char gender;
      int age;
} stu1[10000];
```

```
struct Student_info1  {
      char name[10];
      char gender;
      long num;
      int age;
} stu2[10000];
```

```
sizeof(char)=1
sizeof(int)=4
sizeof(long)=4
char: 1
ing: 4
long: 4
```

- sizeof(struct student_into1)=24; sizeof(stu1)=240KB
- sizeof(struct student_into2)=20; sizeof(stu2)=200KB

◎ 结构体数组的效率

- 考虑下面两种数据结构：

```
struct Student  {
        long num;
        char name[16];
        float math;
        float physics;
} score[10000];
```

```
long num[10000];
char name[10000][16];
float math[10000];
float physics[10000];
```

- 结构体数组更清晰，适宜一个一个学生处理的场合
- 分散的数组适宜批量处理一类数据的场合

◎ 联合体类型

- 有时需要将不同类型变量存储在同一段存储单元中，可以定义联合体类型，也叫共用体类型
  - 任一时刻，只使用其中一个变量
  - 本质是以不同方式解读该段内存的数据
- 联合体的定义与用法类似结构，定义格式

```
union 共用体名 {
    类型1  成员名1;
    类型2  成员名2;
      …… 
    类型n  成员名n;
};
```

```
union ifcd {
    int i;
    float f;
    char c;
    double d;
};
```

◎ 联合体类型

- 联合体的变量定义
  - 先声明联合类类型，后声明变量

    union ifcd x1,x2[5],*px;
  - 或直接在定义联合体类型时声明变量
- 联合体变量的引用
  - 与结构体类型相同，使用 . 及 -> 访问成员
  - 本质上是以不同的类型来读取同一段存储单元
- 联合体与结构体的区别
  - 结构体成员各自占用分配给自己的存储单元，联合体中的成员占用同一个存储单元
  - 联合体变量初始化时，只能也只需对第一个成员赋值

◎ 联合体类型

- 利用联合体类型，可以构造混合的数据结构

```
typedef union {
    int i;
    double d;
} Number;
Number num_array[1000];
```

- 可以混合结构体类型和联合体类型，来为联合添加“标记字段”，以记录联合体包含是什么数据

```
typedef struct {
    int tag;
    union{
        int i;
        double d;
    } u;
} Number;
```

联合体类型

- 使用联合体类型可以提供数据的多个视角
- 例：

```
union int_date {
    long i;  //假定long是8字节
    struct date_time{
        int hour;  //假定int是4字节
        int minute;
    };
};
```

可以方便地将一个长整数与文件日期相互转换

◎ 枚举类型

- 当希望变量的取值离散、有限时，可以定义一个枚举类型
- 枚举类型是基本数据类型，不是构造类型，因为枚举型变量只能取一个值，无法再分解为其他类型
- 枚举类型定义：enum 枚举名 {枚举常量表};

  enum weekday {sun, mon, ... , sat };
- 枚举变量的定义：

  enum weekday d1; 或在定义枚举类型时定义变量
- 枚举变量的取值是枚举常量，实质上是整型序号（从0开始）
- 枚举型变量的值不能通过 scanf 获得，printf 时显示的是其序号

枚举类型

- 可以把枚举常量赋给枚举变量，例9.9

```
#include <stdio.h>
void main(){
    enum weekday{sun, mon, tue, wed, thu, fri, sat} d1, d2, d3;
    d1=sun;
    d2=mon;
    d3=tue;
    printf("%4d,%4d,%4d\n",d1,d2,d3);
}
```

- 如需把数值赋给枚举变量，必须强制类型转换
  - 如 d3=(enum weekday) 2;

◎ 枚举类型

- 枚举声明时，可以给枚举常量指定整数值
  - 例，enum levels={low=100, medium=500, high=2000};

- 枚举变量可以进行++运算
  - 例，enum weekday{sun, mon, tue, wed, thu, fri, sat} d1;

    d1=sun;

    d1++;
  - 但该特性不适用于C++

◎ 动态数组

- 实际应用中，往往无法预先确定所需数组的大小
  - 例如，统计一本书中各单词出现的次数
  - 解决方法1，申请足够大的数组——浪费，有bug
  - 解决方法2，数组大小动态可变

- 动态数组：大小可变的数组
  - C99标准，支持可变长数组（在数组声明时使用表达式来声明数组大小），但并不是真正意义上的动态数组

```
int n;
scanf("%d", &n);
int a[n];
```

◎ 动态分配空间

- 动态分配空间，是在运行时动态申请一段内存空间，并通过指针的类型转换将其视为一个特定对象来使用
  - 常用于创建动态数组、数据结构的结点

- 申请空间：内存分配函数

```
#include <stdlib.h>
void *malloc(unsigned int size);
```

unsigned int的大小是四字节，最大可申请2^{32}字节/4GB空间

  - 分配成功返回指向空类型void的指针，失败返回NULL

- 常用调用方法（类型转换、大小计算）

```
int *p; p=(int *) malloc(n*sizeof(int));
```

◎ 动态分配空间

- 申请空间：内存分配函数

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

  - 为nmemb个元素的数组分配内存空间，其中每个元素的长度都是size个字节
  - 分配成功返回指向空类型void的指针，并将所有字节初始化为0；失败返回NULL

- 分配空间后，可以用 realloc 调整大小

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

  - 为ptr指向的动态分配内存重新分配为大小为size个字节的空间。新尺寸可能会大于或小于原有尺寸
  - 分配成功返回指向空类型void的指针，失败返回NULL
  - ptr必须是通过malloc或calloc获得指针

◎ 动态分配空间

- 释放空间：动态分配的内存在使用完毕后应及时释放

```
#include <stdlib.h>
void free(void *pb);
```

- 常用调用方法

```
free(p);
```

- 注意：
  - free只能释放malloc分配的内存，不能对随便一个指针都应用free
  - 首地址保存好，否则内存无法释放，将造成内存泄漏

◎ 动态分配空间

- 如果内存释放过早，可能造成悬空引用

```
void main() {
    int *a=(int *)malloc(40);
    ......
    free(a);
    ......
    ...*(a+5)...;
    ......
}
```

✗

```
int *fun() {
    int a=5;
    return(&a);
}
void main() {
    int *b;
    b=fun();
    ...*b...;
    ......
}
```

✗

```
int *fun() {
    int *a=(int *)malloc(40);
    return(a);
}
void main() {
    int *b;
    b=fun();
    ...b[2]...;
    free(b);
}
```

√

- 必须谨慎处理指向malloc分配空间的指针

动态分配空间

- 结构体中的可变长数组
  - 动态数组放在结构体末尾
  - 例如：

```
typedef struct {
        int a;
        char buf[0];    // 或者char buf[];
} Node;
......
Node *p = (Node *)malloc(sizeof(Node) + 16);
......
...... p->buf[5] ......
......
free(p);
```

sizeof(Node)=4

◎ malloc 建立动态数组

- 建立一维数组
  - 定义指向所需数据类型的指针变量
  - 按照指定的长度和数据类型申请分配存储空间
  - 将动态分配获取的存储区域首地址强制类型转换后赋值给指针变量
  - 例如：

```
int *pb;
if (pb=(int *)malloc(n*sizeof(int)))==NULL) {
    printf("内存不足\n");
    exit(1);
}
```

  - 使用方法与一般指针相同，也可作为一维数组名使用

◎ malloc 建立动态数组

- 建立二维数组：已知数组的第二维大小
  - 定义二维数组相应的指针变量
  - 申请相应的内容空间
  - 首地址强制类型转换

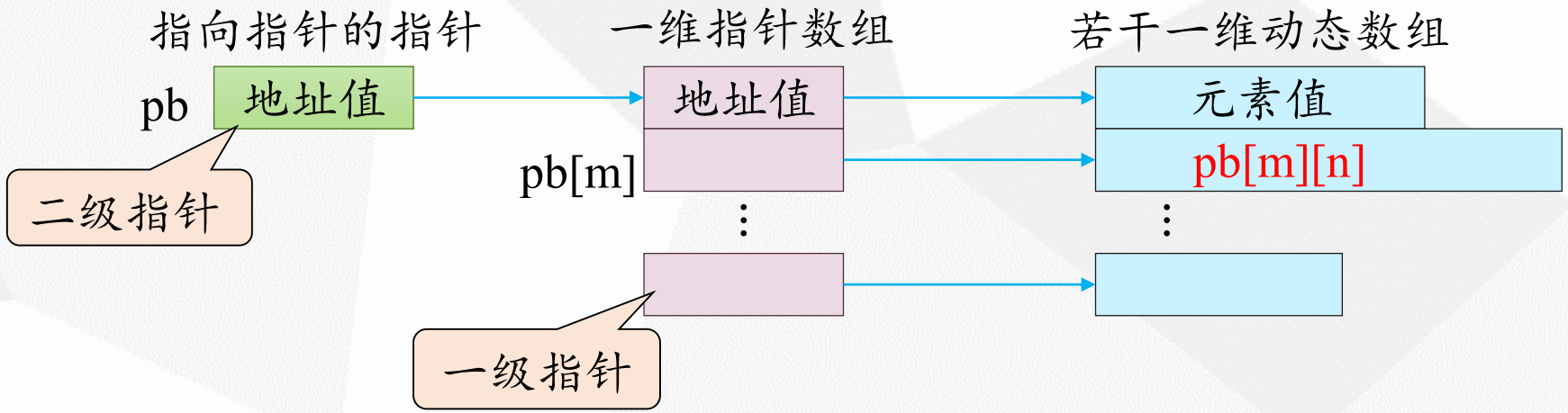
```
int [][10] pb=(int [][10])malloc(5*10*sizeof(int));
```

或

```
int (*pb)[10] = (int (*)[10]) malloc(5*10*sizeof(int));
```

◎ malloc 建立动态数组

- 也可以利用指针数组，构造第二维大小不一致的“数组”
  - 定义指向指向所需数据类型的指针的指针变量（二级指针变量）
  - 以二维数组的行为长度，动态创建一维指针数组，并将其首地址赋值给二级指针变量
  - 以二维数组的列为长度，动态创建若干个（由行数决定）一维数组，并将其首地址分别赋值给指针数组中的对应元素（一级指针）
  - 按二维数组方式或指针方式使用二级指针变量



◎ 内存空间的赋值和拷贝

- 批量赋值函数

```
#include <string.h>
void *memset ( void * ptr, int value, size_t num );
```

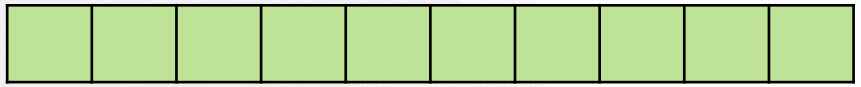
  - 为指针 ptr 所指内存空间的前 num 个字节，赋值为 value
  - value 参数的类型为 int 型，但是执行时系统解析为 unsigned char （0～512）
  - 经常用来为数组清零，如：int a[10]; memset(a, 0, sizeof(a));

- 整块拷贝函数

```
#include <string.h>
void *memcpy ( void * dest, const void * src, size_t num );
```

  - 将指针 src 所指内存空间的前 num 个字节拷贝到指针 dest 所指的内存空间
  - 拷贝是以字节为单位进行的，与数据类型无关，dest 所指的空间需要足够大，避免溢出
  - dest和src的空间不能重叠，如果有重叠需要使用 memmove 函数来完成
  - 经常用于数组的拷贝，如：int a[10], b[10]; memcpy(b, a, sizeof(b));

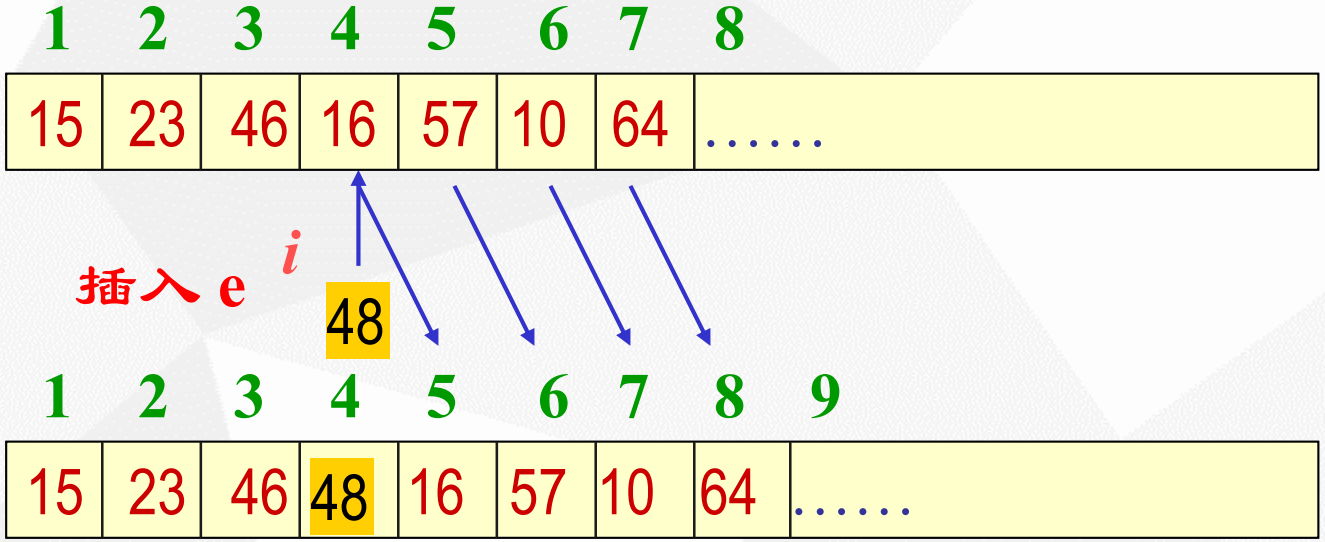
◎ 数组的缺陷



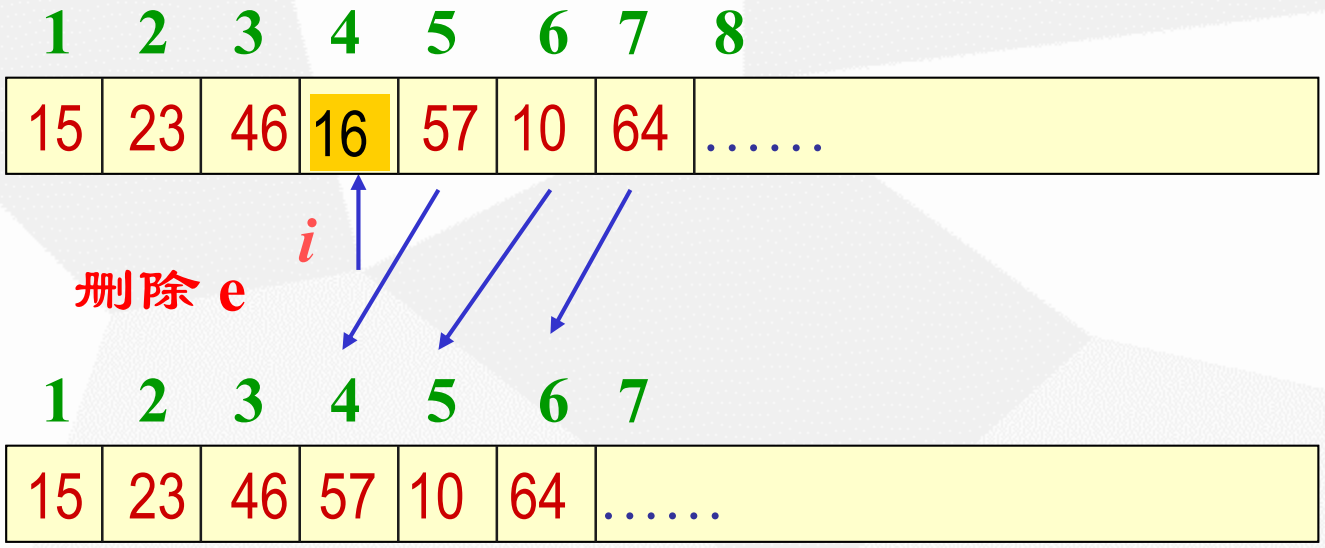
- 常用的数组是静态的数据结构
- 优点：直观、简单、易管理
- 缺点：
  - 当数组的个数不固定时，需要语言的支持
    - 使用malloc等进行动态分配
    - 高版本C语言的功能扩充
  - 当需要随机插入、删除数据时，数组操作麻烦、耗时
    - 大批数据的移动，不宜频繁操作

数组的缺陷

- 特定位置插入元素
  - i 之后的元素向后移



- 特定位置删除元素
  - i 之前的元素向前移



◎ 数组的缺陷

- 时间复杂度分析（以插入操作为例）
  - 频次最高的操作：移动元素
  - 若数组的长度为**n**，则：
    - 最好情况：插入位置**i**为**n+1**，此时无须移动元素，**T(n) = O(1)**
    - 最坏情况：插入位置**i**为**1**，此时须移动n个元素，**T(n) = O(n)**
    - 平均情况：假设p_i为在第i个元素之前插入一个元素的概率，设在数组的各个位置插入是等概率的，则$p_i = 1/(n+1)$，而插入时移动元素的次数为 **n-i+1**，则总的平均移动次数：

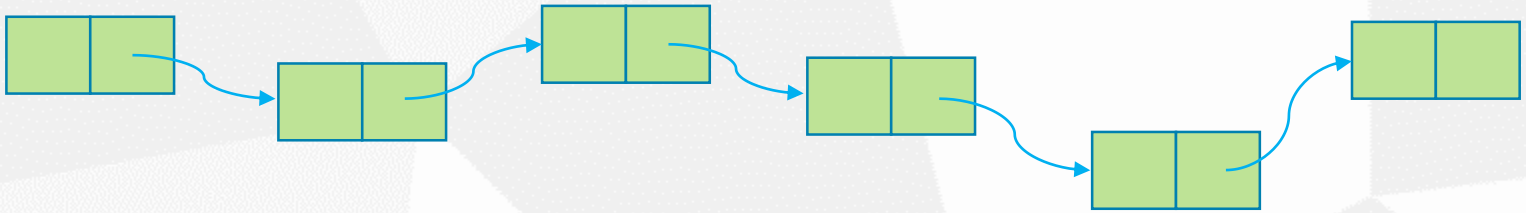
$$E_{insert} = \Sigma_{1 \le i \le n} p_i \times (n-i+1) = n/2$$

即时间复杂度为：**T(n) = O(n)**

- 空间复杂度：S(n) = O(1)

◎ 链表定义

- 链表是一种动态管理的数据结构
  - 用结构体类型定义复杂数据结构的结点
  - 用结构体指针将若干个结点串连在一起
  - 用动态分配空间的方式产生实际数据结点



- 数组的分量称为元素，链表的分量称为结点，一个结点就是一个结构体类型的数据
- 链表可以方便地实现插入、删除操作

◎ 链表定义

- 例：任务链表，将一系列任务连成一个表。其数据结点包括：
  - 任务的内容/索引
  - 指向下一个数据结点的指针

```
struct Node {
    int task_num;
    struct Node *next;
};
```

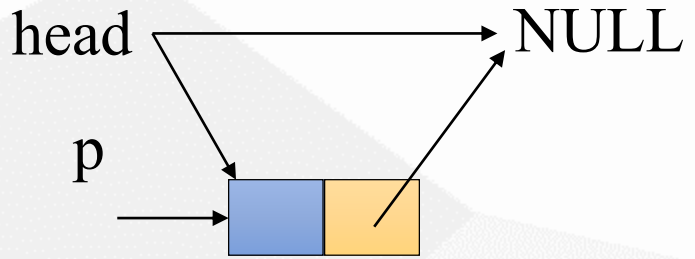
链表的建立

- 链表初始化

```
struct Node *head;
head=NULL;
```

- 建立链表的第一个结点：

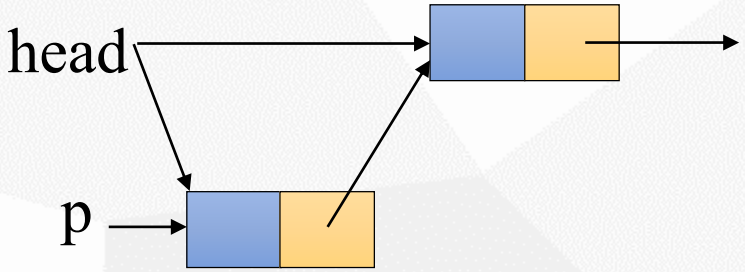
```
struct Node *p;
p=(struct Node *) malloc(sizeof(struct Node));
p->next=head;
head=p;
```



◎ 链表的建立

- 在链表头顺序插入新结点来建立链表（头插法）

```
struct Node *p;
p=(struct Node *) malloc(sizeof(struct Node));
p->next=head;
head=p;
```

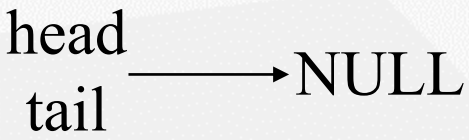


◦ 先插入的在链表尾，后插入的在链表头

链表的建立

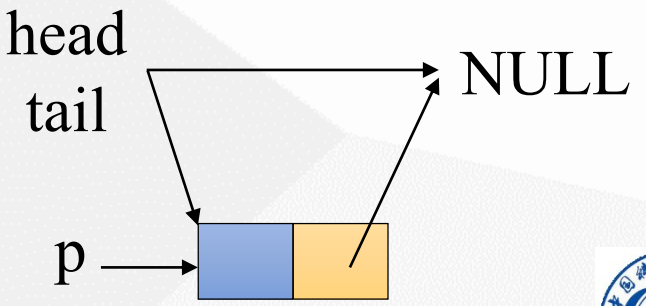
- 在链表尾顺序插入新结点来建立链表（尾插法）
  - 除头指针head外，再增加一个尾指针tail
  - 尾指针tail的初始化

```
struct Node head, *tail;
head=tail=NULL;
```



  - 插入第一个结点

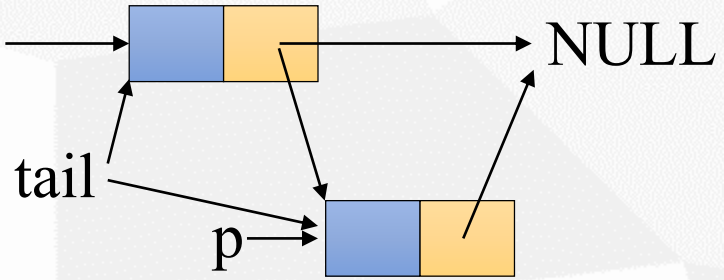
```
struct Node *p;
p=(struct Node *) malloc(sizeof(struct Node));
p->next=NULL;
head=tail=p;
```



链表的建立

- 在链表头顺序插入新结点来建立链表（尾插法）
  - 插入新结点

```
struct Node *p;
p=(struct Node *) malloc(sizeof(struct Node));
p->next=NULL;
tail->next=p;
tail=p;
```



  - 先插入的在链表头，后插入的在链表尾

◎ 遍历链表

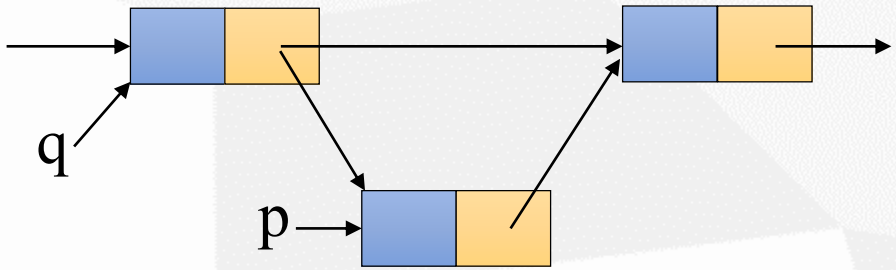
- 用一个指针指向链表头（初始化）
- 将指针指向下一个结点（移动），直至找到目标，或达到链表末尾
- 读取指针所指向结点的内容（访问）

```
struct Node *lookup(struct Node *head, int k)
{    struct Node *q;
     q=head;
     while (q!=NULL && q->num!=k)
          q=q->next;
     if (q==NULL) return(NULL);
     else return(q);
}
```

链表的插入

- 在链表头部插入一个新结点（头插法）
- 在链表任意位置（除链表头外）插入一个新结点

```
struct Node *p, *q;
找到待插入位置q（新结点将插入在q之后）;
p=(struct Node *) malloc(sizeof(struct Node));
p->next=q->next;
q->next=p;
```

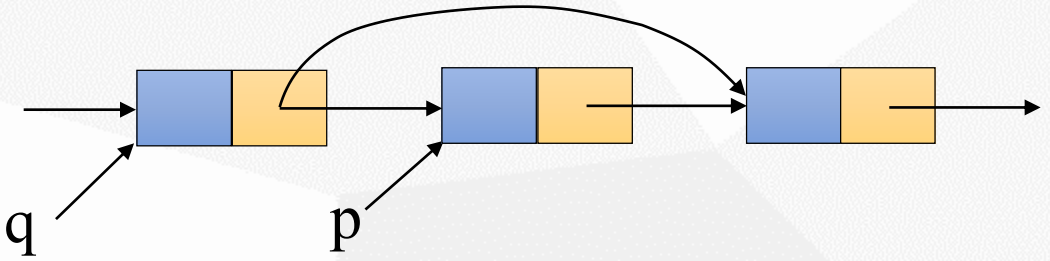


- 处理指针时要注意不要丢失连接：先连接后续结点，再接入前继结点

链表的删除

- 链表结点的删除

```
struct Node *p,*q;
遍历查找链表，找到待删除结点的指针p，以及p的前一个结点指针q;
q->next=p->next;
free(p);
```

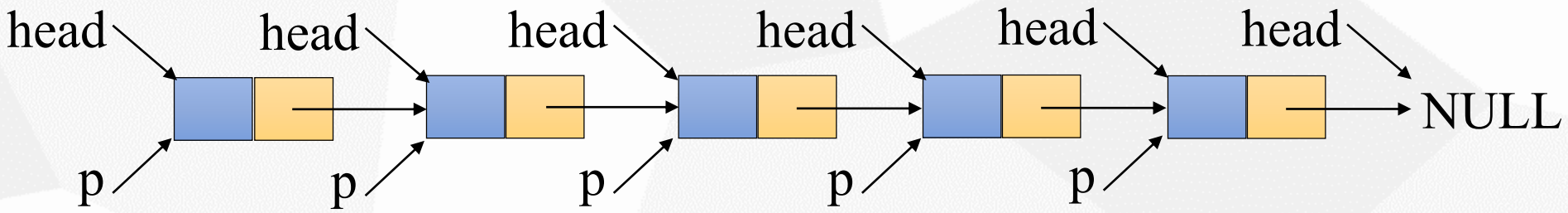


- 删除首结点时，代码应改为head=p->next;

链表的删除

- 整个链表删除
  - 从链表的尾部开始，逐个结点向前删除
  - 从前向后删除

```
struct Node *p;
while (head!=NULL) {
      p=head;
      head=head->next;
      free(p);
}
```

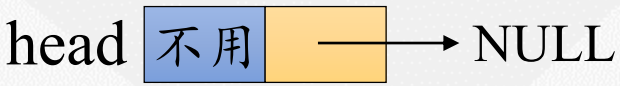


链表的改造: 虚拟首结点

- 在链表的任意位置插入与在链表头插入的过程不一样
  - 一般插入点的前面是结点，表头的前面是head指针
- 虚拟首结点：第一个结点不用

  - 好处：只需调用一般位置插入即可，无需区分是否是首结点插入 q=&head
  - 链表初始化

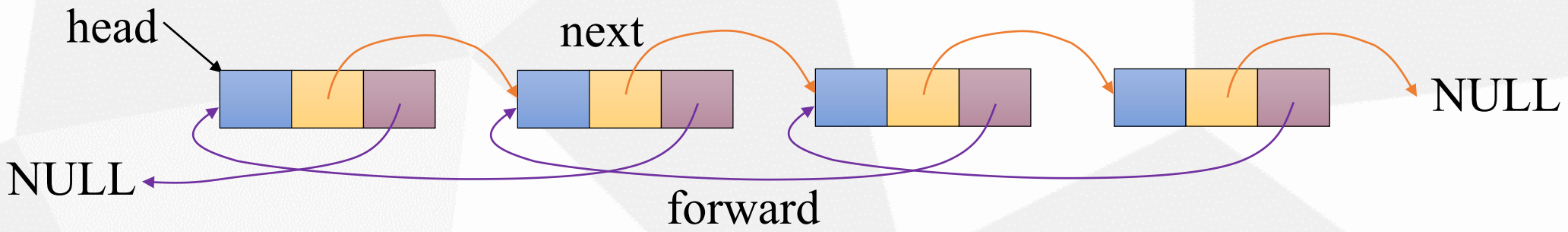
```
struct Node head;
```



链表的改造: 双向链表

- 访问链表时，经常会遇到“求上一个结点”的问题

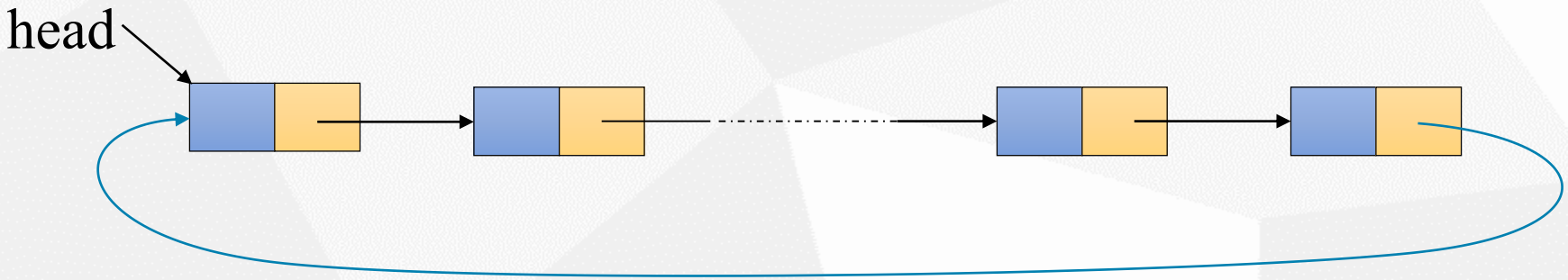
```
struct Node {
    int num;
    struct Node *next, *forward;
};
```



- 优点：节省查询时间；算法更灵活
- 代价：要付出多一倍的指针存储空间

链表的改造: 循环链表

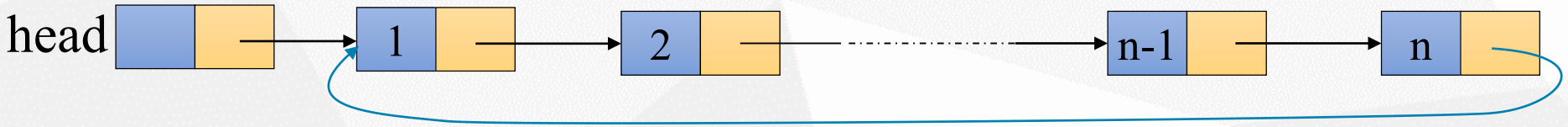
- 将链表尾的指针指回链表开始



- 优点：适用于处理需要循环访问的场景

链表的改造: 循环链表

- 例 (约瑟夫问题)：有n个人，编号为1~n，从第一个人开始从1到m报数，报到m的人出局，然后从第m+1个人开始，重复以上过程。问最后一个人的编号。




```
#include <stdlib.h>
#include <stdio.h>
struct Node {
    int num;
    struct Node *next;
};
```

```
void main() {
    int n=20, m=3, i, j;
    struct Node head, *p, *q;
    p=&head;
    for (i=1; i<=n; i++) {
        p->next=(struct Node *) malloc(sizeof(struct Node));
        p=p->next;
        p->num=i;
    }
    p->next=head.next;
```

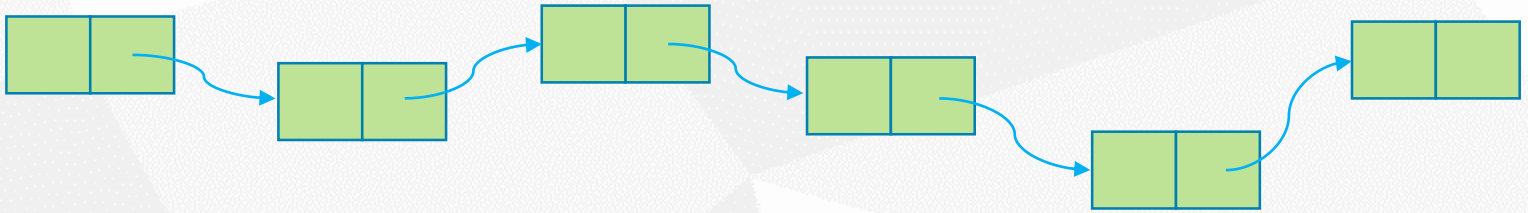
链表的改造: 循环链表

- 例 (约瑟夫问题)：有n个人，编号为1~n，从第一个人开始从1到m报数，报到m的人出局，然后从第m+1个人开始，重复以上过程。问最后一个人的编号。
  - 让p指向当前报数之人，向后移m-2次，用q指向p的后续结点（即出局的人），删除q结点

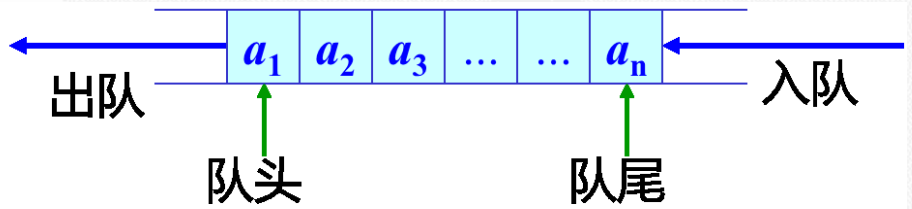
```
    p=head.next;
    for (i=1; i<n; i++) {
        for (j=2; j<=m-1; j++) p=p->next;
        q=p->next;
        p->next=q->next;
        p=q->next;
        free(q);
    }
    printf("%d\n", p->num);
}
```

◎ 构造类型在数据结构中的应用

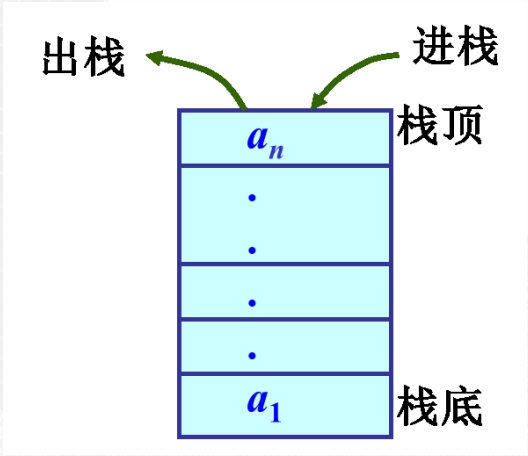
- 链表



- 队列：与链表同样的结构，但约定从头部添加数据（入队），从尾部读取和删除数据（出队）（先入先出）



- 栈：常使用数组而不是用链表，约定从头部添加数据（入栈），从头部读取和删除数据（出栈）（先入后出）



◎ 构造类型在数据结构中的应用

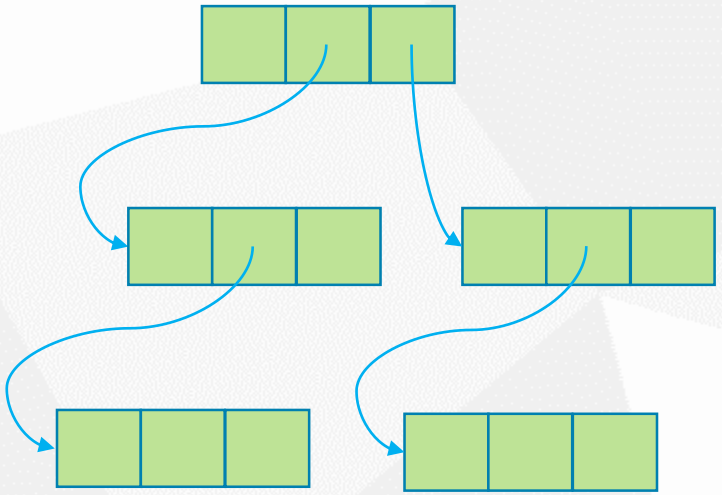
- 栈与递归的实现：调用函数与被调用函数
  - 执行被调用函数前
    - 现场保护：实在参数、返回地址
    - 为被调用函数的局部变量分配存储区
    - 将控制转移到被调函数的入口
  - 从被调用函数返回调用函数前
    - 保存被调函数的计算结果
    - 释放被调函数的数据区
    - 现场恢复：返回
  - 借助系统工作栈来实现

◎ 构造类型在数据结构中的应用

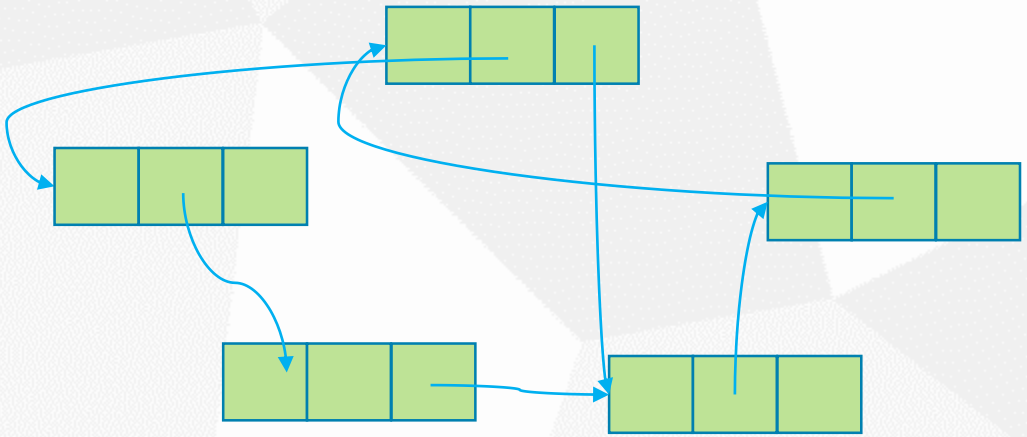
- 递归实现：递归过程与递归工作栈
  - 实际系统中，一般统一处理递归调用和非递归调用
- 递归工作栈
  - 活动记录 (栈帧 Stack Frame)
    - 实在参数、局部变量、上一层的返回地址
  - 每进入一层递归，产生一个新的工作记录入栈
  - 每退出一层递归，就从栈顶弹出一个工作记录
  - 当前执行层的工作记录必是栈顶记录

◎ 构造类型在数据结构中的应用

- 树：



- 图：结点结构与树类似，但结点间的连接更复杂



- 根据问题的不同，设计相应的算法和数据结构

结构体数组举例

- 例：编写程序，定义一个结构体数组来存放学生数据（包括姓名、学号、C语言成绩），从键盘读入数据，按C语言成绩从高到低排序，按上述排序输出学生姓名、学号及成绩。

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

typedef struct {
        char name[10];
        long num;
        int score;
} STU;

#define MAXNUM 100
```

```
int myread(STU *pstu);  //返回值为人数
void mysort(STU *pstu, int n);  //n表示人数
void mywrite(STU *pstu, int n); //n表示人数

void main() {
        int n;
        STU sstu[MAXNUM];
        n=myread(sstu);
        mysort(sstu, n);
        mywrite(sstu, n);
}
```

结构体数组举例

- 例：编写程序，定义一个结构体数组来存放学生数据（包括姓名、学号、C语言成绩），从键盘读入数据，按C语言成绩从高到低排序，按上述排序输出学生姓名、学号及成绩。

```
int myread(STU *pstu) {
    int n=0;
    while (strlen(gets(pstu->name))!=0) {
        scanf("%ld",&(pstu->num));
        scanf("%f",&(pstu->score));
        n++;
        pstu++;
        getchar();
    }
    return(n);
}
```

```
void mysort(STU *pstu, int n) {
    STU temp;
    int i,j;
    for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++)
        if ((pstu+i)->score < (pstu+j)->score) {
            temp=*(pstu+i);
            *(pstu+i)=*(pstu+j);
            *(pstu+j)=temp;
        }
}
```

结构体数组举例

- 例：编写程序，定义一个结构体数组来存放学生数据（包括姓名、学号、C语言成绩），从键盘读入数据，按C语言成绩从高到低排序，按上述排序输出学生姓名、学号及成绩。

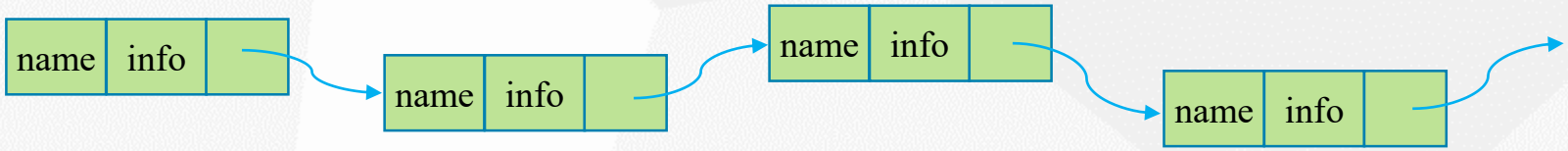
```
void mywrite(STU *pstu, int n) {
        int i;
        for (i=0; i<n; i++,pstu++)
                printf("%s ( %ld ): %.1f\n",pstu->name, pstu->num, pstu->score);
}
```

复杂数据结构举例

- 为一个图书馆的藏书建立一个按书名检索的系统。
  - 建立一个结构体数组

```
struct book_list{
        char name[101];
        char info[1001];
} books[1000];
```

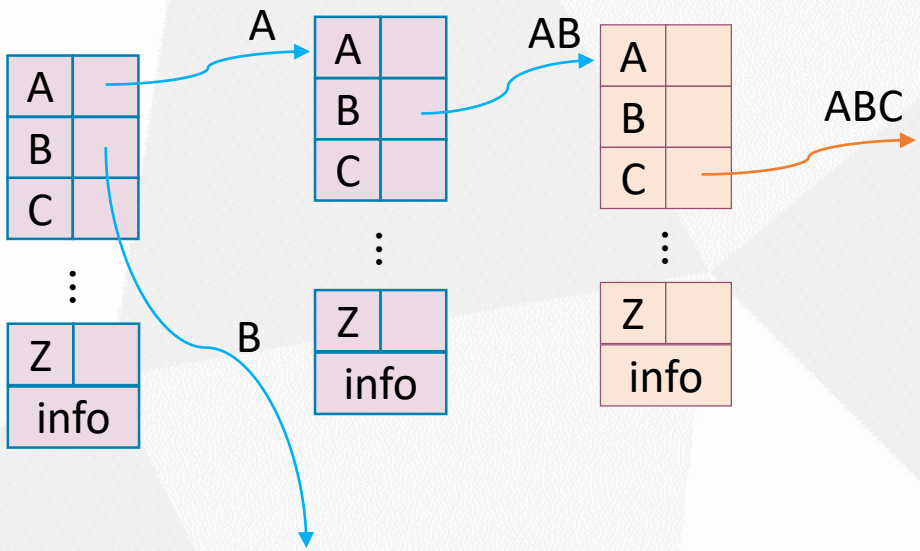
    - 需要设定书册上限
    - 配合快速查找算法、插入和删除操作
  - 建立一个书籍的链表



    - 查找困难

复杂数据结构举例

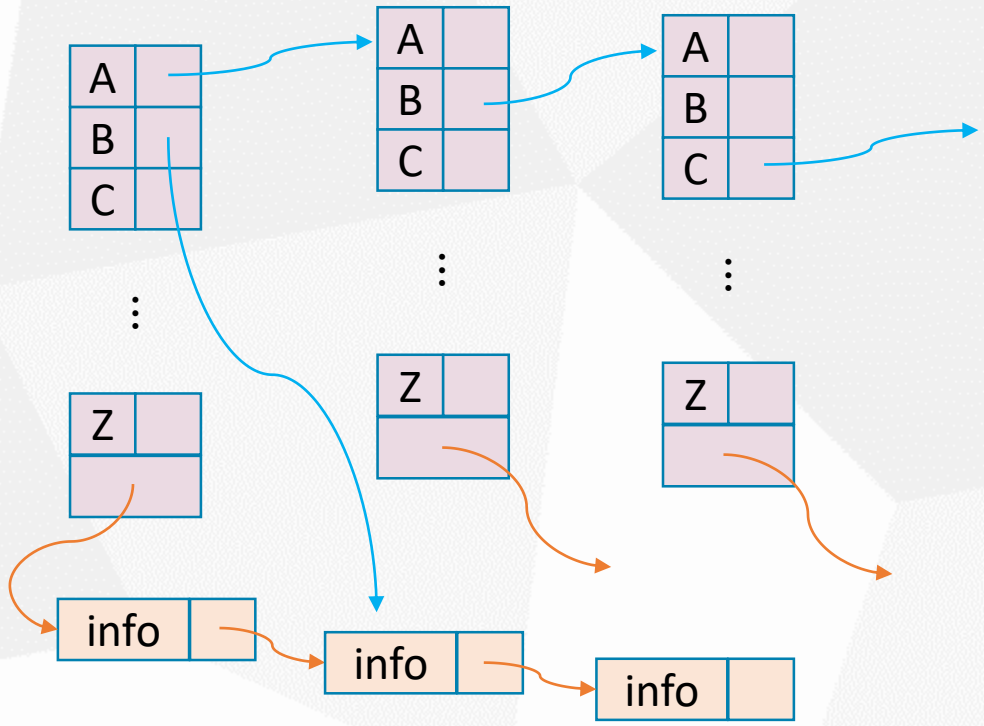
- 为一个图书馆的藏书建立一个按书名检索的系统。
  - 建立一个书目的数据结构




```
struct book_tree{
        char info[1000];
        struct book_tree *next[26];
};
struct book_tree *head;
……
char * lookup(struct book_tree *p, char *bookname) {
        int c;
        if (*bookname==‘\0’) return(p->info);
        c=*bookname-’A’;
        if (p->next[c]==NULL) return(error);
        lookup(p->next[*bookname-’A’], bookname+1);
};
……
lookup(head, "ABC");
```

复杂数据结构举例

- 为一个图书馆的藏书建立一个按书名检索的系统。
  - 建立一个书目的数据结构
  - 重名书籍的处理：将info部分扩展为一个链表



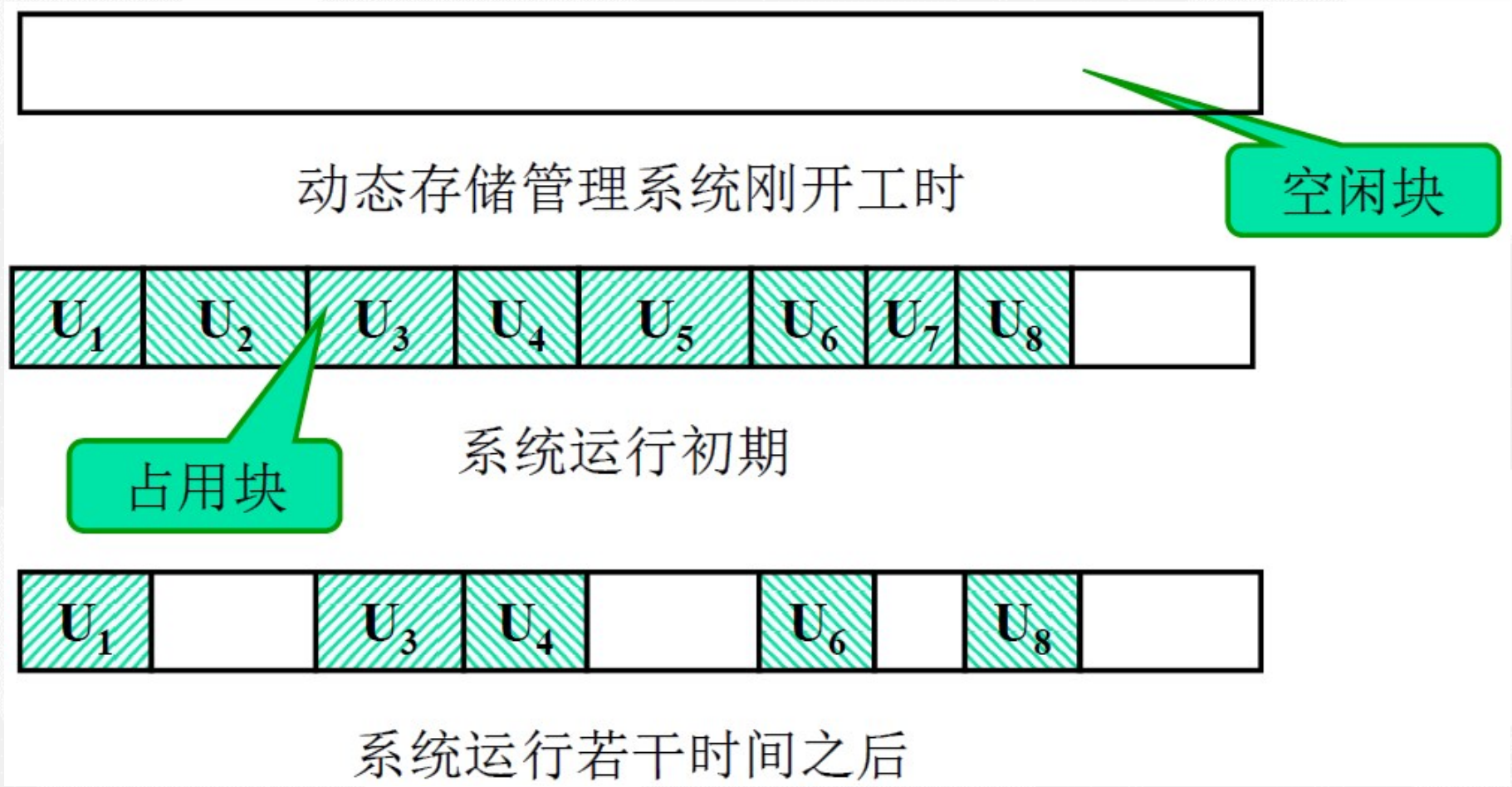

```
struct book_info{
        char info[1000];
        struct book_info *next;
}

struct book_tree{
        struct *book_info;
        struct book_info *next[26];
};
```

◎ 动态存储管理原理 *

- 动态存储管理的基本问题
  - 系统如何根据用户提出的“请求”分配内存？
  - 如何回收那些用户不再使用而“释放”的内存，以备新的“请求”产生时重新进行分配？
  - 用户提出的“请求”：
    - 可能是进入系统的一个作业
    - 也可能是程序执行过程中的一个动态变量
  - 系统每次分配给用户都是一个地址连续的内存区。
    - 已分配给用户使用的地址连续的内存区称为“占用块”
    - 未曾分配的地址连续的内存区称为“可利用空间块”或“空闲块”

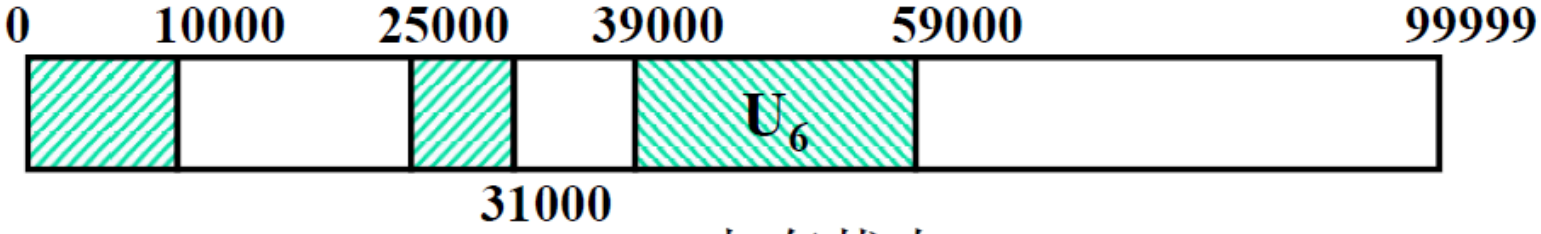
◎ 动态存储管理原理 *


空闲块
动态存储管理系统刚开工时
U1
U2
U3
U4
U5
U6
U7
U8
占用块
系统运行初期
U1
U3
U4
U6
U8
系统运行若干时间之后

◎ 动态存储管理原理 *

- 当有新的用户进入系统请求分配内存，系统将如何做呢？
  - 策略一：从高地址的空闲块中进行分配，当分配无法进行时，系统回收所有用户不再使用的空闲块，并重新组织内存，将所有空闲的内存区连接在一起成为一个大的空闲块
  - 策略二：从所有空闲块中找出一个“合适”的空闲块分配之。系统需要建立一张记录所有空闲块的“可利用空间表”，它可以是“目录表”，也可以是“链表”

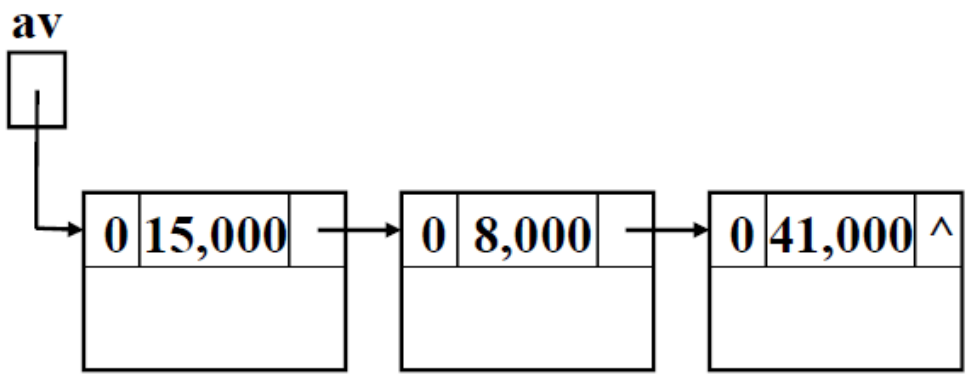
◎ 动态存储管理原理 *



(a) 内存状态

起始地址	内存块大小	使用情况
10,000	15,000	空闲
31,000	8,000	空闲
59,000	41,000	空闲

(b) 目录表



(c) 链表

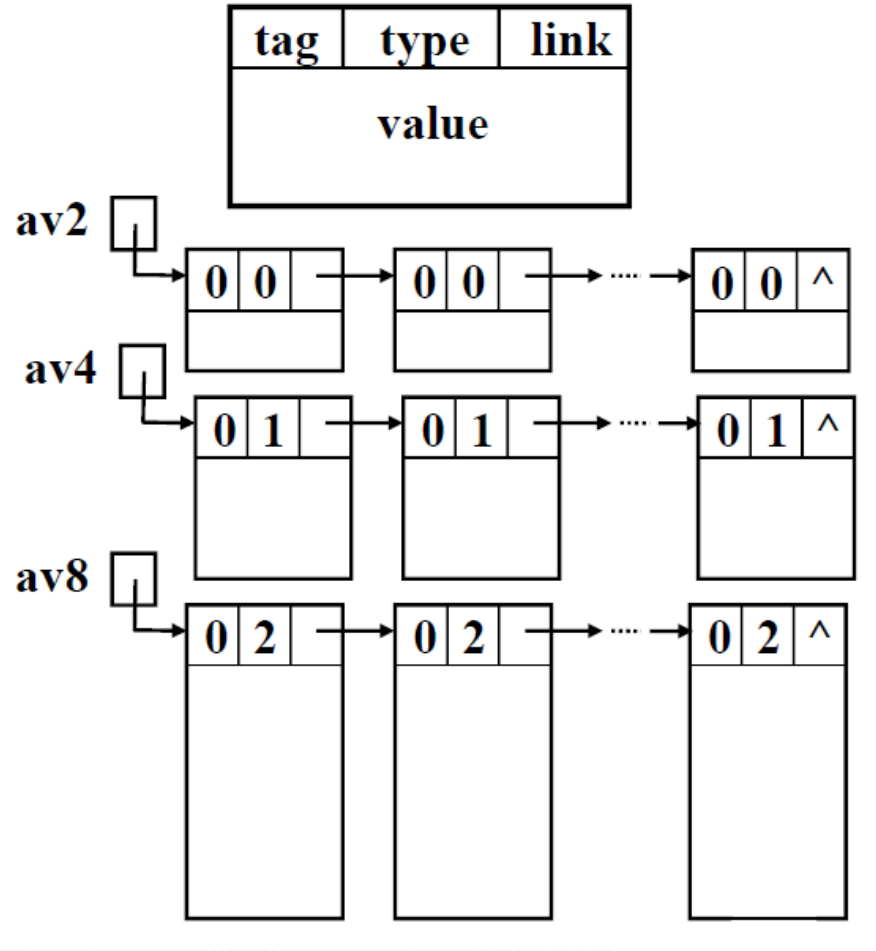
这里仅就链表的情况进行讨论

◎ 动态存储管理原理 *

- 利用可利用空间表管理内存：
  - 建立链表，一个空闲块用一个结点来表示
  - 用户请求分配时，系统从表中删除一个结点分配之
  - 用户释放所占内存时，系统即回收并将它插入到表中
- 最简单的可利用空间表的结构：
  - 系统运行期间所有用户请求分配的存储量大小相同
  - 由于表中结点大小相同，则在分配内存时无需查找

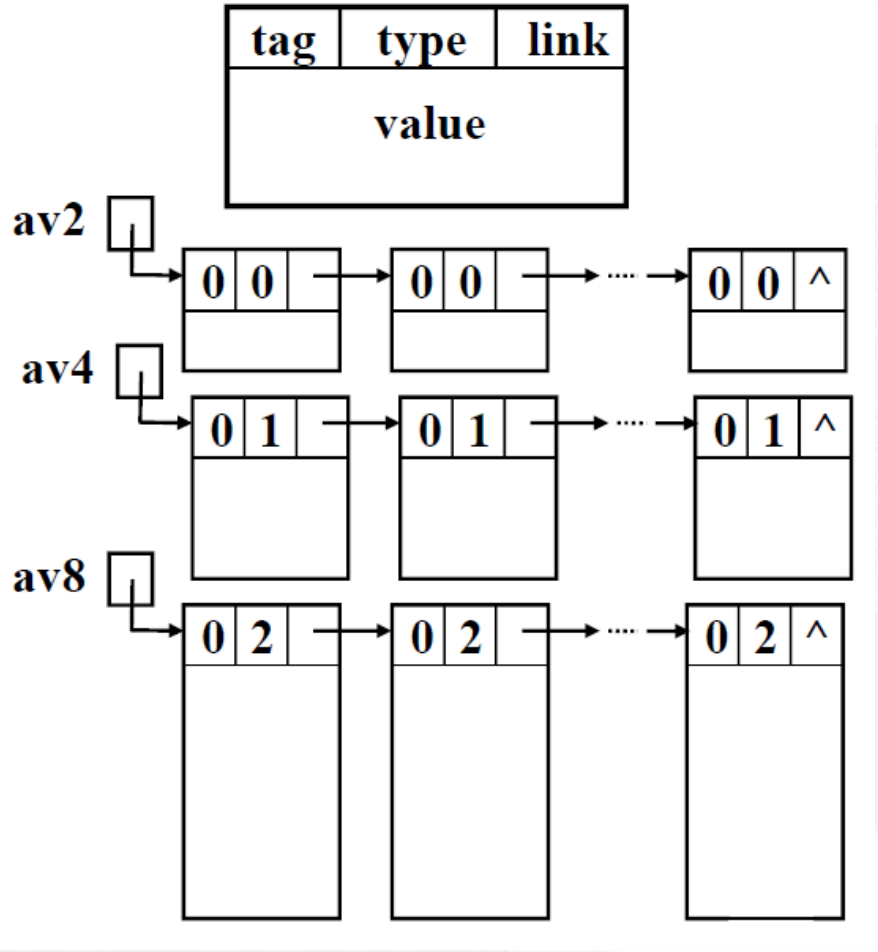
◎ 动态存储管理原理 *

- 系统运行期间用户请求分配的存储量有若干种大小的规格
  - 建立若干个可利用空间表，同一链表中的结点大小相同
  - 每个结点的第一个字设有
    - 链域(link)：指向同一链表中下一结点的指针
    - 标志域(tag)：0-空闲块、1-占用块
    - 结点类型域(type)：区分大小不同的结点



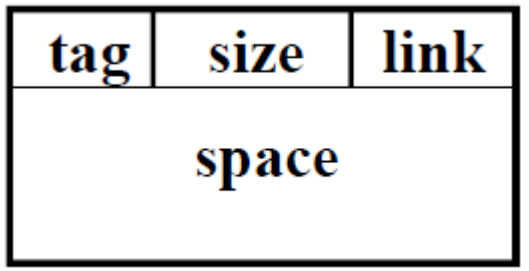
◎ 动态存储管理原理 *

- 系统运行期间用户请求分配的存储量有若干种大小的规格
  - 分配时，执行：
    - 从结点大小和请求分配的量相同的链表中查找结点并分配之
    - 若没有，则从结点较大的链表中查找结点，将其中一部分分配给用户，剩余的插入到相应大小的链表中
    - 若各链表都没有合适的结点，则要执行“存储紧缩”将小块合并
  - 回收时，将释放的空闲块插入到相应大小的链表的表头



◎ 动态存储管理原理 *

- 系统运行期间分配给用户的内存块的大小不固定
  - 开始时，整个内存空间是一个空闲块；随着分配和回收的进行，可利用空间表中的结点大小和个数也随之而变
  - 若链表中仅有一块大小为 m≥n 的空闲块：将其中大小为n的一部分分配给用户，剩余大小为 m-n 的作为一个结点留在链表中
  - 结点的结构代表操作系统中的可变分区管理
    - 链域(link)：指向同一链表中下一结点的指针
    - 标志域(tag)：0-空闲块、1-占用块
    - 大小域(size)：指示该空闲块的存储量
    - space：地址连续的内存空间

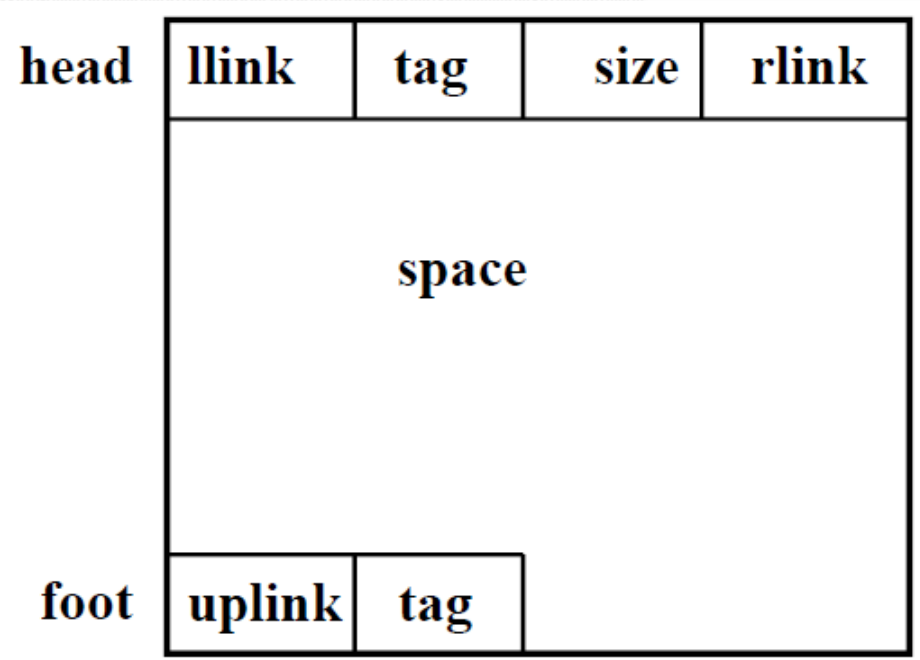


◎ 动态存储管理原理 *

- 可利用空间表的数据结构

```
struct WORD{ // 内存字类型
        union {
                WORD *llink; // 指向前驱结点
                WORD *uplink; // 指向本结点头部
        };
        int tag; // 0-空闲; 1-占用
        int size; // 块大小
        WORD *rlink; //指向后继结点
        OtherType other; //字的其它部分
} head, foot, *space;

// 指向p所指结点的底部
#define FootLoc(p) p+p->size-1
```



◎ 动态存储管理原理 *

- 分配算法（假设请求分配的存储量为n）：
  - 假设找到的待分配空闲块的容量为 m 个字(包括头部)，选定一个适当的常量 e，当 m-n≤e 时，就将容量为m的空闲块整块分配给用户；否则只分配其中 n 个字的内存块。为避免修改指针，约定将该结点中的高地址部分分配给用户
  - 每次分配从不同的结点(如刚进行分配的结点的后继)开始进行查找，使分配后剩余的小块均匀地分布在链表中。避免小块集中在头指针所指结点附近，从而增加查询较大空闲块的时间

◎ 动态存储管理原理 *

- 回收算法（假设用户释放的内存区为M）：
  - 要检查刚释放的占用块 M 的左、右紧邻是否为空闲块
    1. M的左、右邻区均为占用块：简单插入
    2. M的左邻区为空闲块，右邻区为占用块：合并左邻区和M，增加左邻空闲块结点的size域的值，重新设置结点的底部
    3. M的右邻区为空闲块，左邻区为占用块：合并M和右邻区，结点的底部位置不变，头部要变，链表的指针也要变
    4. M的左、右邻区均为空闲块：合并左邻区、M和右邻区，增加左邻空闲块的size值，在链表中删除右邻空闲块结点，修改底部

◎ 动态存储管理原理 *

- 无用单元收集
  - 在存储管理中，用户必须明确给出“请求”和“释放”的信息
  - 因为用户的疏漏或结构本身的原因致使系统在不恰当的时候或没有进行回收而产生“无用单元”或“悬挂访问”的问题
    - 无用单元：那些用户不再使用而系统没有回收的结构和变量

      例： p = malloc(size); …. p=NULL;
    - 悬挂访问：访问已经被释放的结点

      例： p = malloc(size); …. q=p; free(p); a = *q;
  - 对无用单元进行有效的收集十分复杂（如：Java的垃圾回收机制）
    - 策略一：使用访问计数器实时监测内存的使用，但计数器为0时回收
    - 策略二：内存接近用尽时，一次性扫描内存进行回收

作业

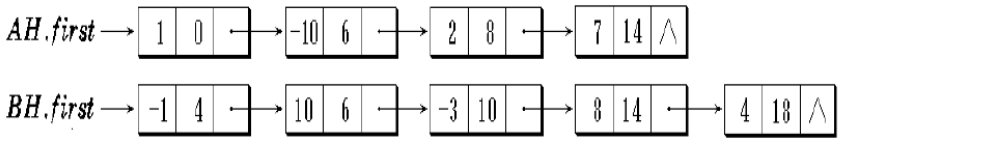
1. 实现课件中介绍的按书名检索的系统
   - 支持插入、删除、查找

2. 编写一个程序，实现两个多项式的加法运算。
   - 提示：用一个有序的链表表示一个多项式，每一项用一个结点表示。在链表中按照项的幂数进行排列，然后对幂数相同的结点进行归并。

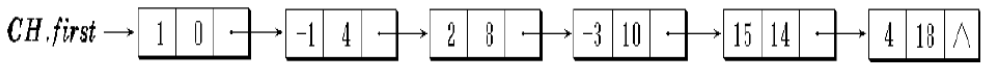
$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$

$$BH = -x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$

$$CH = AH + BH$$



(a) 两个相加的多项式



(b) 相加结果的多项式

作业

3. 最小交换次数：输入具有 N 个不同值的数组 A[ ]，找出将数组排序所需的最小交换次数。
   - 例如：4 3 2 1，需要交换0次。1 5 4 3 2，需要交换2次。
   - 思路：依次从前往后，找出第i小的数放在第i位，记录需要交互次数
   - 注意：数组元素可以是任意类型（包括结构体），排序可以是从小到大或从大到小

4. 迷宫密码：猪无戒将蓝兔关在了一座迷宫里。虹猫来到紧闭的迷宫大门前，发现需要输入密码才能将大门打开。守门人告诉他密码就隐藏在门旁的一个方形木盘中。方形木盘中有 M 行 N 列格子，每个格子中刻有一个整数，虹猫需要选择一个数字之和最大的长方形区域来开启迷宫大门。虹猫应该怎么做呢？
   - 输入矩阵，输出最大长方形的左上角、右下角坐标，以及该长方形中的数字之和。
   - 例如：输入下面矩阵，输出（2，3）（4，4）23

	1	2	3	4
1	-1	-2	-3	-4
2	-3	-2	2	4
3	-3	-4	3	5
4	4	-5	3	6
5	-3	-2	-1	0