# 第14讲 动态数据结构

## 教材11.4，12.8节

## MOOC第13-14周

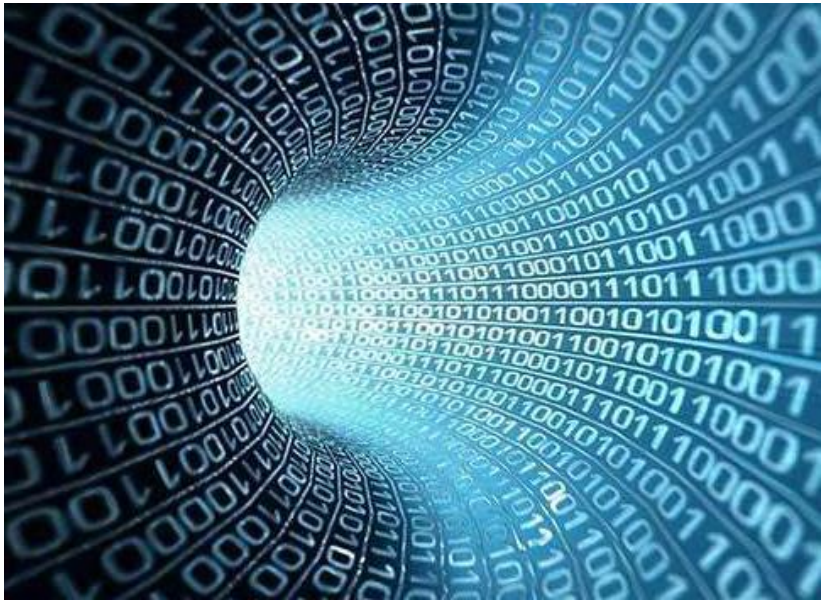
哈尔滨工业大学

苏小红

sxh@hit.edu.cn

# 小明的故事

- 小明是一个程序猿，他为公司写了一个数据处理的程序，这些数据将从输入设备读入到一个数组中。小明想定义一个大一点的数组，肯定能放下所有数据。
- 于是他申请了10000个大小的数组。结果平时每次一批数据也就几百个。10000个单元只用了几百个，小明想有点小浪费啊！
- 突然有一天“大数据”来了，1000个单元也不够用了。



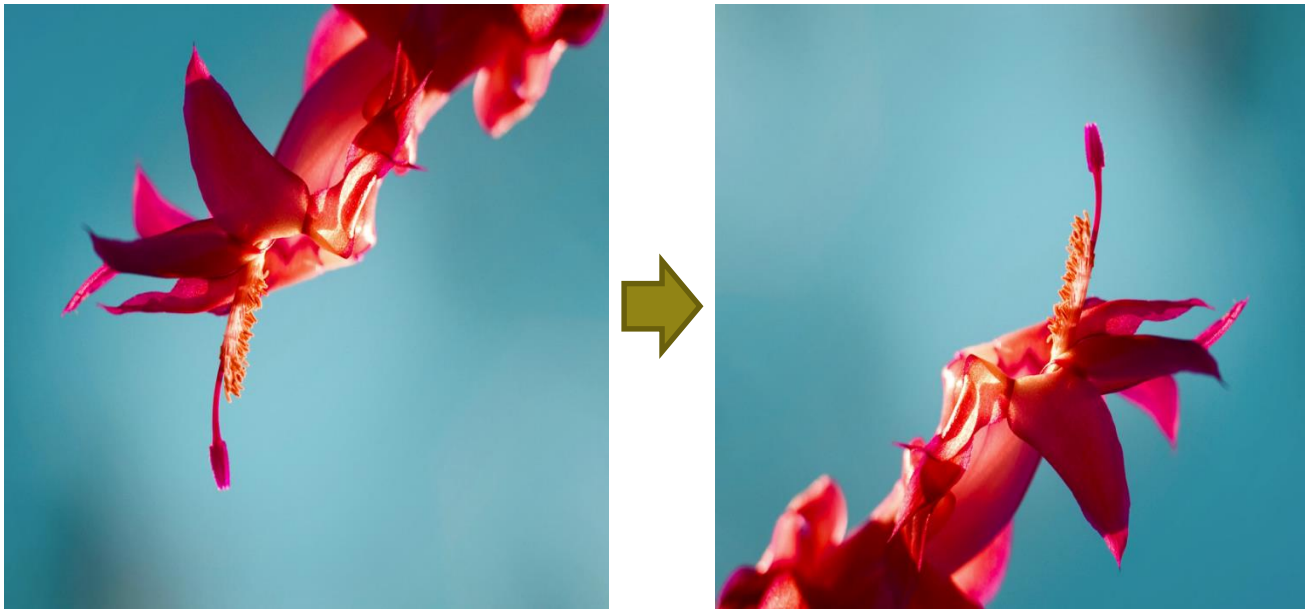
# 图像矩阵转置

- 计算机中的图像可用一个矩阵来表示
  - 假设图像矩阵中的数都是不超过1000的非负整数
  - 那么，n行m列的图像矩阵转置需要开多大的数组？
  - 假设1≤n,m≤1000
    - 样例输入

```
3 3
1 5 3
0 2 4
7 8 9
```

    - 样例输出

```
1 0 7
5 2 8
3 4 9
```



# 矩阵转置—定长数组

```
/* 函数功能：计算m*n矩阵a的转置矩阵at */
void Transpose(int a[][N], int at[][M], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            at[j][i] = a[i][j];
        }
    }
}
```

栈空间溢出

```
#include <stdio.h>
#define M 1000
#define N 1000
void Transpose(int a[][N], int at[][M], int m, int n);
void InputMatrix(int a[][N], int m, int n);
void PrintMatrix(int at[][M], int n, int m);
int main()
{
    int s[M][N], st[N][M], m, n;
    printf("Input m, n:");
    scanf("%d,%d", &m, &n);
    InputMatrix(s, m, n);
    Transpose(s, st, m, n);
    printf("The transposed matrix is:\n");
    PrintMatrix(st, n,  m);
    return 0;
}
```

# 矩阵转置—变长数组（C99）

```
/* 函数功能：计算m*n矩阵a的转置矩阵at */
void Transpose(int m, int n, int a[m][n], int at[n][m])
{
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            at[j][i] = a[i][j];
        }
    }
}
```

m,n必须在数组前面声明

允许数组的维度是表达式，在分配时计算
在程序运行时确定
一旦确定不能改变

```
#include <stdio.h>
void Transpose(int m, int n, int a[m][n], int at[n][m]);
void InputMatrix(int m, int n, int a[m][n]);
void PrintMatrix(int m, int n, int at[n][m]);
int main()
{
    int m, n;
    printf("Input m, n:");
    scanf("%d,%d", &m, &n);
    int s[m][n], st[n][m];
    InputMatrix(m, n, s);
    Transpose(m, n, s, st);
    printf("The transposed matrix is:\n");
    PrintMatrix(n, m, st);
    return 0;
}
```

# 当你不能预知数据有多大时

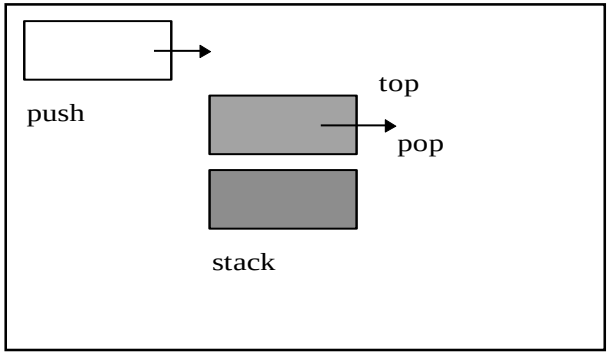
- 有木有提高空间效率的办法呢？
- 动态数组
  - 在程序运行时确定
  - 需要时动态申请资源，不需要时释放资源

# 动态数组需使用的函数

## 动态分配内存

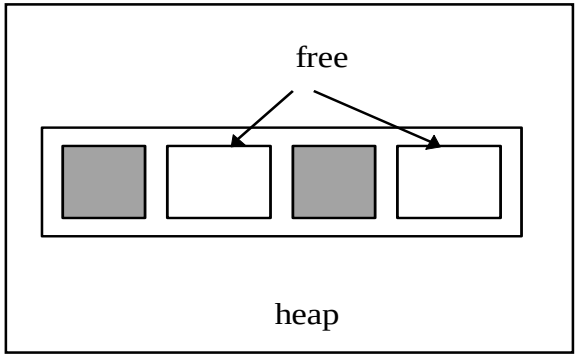
```
void* malloc(unsigned int size);
```

函数参数、局部变量等在栈上分配

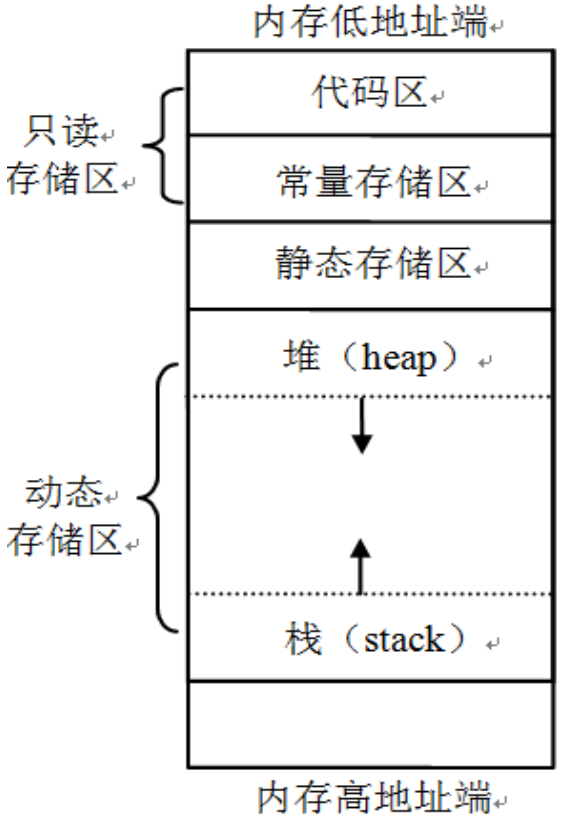



栈：后进先出
顺序访问
向低地址扩展

程序运行期间用动态内存分配函数来申请的内存，从堆上分配




堆：顺序随意
向高地址扩展
随机访问

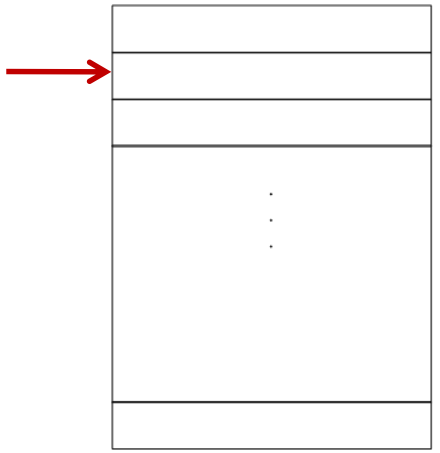



对于malloc的内存申请，系统找到一块未占用的内存，将其标记为已占用

# void*型指针

```
void* malloc(unsigned int size);
```

向系统申请size个字节的连续内存块，把首地址返回



- void*型指针
  - 不指定其指向的变量的类型，可指向任意类型的变量
  - 是一种generic（通用）或typeless（无类型）的指针
- 使用时，需强转(Type*)为其他类型
- 为什么malloc不指定返回指针的基类型？

molloc不能事先确定这些内存块中放什么类型的数据

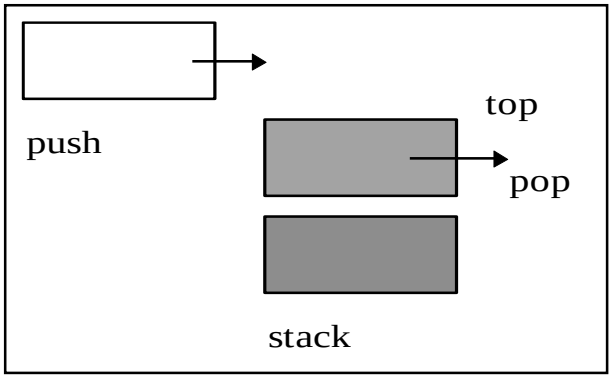
```
p = (float *)malloc(n * sizeof(float));
```

灵活性　　可移植性

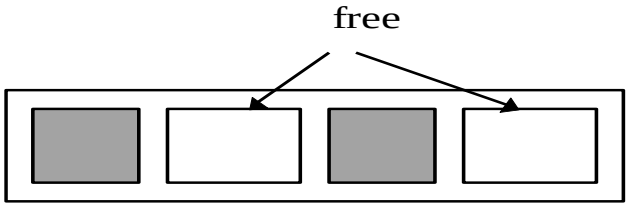
```
free(p); //释放p指向的内存
```

# 栈 vs 堆

- 在栈上分配的内存
  - 函数调用时自动分配
  - 函数返回时自动释放
- 优点
  - 编译系统自动分配释放，无需程序员管理



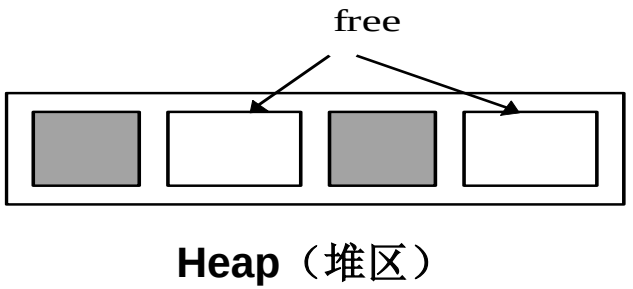

- 在堆上分配的内存
  - 一般由程序员申请和释放
  - 程序结束时由OS回收
- 缺点
  - 频繁申请/释放，速度慢，易造成内存碎片
  - 程序员不释放→内存泄漏（Memory Leakage）
  - 释放了仍继续使用→野指针




Heap（堆区）

# 空指针 vs 无类型指针

```
int *p;
p = (int *) malloc(n * sizeof (int));
if (p == NULL) //判断内存申请是否成功
{
    printf("No enough memory!\n");
    exit(0);
}
```

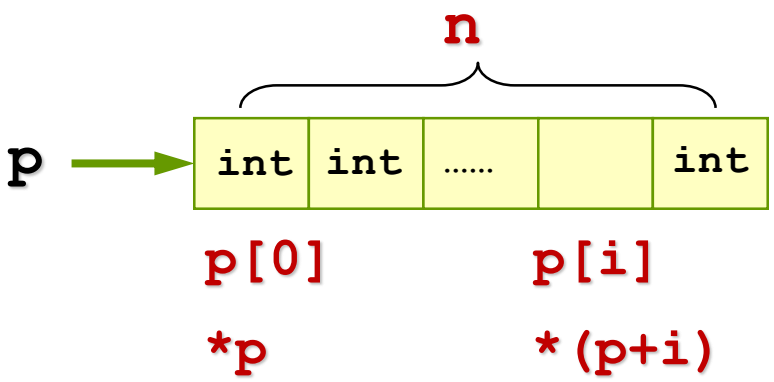



若申请不成功则返回NULL

- 空指针的用途
  - 定义指针时进行初始化
  - 在程序中常用作状态比较
- 指针不能与非指针类型变量进行比较
  - 但可与NULL（即0值）进行相等或不等的关系运算

# 像使用一维数组一样使用一维动态数组

```
int *p = NULL;
printf("Input n:");
scanf("%d", &n); //n的值在运行时确定
p = (int *)malloc(n*sizeof(int));
if (p == NULL)
{
   printf("No enough memory!\n");
   exit(0);
}
Input(p, n);
printf("%d\n", Aver(p, n));
```

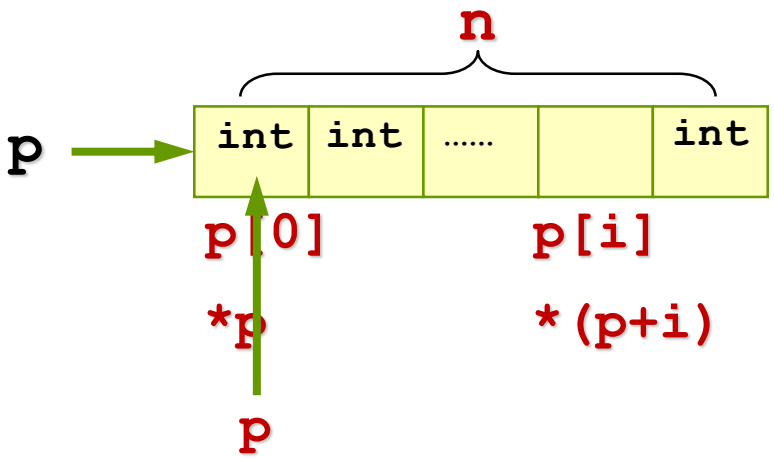



```
void Input(int *p, int n)
{
   int i;
   for (i=0; i<n; i++)
   {
       scanf("%d", &p[i]);
   }
}
```

```
double Aver(int *p, int n)
{
   int i, sum = 0;
   for (i=0; i<n; i++)
   {
        sum = sum + p[i];
   }
   return n>0 ? (double)sum/n : 0;
}
```

利用指针和动态内存分配函数实现动态数组

# 利用指针提高动态数组的访问效率




```
void Input(int *p, int n)
{
    int i;
    for (i=0; i<n; i++, p++)
    {
        scanf("%d", p);
    }
}
```

如何理解 p++？ p++加的字节数取决于p的基类型

```
double Aver(int *p, int n)
{
    int i, sum = 0;
    for (i=0; i<n; i++)
    {
        sum = sum + p[i];
    }
    return n>0 ? (double)sum/n : 0;
}
```

```
double Aver(int *p, int n)
{
    int i, sum = 0;
    for (i=0; i<n; i++,p++)
    {
        sum = sum + *p;
    }
    return n>0 ? (double)sum/n : 0;
}
```

# 矩阵转置——二维动态数组

```
#include <stdio.h>
#include <stdlib.h>
#define M 1000
#define N 1000
void Transpose(short a[][N], short *at, short m, short n);
void InputMatrix(short a[][N], short m, short n);
void PrintMatrix(short *at, short n, short m);
int main()
{
    short s[M][N], *st, m, n;
    printf("Input m, n:");
    scanf("%hd,%hd", &m, &n);
    st = (short*)malloc(m * n * sizeof(short));
    if (st == NULL)
    {
        printf("Out of memory\n");
        exit(0);
    }
    InputMatrix(s, m, n);
    Transpose(s, st, m, n);
    printf("The transposed matrix is:\n");
    PrintMatrix(st, n,  m);
    return 0;
}
```

# 矩阵转置——二维动态数组

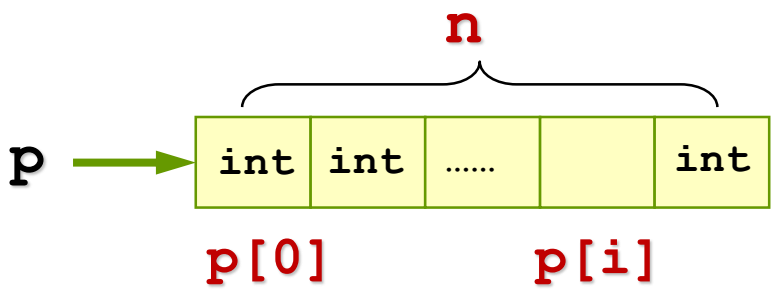
```
/* 函数功能：计算m*n矩阵a的转置矩阵at */
void Transpose(short a[][N], short *at, short m, short n)
{
    short i, j;
    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            at[j*n+i] = a[i][j];
        }
    }
}
```

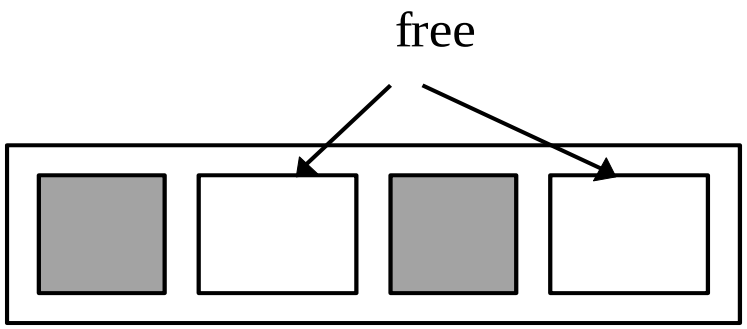
把at当一维数组使用
指向数组中的每一个元素

```
/* 函数功能：输出n*m矩阵at的值 */
void PrintMatrix(short *at, short n, short m)
{
    short i, j;
    for (i=0; i<n; i++)
    {
        for (j=0; j<m; j++)
        {
            printf("%6hd", at[i*n+j]);
        }
        printf("\n");
    }
}
```

# 动态数组的存储结构

- 仍属于顺序存储



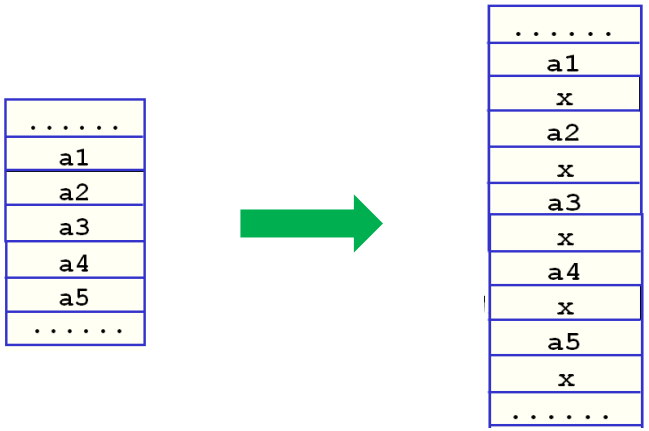

Heap（堆区）

# 生活中的顺序存储

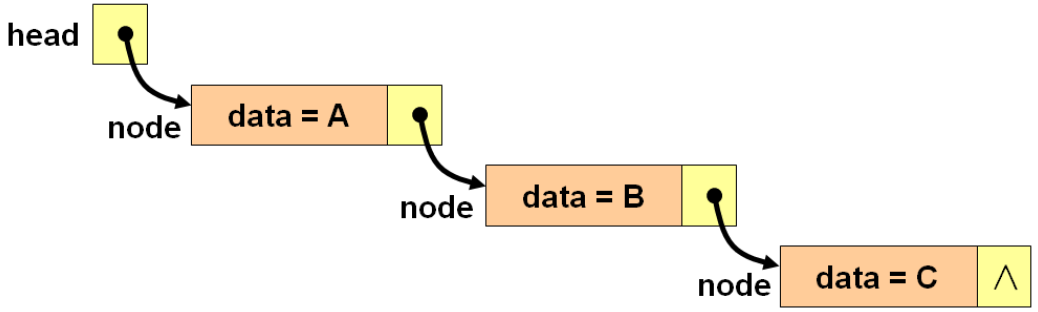
- 排队购票，医院挂号
  - 有人想要上厕所怎么办？
  - 对经常变动的结构，用顺序存储科学吗？
  - 有木有更科学、人性化的管理方式呢？
- 现在到银行办理业务
  - 排队机：领号→等着叫号→…
  - 只要关注前一个号是否被叫到即可
  - 放弃，VIP号
- 链式存储—动态数据结构

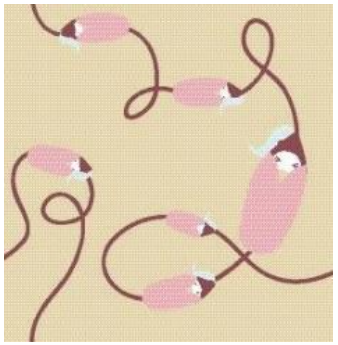
# 从顺序存储到链式存储

- 如果不要求线性表中的数据连续存储，则自然可以想到这样的分散存储：




- 分散存储仅解决了存储问题，不能表达元素之间的线性（顺序）关系
  - 需要用指针建立元素之间的线性关系，指针指向一个元素的后继在内存中的存放位置
  - 像绳子一样把离散的数据串起来



# 用什么样的结构描述每个节点的信息？

```
struct link
{
    int           data;
    struct link   next;
};
```

不能包含本结构体类型的成员
系统无法预知占据多少内存

- CB下的错误提示：
  - field 'next' has incomplete type
- VC下的错误提示：
  - 'next' uses undefined struct 'link'

```
struct link
{
    int           data;
    struct link   *next;
};
```

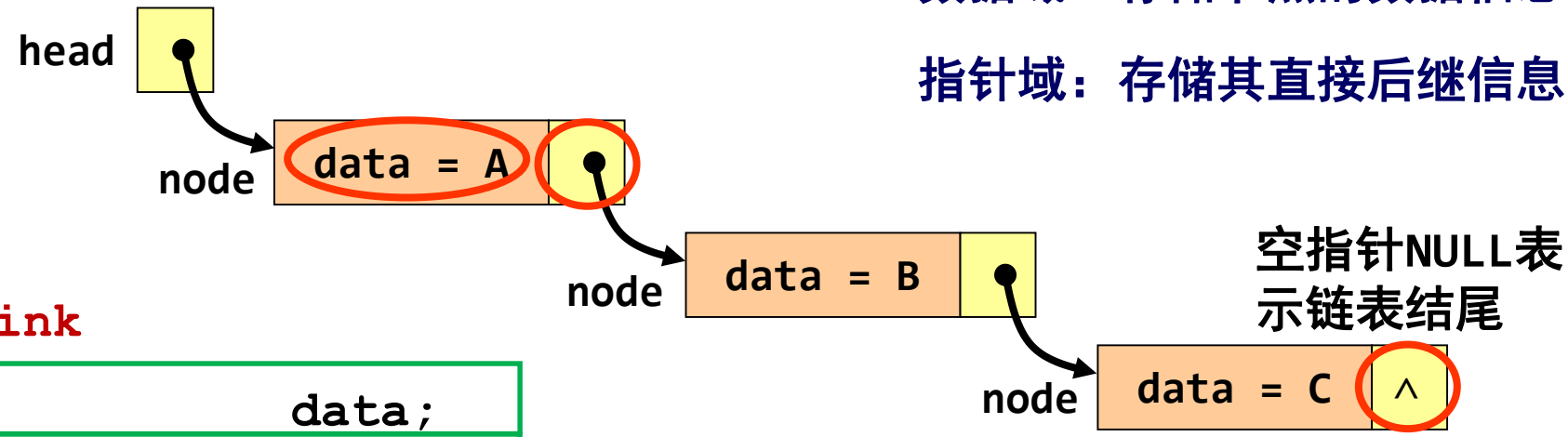
可包含指向本结构体类型的指针变量

—递归数据结构

# 动态数据结构

- 典型代表—单向链表（Linked Table）
  - 用一组任意的存储单元（不一定连续）存储线性表数据

头指针：
访问链表
的关键




数据域：存储节点的数据信息

指针域：存储其直接后继信息

空指针NULL表示链表结尾

```
struct link
{
    int             data;
    struct link *next;
};
```

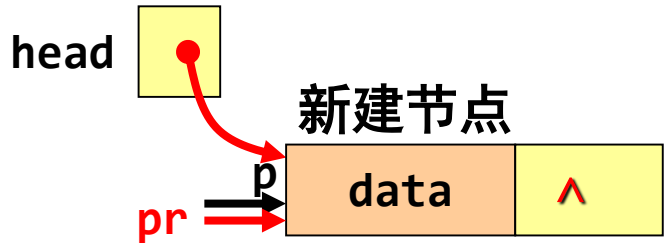
n个节点链接成一个表—线性表的链式存储结构

只包含一个指针域，又称线性链表或单向链表

# 向链表中添加节点

- 若原链表为空表(head == NULL)，则将新建节点p置为头节点

(1) head = p




```
struct  link
{
    int data;
    struct link *next;
};
```

(2) pr = p

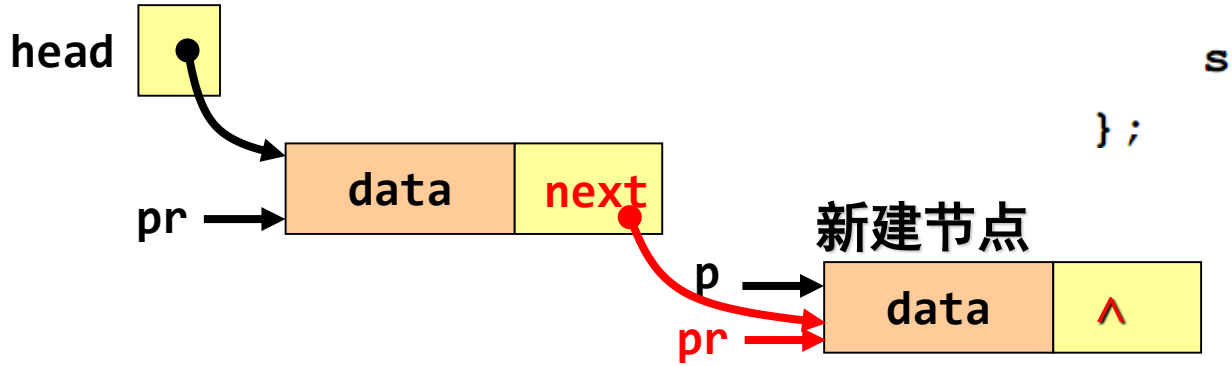
(3) pr->next = NULL

```
p = (struct link *)malloc(sizeof(struct link));
p->data = nodeData;
```

# 向链表中添加节点

- 若原链表为非空，则将新建节点p添加到表尾

(1) pr->next = p




```
struct  link
{
    int data;
    struct link *next;
};
```

(2) pr = p

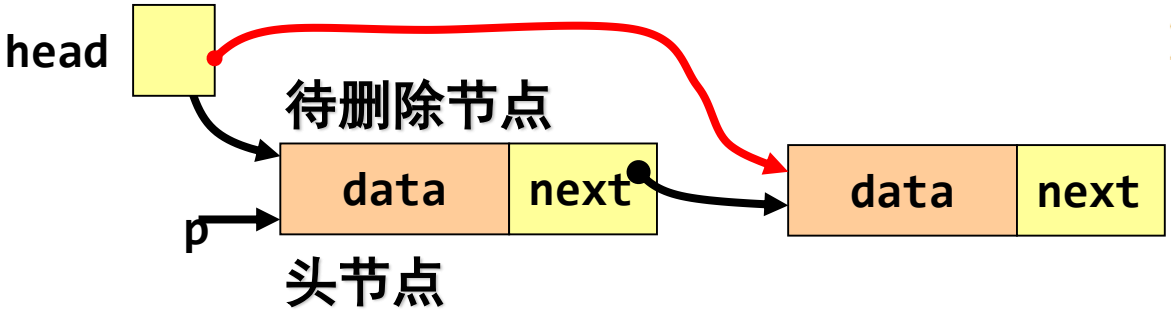
(3) pr->next = NULL

# 链表的删除操作

- 若原链表为空表，则退出程序
- 若待删除节点p是头节点，则将head指向当前节点的下一个节点即可删除当前节点

(1) head = p->next

(2) free(p)




```
struct  link
{
    int data;
    struct link *next;
};
```

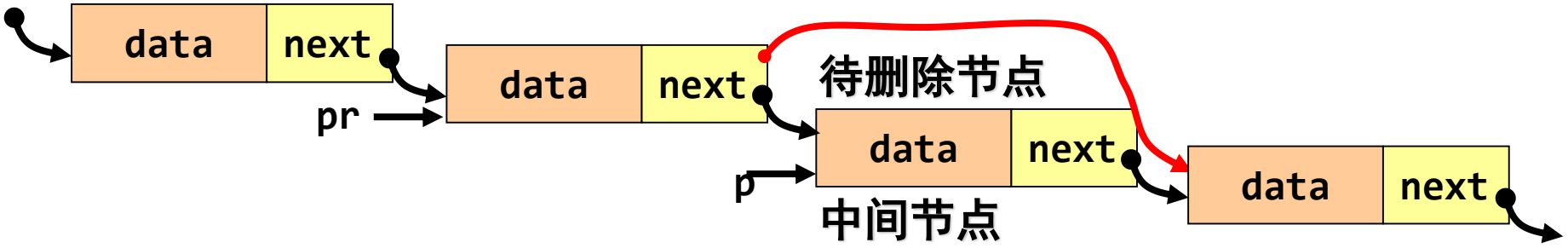
# 链表的删除操作

- 若待删除节点不是头节点，则将前一节点的指针域指向当前节点的下一节点

  (1) pr->next = p->next

  (2) free(p)

```
struct  link
{
    int data;
    struct link *next;
};
```

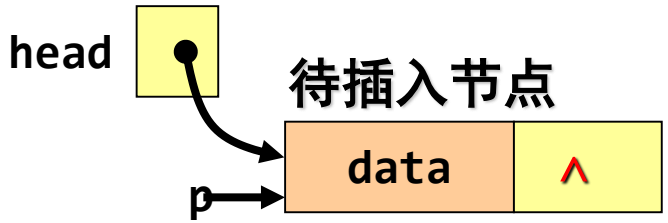



- 若已搜索到表尾（p->next == NULL）仍未找到待删除节点，则显示“未找到”

# 链表的插入操作

- 若原链表为空表，则将新节点p作为头节点，让head指向新节点p

(1) head = p




```
struct  link
{
     int data;
     struct link *next;
};
```

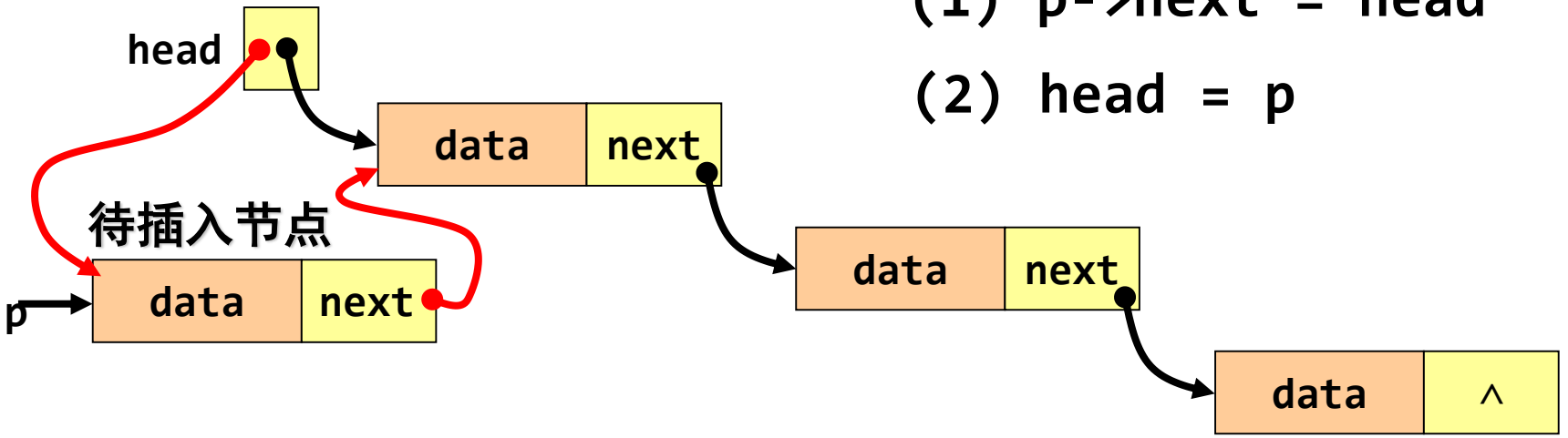
(2) p->next = NULL

# 链表的插入操作

- 若原链表为非空，则按节点值的大小（假设已升序排序）确定插入新节点的位置
- 若在头节点前插入节点，则将新节点的指针域指向原链表的头节点，且让head指向新节点

能否换一下赋值顺序？

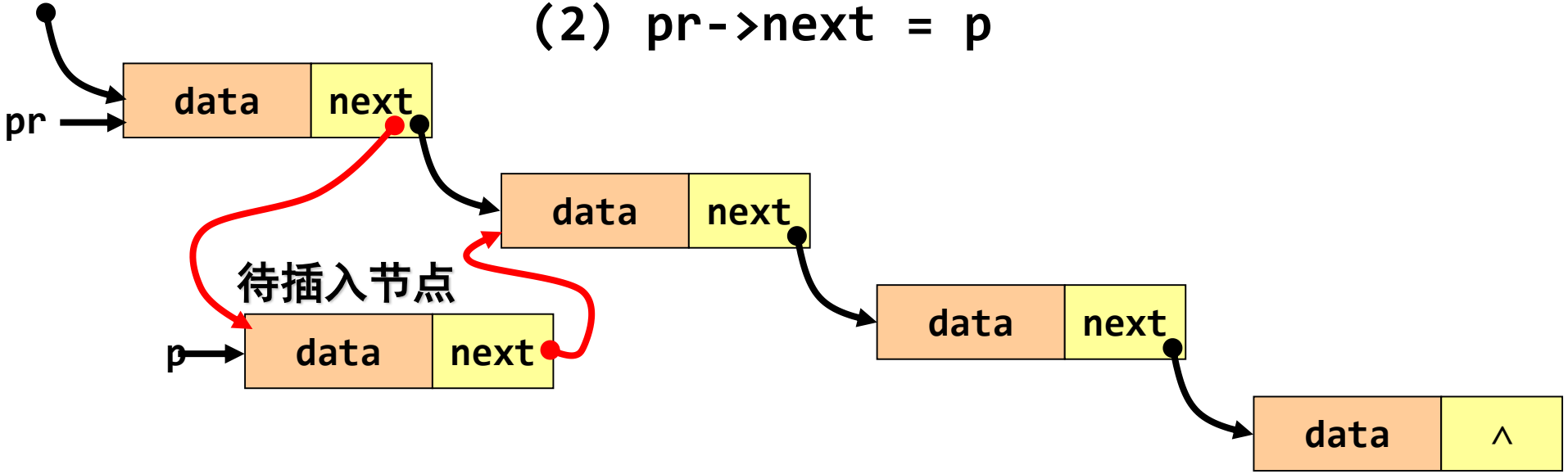
```
(1) p->next = head
(2) head = p
```



# 链表的插入操作

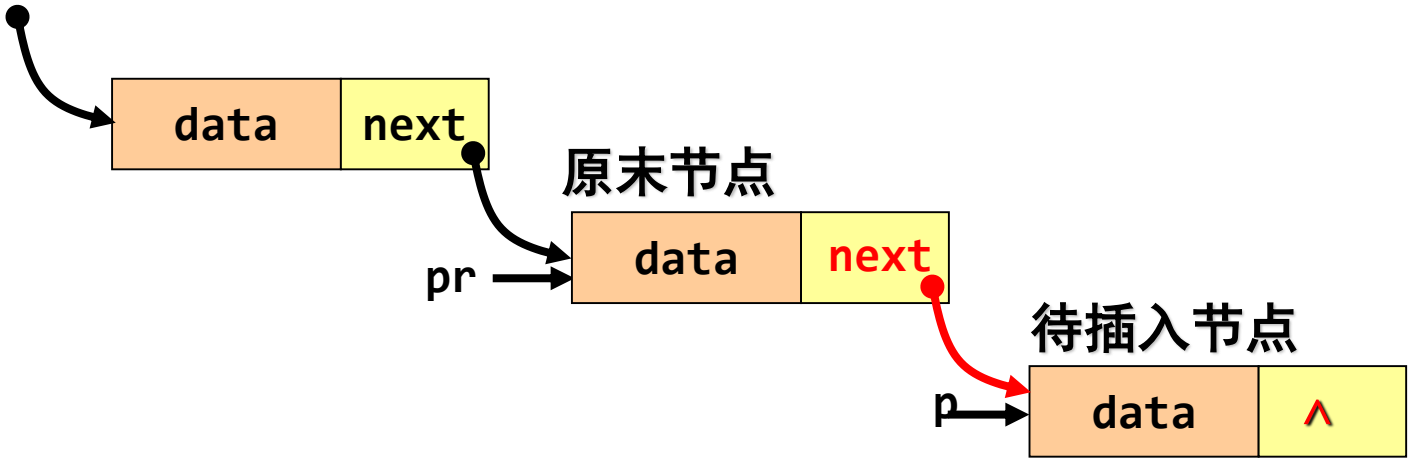
- 若在链表中间插入新节点，则将新节点的指针域指向下一节点且让前一节点的指针域指向新节点

```
(1) p->next = pr->next
(2) pr->next = p
```



# 链表的插入操作

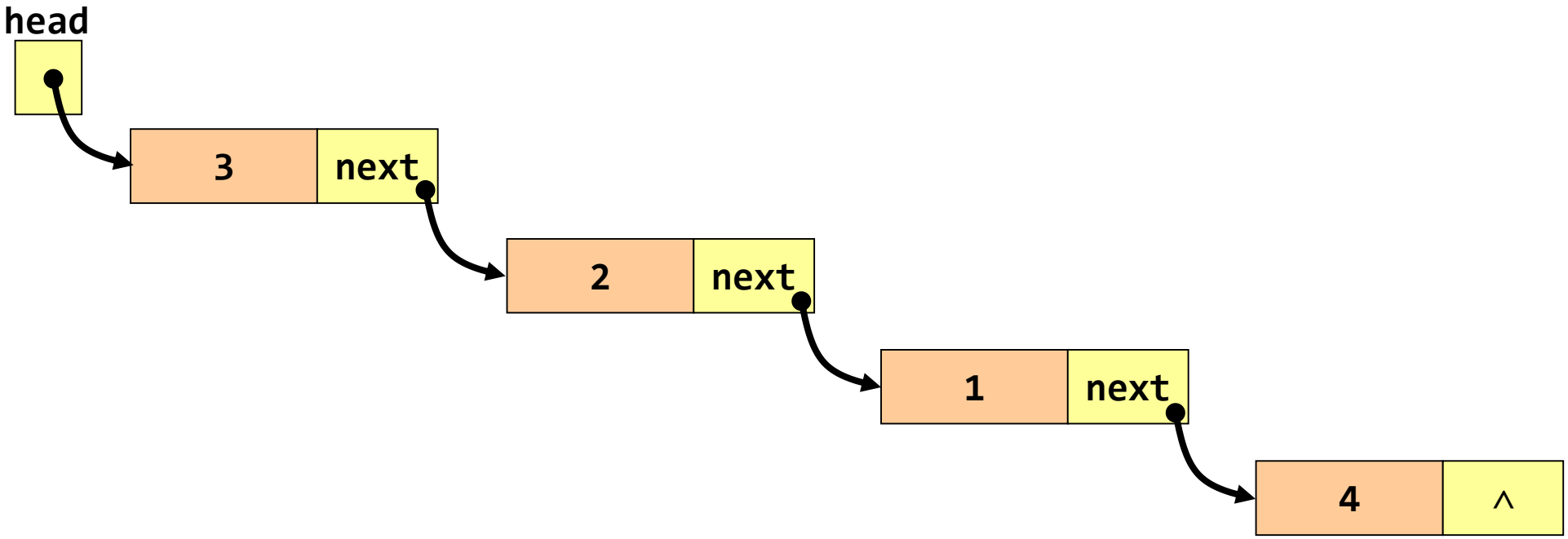
- 若在表尾插入新节点，则末节点指针域指向新节点




```
(1) pr->next = p
(2) p->next = NULL
```

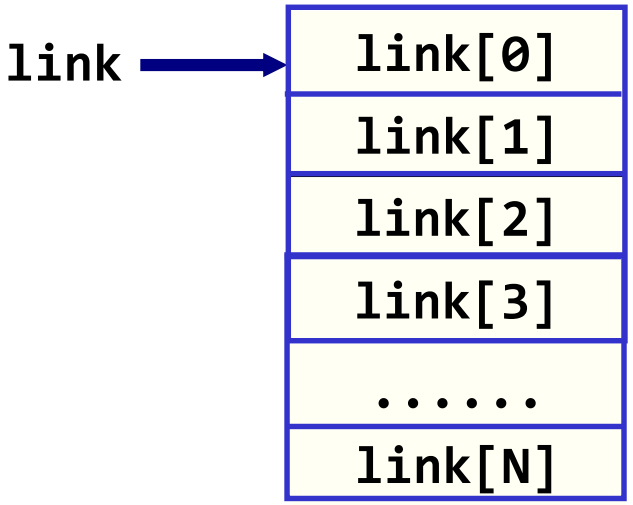
# 链表的排序

- 策略1：
  - 交换节点内容
- 策略2：
  - 通过增删节点，改变链接顺序

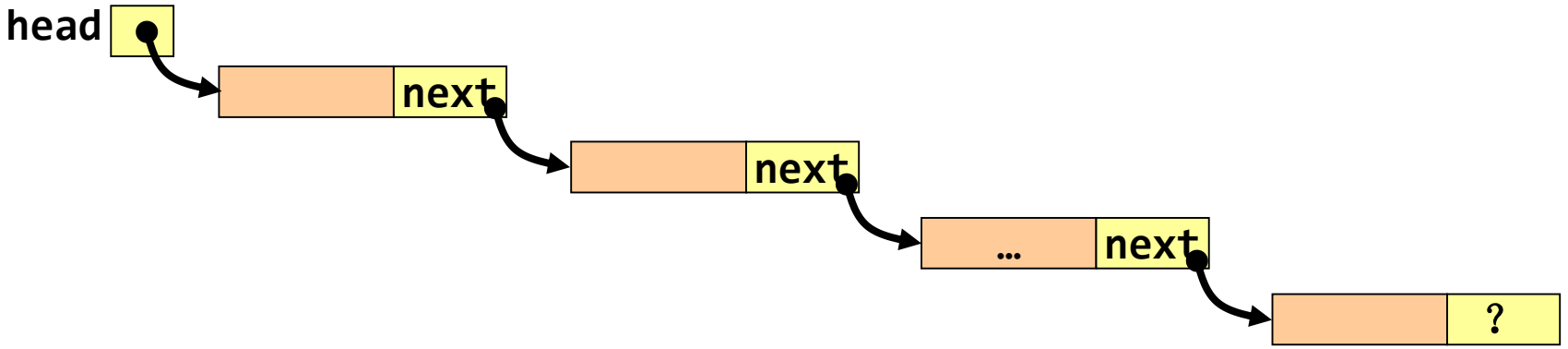


# 循环报数问题

```
typedef struct person
{
    int num;      //自己的编号
    int nextp;    //下一个人的编号
}LINK;
LINK link[N+1];   //结构体数组，顺序存储
```




```
typedef struct person
{
    int num;
    struct person *next; //下一个节点的指针
}LINK;
LINK *head;      //循环链表的头指针，链式存储
```



# 循环报数问题

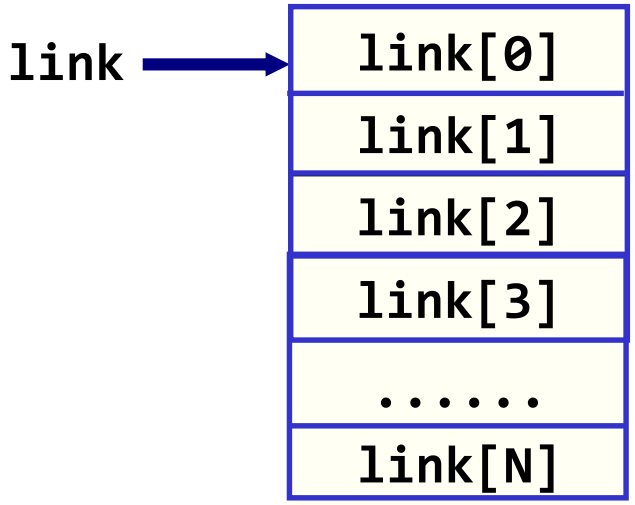
```
void CreatQueue(LINK link[], int n);
void NumberOff(LINK link[], int n, int m);
void PrtLastNum(LINK link[], int n);
int main()
{
    int m, n;
    LINK link[N+1];

    printf("Input n,m(n>m):");
    scanf("%d,%d", &n, &m);

    CreatQueue(link, n);
    last = NumberOff(link, n, m);
    printf("\n最后的成员是：%d\n", last);

    return 0;
}
```

```
#include <stdio.h>
#define  N  150
typedef struct person
{
    int num;     //自己的编号
    int nextp;   //下一个人的编号
}LINK;
```



# 循环报数问题

```
void CreatQueue(LINK link[], int n)
{
    int i;
    for (i=1; i<=n; i++)
    {
        link[i].num = i;
        if (i == n)
        {
            link[i].nextp = 1;
        }
        else
        {
            link[i].nextp = i + 1;
        }
    }
}
```

```
typedef struct person
{
    int num;    //自己的编号
    int nextp;  //下一个人的编号
}LINK;
```

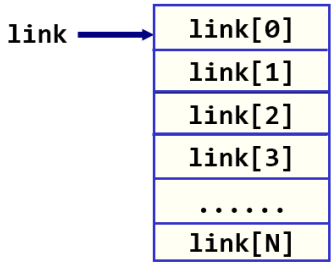
```
CreatQueue(link, n);
last = NumberOff(link, n, m);
```

```
int NumberOff(LINK link[], int n, int m)
{
    int h = n, i, j, last;
    printf("出圈成员及顺序: ");
    for (j=1; j<n; j++) //出队n-1人
    {
        i = 0;
        while (i != m) //没有报到m
        {
            h = link[h].nextp;
            if (link[h].num != 0)
            {
                i++;  //报数计数器
            }
        }
        printf("%3d", link[h].num);
        link[h].num = 0; //出队元素标记为0
    }
    for (i=1; i<=n; i++)
    {
        if (link[i].num != 0)
        {
            last = link[i].num;
        }
    }
    return last;
}
```

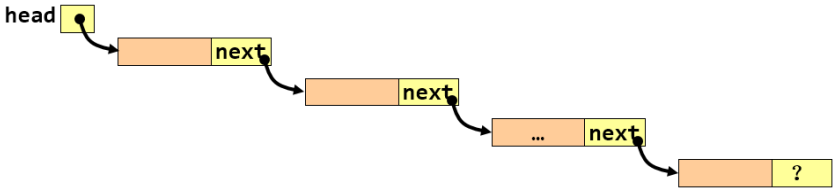
# 循环链表解决循环报数问题

```
int main()
{
    int m, n;
    LINK link[N+1];  //结构体数组
    printf("Input n,m(n>m):");
    scanf("%d,%d", &n, &m);
    CreatQueue(link, n);
    last = NumberOff(link, n, m);
    printf("\n最后的成员是：%d\n", last);
    return 0;
}
```

```
typedef struct person
{
    int num;     //自己的编号
    int nextp;   //下一个人的编号
}LINK;
```



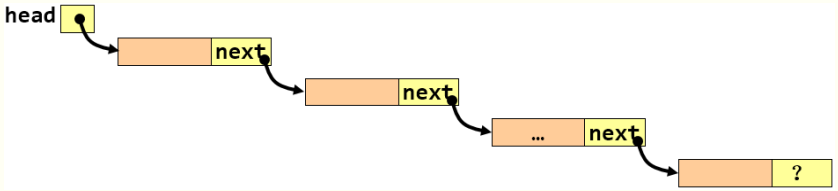

```
typedef struct person
{
    int num;
    struct person *next; //下一个节点的指针
} LINK;
```




```
int main()
{
    LINK *head; //循环链表的头指针
    int m, n, last;
    printf("Input n,m(n>m):");
    scanf("%d,%d", &n, &m);
    head = Create(n);
    last = NumberOff(head, n, m);
    printf("最后的成员是：%d\n", last);
    DeleteMemory(head, n);
    return 0;
}
```

# 循环链表解决循环报数问题

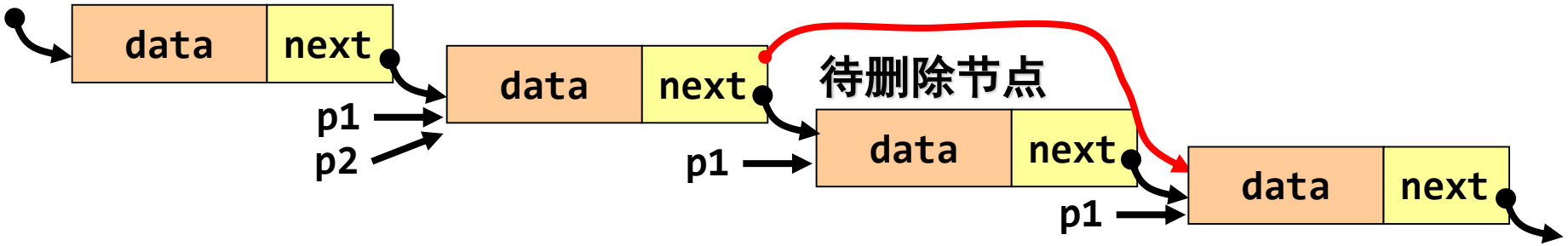
```
LINK *Create(int n)  //未考虑节点内存申请是否成功
{
    int i;
    LINK *p1, *p2, *head = NULL;
    p2 = p1 = (LINK*)malloc(sizeof(LINK)); //申请新节点内存
    for (i=1; i<=n; i++)
    {
        if (i == 1)
            head = p1;      //头指针指向首节点
        else
            p2->next = p1; //前驱节点链上后继（新）节点
        p1->num = i;
        p2 = p1;            //记录前驱节点的指针
        p1 = (LINK*)malloc(sizeof(LINK)); //申请新节点内存
    }
    p2->next = head; //尾节点链到头节点，循环链表
    return head;
}
```



```
LINK *Create(int n) //加入了节点内存申请是否成功的判断
{
    int i;
    LINK *p1, *p2, *head = NULL;
    p2 = p1 = (LINK*)malloc(sizeof(LINK));
    if (p1 == NULL)
    {
        printf("No enough memory to allocate!\n");
        exit(0);
    }
    for (i=1; i<=n; i++)
    {
        if (i == 1)
            head = p1;
        else
            p2->next = p1;
        p1->num = i;
        p2 = p1;
        p1 = (LINK*)malloc(sizeof(LINK));
        if (p1 == NULL)
        {
            printf("No enough memory to allocate!\n");
            DeleteMemory(head, i); //释放前i个成功分配的节点的内存
            exit(0);
        }   //最后一次循环分配的这个内存未被使用哦
    }
    free(p1); //释放最后一次循环中分配但未使用的新节点内存
    p2->next = head;
    return head;
}
```

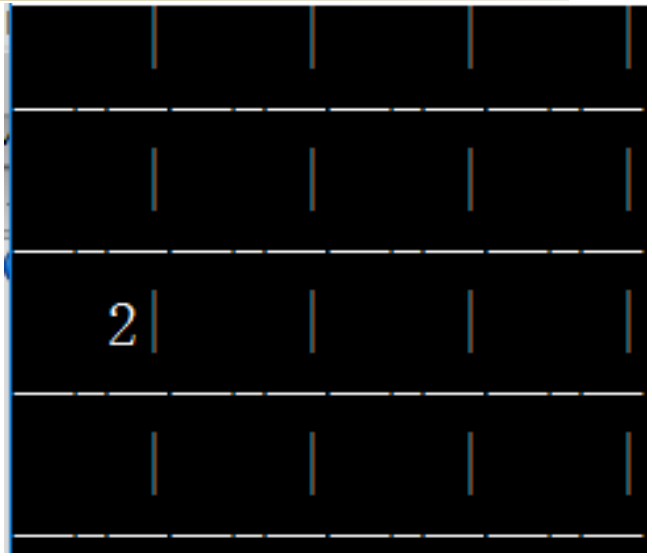
```
void DeleteMemory(LINK *head,int n)
{
    LINK *p = head, *pr = NULL;
    int i;
    for (i=1; i<=n; i++)
    {
        pr = p;
        p = p->next;
        free(pr);
    }
}
```

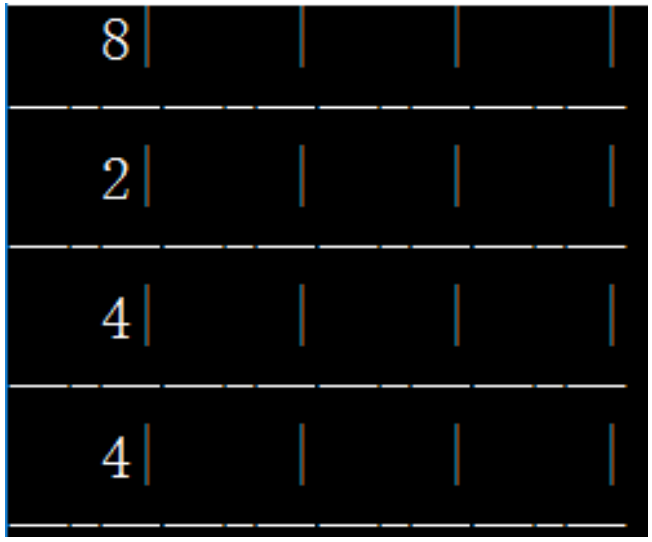
```
int NumberOff(LINK *head, int n, int m)
{
    int i, j;
    LINK *p1 = head, *p2 = p1;
    if (n == 1 || m == 1)
    {
        return n;
    }
    for (i=1; i<n; i++)        //将n-1个节点删掉
    {
        for (j=1; j<m-1; j++) //数出前m-2个节点
        {
            p1 = p1->next;
        }
        p2 = p1;
        p1 = p1->next;
        p1 = p1->next;
        p2->next = p1;         //删掉第m个节点
    }
    return p1->num;
}
```

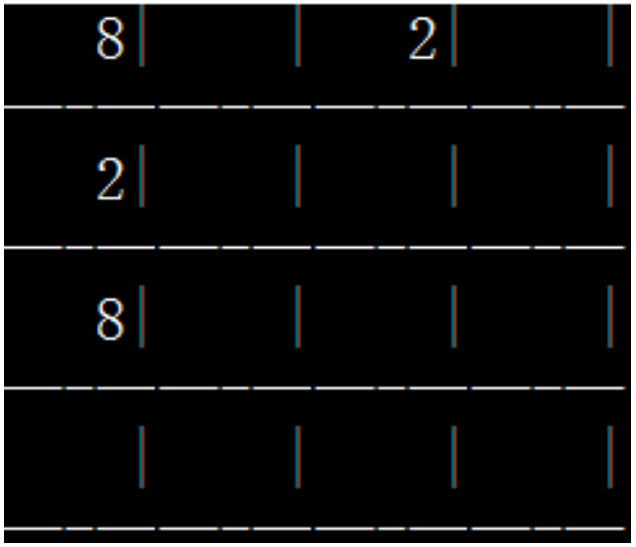


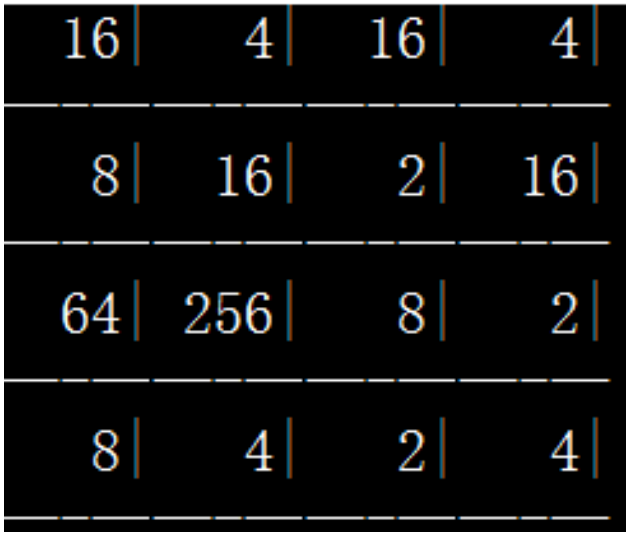
# 2048数字游戏

- 用户选择移动操作后，在方格中寻找可以相加的相邻且相同的数字，相邻且相同的数字相消，变为更大的数字。
- 玩家每次移动数字后会新增一个方块：2或4，2的概率>4的概率。
- 所有方格填满，还没有加到2048，则游戏失败



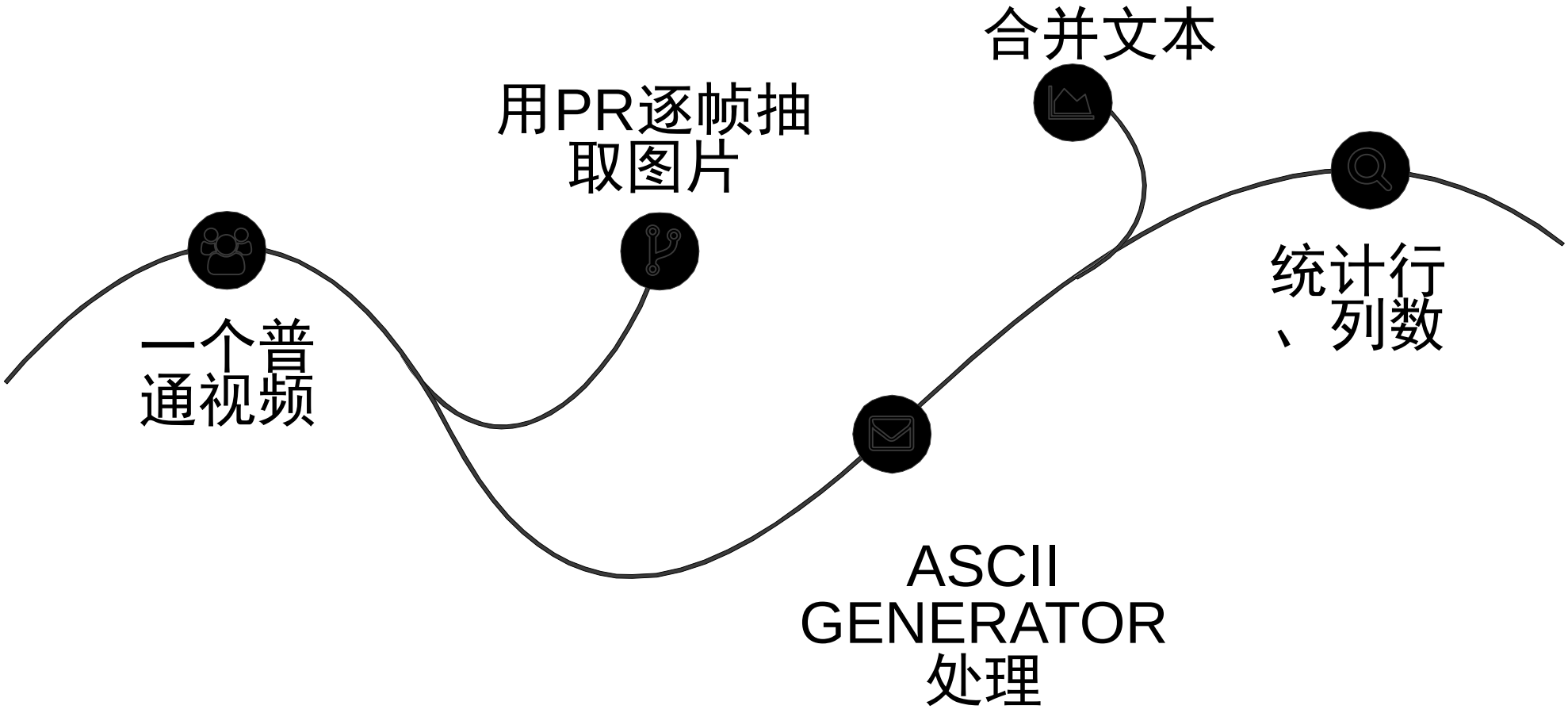



# 2048Bricks


Exe_final
2048 BRICKS
SCORE
128
2
128
32
8
4


2
16
You win!
2
4
2
Keep going
Try again
2048
2
8
4

# 字符动画

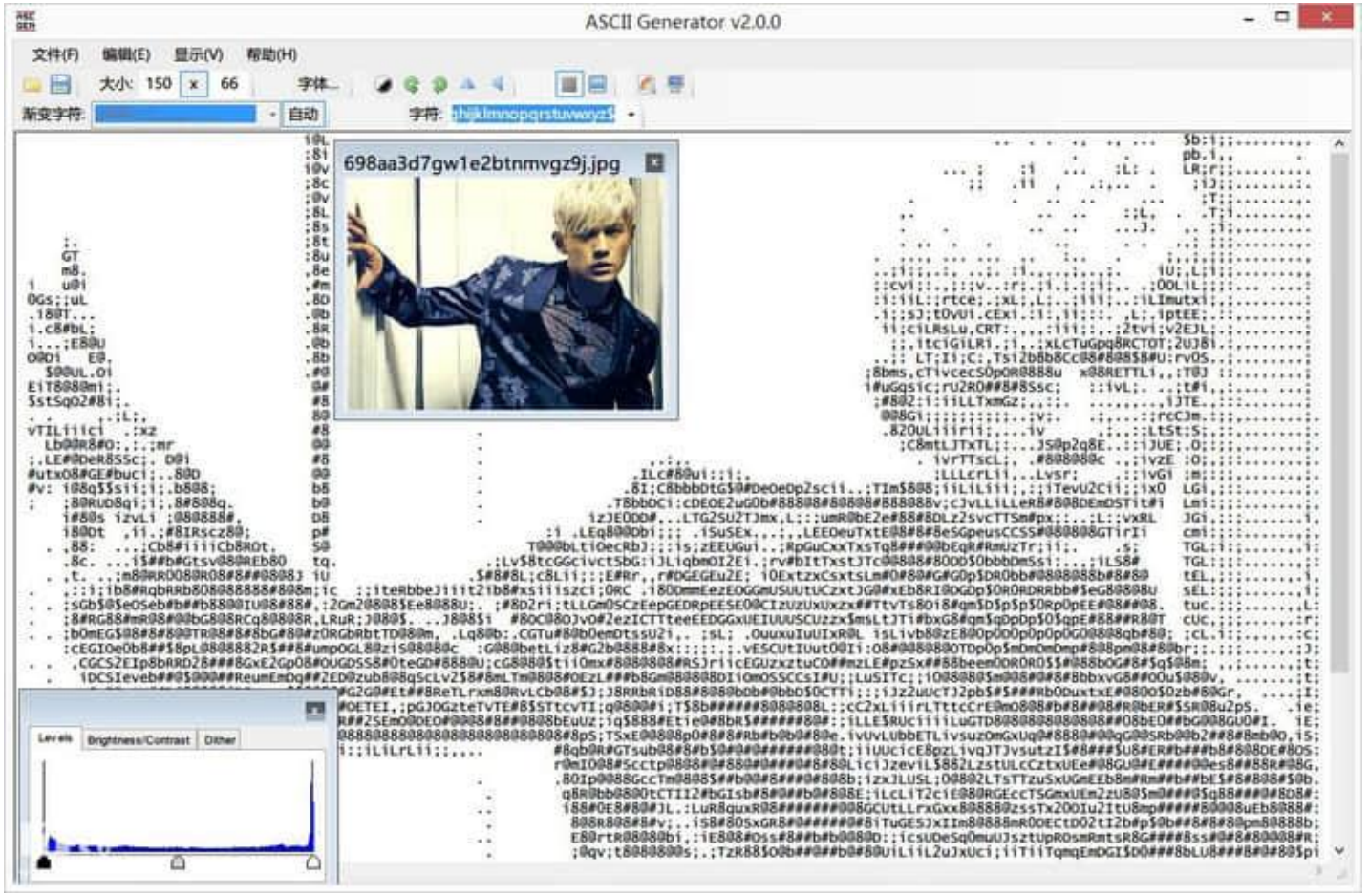


# 从视频逐帧提取导出图片的工具

- adobe premiere pro 4

# 制作ASCII图片的工具

- **ASCII Generator 2.0**



# 字符动画制作工具

- **ASCII Animator 1.7**
  - 将GiF动画转换成ASCII动画