

Day01

一. 数据结构

1. 什么是数据结构

数据结构：数据的逻辑结构、存储结构及操作

2. 数据

数据：信息的载体，不再是单纯的数值

数据元素：由若干个基本项(数据项)组成(图书信息系统 一本一本的图书就是数据元素)

一本书是数据元素，一本书(书名、作者、出版日期....)

数据元素是数据的基本单位

数据项：(书名、作者、出版日期....)

节点：数据元素又称为节点

3. 逻辑结构

逻辑结构：数据与数据之间的联系和规律

逻辑关系	逻辑结构	特点	应用
线性关系	线性结构	一对一	线性表 (顺序表、链表、栈、队列...)
层次关系	树状结构	一对多	树
网状关系	图状结构	多对多	图

4. 存储结构

存储结构：数据的逻辑结构在计算机上具体实现

[用代码来表达逻辑结构](#)(表达数据与数据之间的关系)

(1)顺序存储结构：顺序存储，在内存当中是 连续 存储的 数组，顺序存储结构

(2)链式存储结构：链式存储结构，在内存当中是 不连续 存储的，通过指针将节点连接在一起

(3)索引存储结构：两张表 源数据表 + 索引表 索引表用来确定在那一块中

(4)散列存储结构：哈希表，存的时候按照对应关系存，取的时候按照对应关系取

5. 操作

增 删 改 查

1. 购买了一本书，增加图书信息

2. 丢了一本书，删除图书信息

3. 一本图书信息登记错了，改信息

4. 查询系统中剩余的《三国演义》有几本，查询书的价格

二. 算法

算法：解决问题的思想办法
如何评价一个算法的好坏???? 时间复杂度和空间复杂度
消耗时间的多少
占用内存的大小
代码的 可读性、可维护性、可移植性

#练习1:

定义一个全局变量 `int last = 5;` //last的值，时刻代表当前数组中有效元素的个数，last是几，有效元素就是几

`int a[100] = {11,22,33,44,55};`//部分初始化，当前的5个数，被称为有效元素个数

//顺序表的插入和删除操作

`int a[100] = {11,22,33,44,55};`

`showArray(a);`

//增加,在第3个位置，插入一个数1000

`insertIntoA(a, 3, 1000);`

`showArray(a);`//11 22 1000 33 44 55

`deleteFromA(a, 3);`//删除第3个位置的数据

`showArray(a);`//11 22 33 44 55

`int last = 5;`//全局变量,5代表 有效元素 的个数

//int* p 保存数组首地址

//int post 插入的位置

//int x 插入的数据

`void InsertInotA(int* p, int post, int x)`

{

}

//int* p 保存数组首地址

//int post 删除的位置

`void deleteFromA(int* p, int post)`

{

}

//打印数组中的有效元素

`void showArray(int* p)`

{

int i;

for(i = 0; i < last; i++)

{

printf("%d ",p[i]);

}

printf("\n");

}

`int main()`

{

int a[100] = {10,20,30,40,50};

return 0;

}

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

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

```

int last = 5; //全局变量, 5代表 有效元素 的个数
//int* p 保存数组首地址
//int post 插入的位置
//int x 插入的数据
void InsertInotA(int* p, int post, int x)
{
    //1. 找出需要整体移动的下标范围 插入位置下标 --- 最后一个有效元素下标
    //插入位置下标 post-1
    //最后一个有效元素下标 last-1
    // last-1 ---- post-1 的下标范围的元素需要整体向后移动一个位置
    //2. 整体向后移动
    int i;
    for(i = last-1; i >= post-1; i--)
        p[i+1] = p[i];
    //3. 在插入位置, 插入新的数据x
    p[post-1] = x; //post代表的是第几个, 第几个-1得到的插入位置的下标
    //4. 有效元素个数+1
    last++;
}
//int* p 保存数组首地址
//int post 删除的位置
void deleteFromA(int* p, int post)
{
    //1. 找出需要整体向前移动的下标范围 删除位置的下一个位置元素的下标 --- 最后一个有效元素
    //删除位置的后一个位置的下标 post
    //最后一个有效元素下标 last-1
    //post ---- last-1
    //2. 整体向前移动
    int i;
    for(i = post; i <= last-1; i++)
        p[i-1] = p[i];
    //3. 删除之后, 有效元素个数-1
    last--;
}
//打印数组中的有效元素
void showArray(int* p)
{
    int i;
    for(i = 0; i < last; i++)
    {
        printf("%d ", p[i]);
    }
    printf("\n");
}
int main()
{
    int a[100] = {11, 22, 33, 44, 55};
    showArray(a);
    InsertInotA(a, 3, 1000);
    showArray(a);
    deleteFromA(a, 3);
    showArray(a);
    return 0;
}

```

```
linux@ubuntu:~$ ./a.out
11 22 33 44 55
11 22 1000 33 44 55
11 22 33 44 55
```

三. typedef 关键字

typedef //type 类型 dedfine 定义
作用：类型重定义 //给数据类型重新起个名字

1. typedef 与 结构体

(1) 方法一

先定义结构体，再进行类型重定义

```
//方法一. 先定义结构体，再进行类型重定义
#include <stdio.h>
struct student
{
    char name[20];
    int age;
};

typedef int MMM; //将数据类型int 重定义为MMM
typedef struct student stu_t; //将struct student 类型重定义为 stu_t

int main(int argc, const char *argv[])
{
    int a = 10;
    MMM b = 20; // MMM 等价于 int类型
    printf("%d %d\n", a, b);

    struct student s1 = {"asan", 19};
    stu_t s2 = {"xixi", 20}; //stu_t 等价于 struct student
    printf("%s %d    %s %d\n", s1.name, s1.age, s2.name, s2.age);
    return 0;
}
```

```
linux@ubuntu:~$ ./a.out
10 20
asan 19    xixi 20
```

(2) 方法二

在定义结构体的同时，进行类型重定义

```
#include <stdio.h>

//方法二 在定义结构体的同时,进行类型重定义
typedef struct student
{
    char name[20];
    int age;
}stu_t;

int main(int argc, const char *argv[])
{
    struct student s1 = {"asan",19};
    stu_t s2 = {"xixi",20}; //stu_t 等价于 struct student
    printf("%s %d %s %d\n",s1.name,s1.age,s2.name,s2.age);
    return 0;
}
```

(3) 方法三

匿名结构体重命名

```
#include <stdio.h>

//方法三 匿名结构体 重定义
typedef struct
{
    char name[20];
    int age;
}stu_t;

int main(int argc, const char *argv[])
{
    //因为是匿名结构体,只能直接用小名
    stu_t s2 = {"xixi",20}; //stu_t 就是结构体类型
    printf("%s %d\n",s2.name,s2.age);
    return 0;
}
```

四. 顺序表

线性表：顺序表、链表(单向链表，单向循环链表，双向链表，双向循环链表)、栈、队列
线性表特征：一对一 每个节点最多有一个前驱和一个后继(首节点无前驱，尾节点无后继)

顺序表：

sequence //顺序

list //表

create //创建

```
full //满
empty //空
insert into //插入
delete from //删除
```

逻辑结构： 线性结构

存储结构： 顺序存储结构

操作：

//顺序表的结构体定义

```
typedef struct
{
    int a[100]; //顺序表操作的数组
    int last; //last时刻代表顺序表中有效元素的个数
} seqlist_t; //seq sequence 顺序 list 表
int a[100] = {1, 2, 34, 56, 9, 10};
```

```
//1. 创建一个空的顺序表
//2. 判断顺序表是否为满, 满返回1, 未返回0
//3. 向顺序表的指定位置插入数据
//4. 遍历顺序表的有效元素
//5. 判断顺序表是否为空, 空返回1, 未空返回0
//6. 删除顺序表指定位置的数据
//7. 求顺序表的长度
//8. 查找指定数据, 出现在顺序表中的位置
//9. 清空顺序表
```

4.1 指定位置插入数据操作

编程思想：

1. 找到需要整体向后移动的有效元素的下标范围
post-1 --- last-1
2. 整体向后移动一个位置
3. 在插入位置，放上插入元素
4. 有效元素个数+1

顺序表插入操作 第3个位置插入数据 1000

```
int post = 3; //第3个位置插入
p->last = 5; //有效元素个数5个
p->a[100] = {11, 22, 33, 44, 55};
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
11	22	1000	33	44	55	0

```
int post = 4;
int last = 5;
移动的范围 3 --- 4
```

```
int post = 2;
int last = 5;
移动的范围 1 ---- 4
```

编程思想：

1. 找到需要整体向后移动的有效元素的下标范围
2 - 4 整体向后移动一个位置
post-1 --- last-1
2. 整体向后移动一个位置
3. 在插入位置，放上插入元素
4. 有效元素个数+1

```
a[5] = a[4];
a[4] = a[3];
a[3] = a[2];
//i代表的是=右边 4 3 2
for(i = last-1; i >= post-1; i--)
{
    p->a[i+1] = p->a[i];
}
p->a[post-1] = 1000;
p->last++;
```

//i代表的是=左边 5 4 3
for(i = last; i >= post; i--)
{
 p->a[i] = p->a[i-1];
}

```

int insertIntoSeqList(seqList_t* p, int post, int x)
{
    //0. 容错判断
    if(post <= 0 || post > p->last+1 || isFullSeqList(p))
    {
        printf("insertIntoSeqList failed!!\n");
        return -1; //通常用-1来表达失败
    }
    //1. 将从插入位置的元素下标到最后有效元素整体向后移动一个位置
    //post-1 ----- p->last-1
    int i;
    for(i = p->last-1; i >= post-1; i--)
        p->a[i+1] = p->a[i];
    //2. 在插入位置, 放上插入的数据x
    p->a[post-1] = x; //post-1得到插入位置的下标
    //3. 有效元素个数+1
    p->last++;
    return 0; //代表插入成功
}

```

4.2 指定位置删除数据

编程思想:

1. 找到需要整体向前移动的有效元素的下标范围

post --- last-1

2. 整体向前移动一个位置
3. 有效元素个数-1

顺序表删除操作 删除第3个位置的数据

```

int post = 3; //第3个位置删除
int last = 6; //有效元素个数6个
int a[100] = {11, 22, 1000, 33, 44, 55};

```

1	2	3	4	5	6	
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
11	22	33	44	55	55	0

```

int post = 3;
int last = 6;
移动的范围是 3 --- 5

```

编程思想:

1. 找到需要整体向前移动的有效元素的下标范围
3 - 5 整体向后移动一个位置

post --- last-1

2. 整体向前移动一个位置
3. 有效元素个数-1

```

a[2] = a[3];
a[3] = a[4];
a[4] = a[5];
//i代表的是=右边 3 4 5
for(i = post; i <= last-1; i++)
{
    a[i-1] = a[i];
}
last--;

```

```

int post = 2;
int last = 6;
移动的范围是 2 ---- 5

```

```

int deleteFromSeqList(seqList_t* p, int post)
{
    //0. 容错判断

```

```

if(post <= 0 || post > p->last || isEmptySeqList(p))
{
    printf("deleteFromSeqList failed!!\n");
    return -1;
}
//1. 将从删除位置的后一个位置的下标到最后一个有效元素, 整体向前移动一个位置
//post ---- p->last-1
int i;
for(i = post; i <= p->last-1; i++)
    p->a[i-1] = p->a[i];
//2. 有效元素个数-1
p->last--;
return 0; //删除成功
}

```

4.3 顺序表9个函数实现

```

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

#define N 100
typedef struct
{
    int a[N]; //顺序表操作的数组
    int last; //last的值, 时刻代表的是数组中有效元素的个数
} seqList_t;

//1. 创建一个空的顺序表
seqList_t* createEmptySeqList()
{
    seqList_t* p = malloc(sizeof(seqList_t));
    if(p == NULL)
    {
        printf("malloc failed!!\n");
        return NULL;
    }
    p->last = 0; //因为是空的顺序表, 有效元素个数初始化为0
    return p; //将malloc申请空间的首地址返回
}

//2. 判断顺序表是否为满 表满返回值是1, 未满是0
int isFullSeqList(seqList_t* p)
{
    #if 0
    //方法一
    if(p->last == N) //有效元素个数 == 数组的长度, 说明表满
        return 1;
    else
        return 0;
    //方法二
    return p->last == N ? 1 : 0;
    #endif
    //方法三
    return p->last == N; //p->last == N 是表达式, 结果只有两个 1 或 0, 条件为真, 真用1表达
    //条件为假, 假用0表达, 正好1或0, 做为返回值返回
}

```



```

//3. 向顺序表的指定位置插入数据x
int insertIntoSeqList(seqList_t* p, int post, int x)
{
    -1 1000
    //0. 容错判断
    if(post <= 0 || post > p->last+1 || isFullSeqList(p))
    {
        printf("insertIntoSeqList failed!!\n");
        return -1; //通常用-1来表达失败
    }
    //1. 将从插入位置的元素下标到最后一个有效元素整体向后移动一个位置
    //post-1 ----- p->last-1
    int i;
    for(i = p->last-1; i >= post-1; i--)
        p->a[i+1] = p->a[i];
    //2. 在插入位置, 放上插入的数据x
    p->a[post-1] = x; //post-1得到插入位置的下标
    //3. 有效元素个数+1
    p->last++;
    return 0; //代表插入成功
}

//4. 遍历顺序表有效元素
void showSeqList(seqList_t* p)
{
    int i;
    for(i = 0; i < p->last; i++)
    {
        printf("%d ", p->a[i]);
    }
    printf("\n");
}

//5. 判断顺序表是否为空, 空返回1, 未空返回0
int isEmptySeqList(seqList_t* p)
{
    return p->last == 0; //判断有效元素个数是否 == 0
}

//6. 删除顺序表中指定位置的数据
int deleteFromSeqList(seqList_t* p, int post)
{
    //0. 容错判断
    if(post <= 0 || post > p->last || isEmptySeqList(p))
    {
        printf("deleteFromSeqList failed!!\n");
        return -1;
    }
    //1. 将从删除位置的后一个位置的下标到最后一个有效元素, 整体向前移动一个位置
    //post ---- p->last-1
    int i;
    for(i = post; i <= p->last-1; i++)
        p->a[i-1] = p->a[i];
    //2. 有效元素个数-1
    p->last--;
    return 0; //删除成功
}

//7. 查找指定的数据x, 出现在顺序表中的位置
int searchDataSeqList(seqList_t* p, int x)
{
    int i;

```

```

//遍历有效元素查找
for(i = 0; i < p->last; i++)
{
    if(p->a[i] == x)
        return i; //返回值代表的是,元素在数组中的下标
}
//如果程序能走到这,说明不存在
return -1;

}

//8.求顺序表的长度
int getLengthSeqList(seqList_t* p)
{
    return p->last; //有效元素个数就是顺序表的长度
}

//9.清空顺序表
void clearSeqList(seqList_t* p)
{
    p->last = 0; //有效元素个数赋值为0
}

int main(int argc, const char *argv[])
{
    //1.创建一个空的顺序表
    // seqList_t s;
    // s.last = 0; //将有效元素的个数初始化为0,代表这是空的顺序表
    seqList_t* p = createEmptySeqList();
    //2.向顺序表中插入数据
    insertIntoSeqList(p, 1, 11);
    insertIntoSeqList(p, 2, 22);
    insertIntoSeqList(p, 3, 33);
    insertIntoSeqList(p, 4, 44);
    insertIntoSeqList(p, 5, 55);
    printf("33在数组中的下标是%d\n", searchDataSeqList(p, 33));
    showSeqList(p);
    insertIntoSeqList(p, 3, 1000);
    showSeqList(p);
    deleteFromSeqList(p, 3);
    showSeqList(p);
    printf("len is %d\n", getLengthSeqList(p));
    clearSeqList(p);
    printf("len is %d\n", getLengthSeqList(p));

    free(p); //手动释放
    return 0;
}

```

```
linux@ubuntu:~$ ./a.out
33在数组中的下标是2
11 22 33 44 55
11 22 1000 33 44 55
11 22 33 44 55
len is 5
len is 0
```

4.4 顺序表的总结

顺序表有什么缺点???

1. 插入和删除麻烦
2. 顺序表的长度是固定的，顺序表本质操作的是数组，数组的长度是固定的

顺序表有什么优点???

1. 查找速度快，可以直接通过数组下标访问元素
2. 顺序表可以实现随机访问(只要有了数据的位置下标，就可以实现访问数据)

@复习

1. 数据结构：数据的逻辑结构、存储结构及操作
2. 逻辑结构：数据与数据之间的联系和规律

逻辑关系	逻辑结构	特点	应用
线性关系	线性结构	一对一	线性表(顺序表、链表、栈、队列)
层次关系	树状结构	一对多	树
网状关系	图状结构	多对多	图

3. 存储结构

顺序存储结构：在内存当中是 连续 存储的，本质上操作是数组

链式存储结构：在内存当中是 不连续 存储的， 通过指针将每个节点连接在一起

```
struct node
{
    int data; //数据域
    struct node* next;//指针域 指向下一个节点的指针
};
```

索引存储结构

散列存储结构

4. 顺序表的操作

```
typedef struct
{
    int a[100]; //顺序表
    int last; //last的值时刻代表顺序表中有效元素的个数
}seqlist_t;
```

(1)插入操作

- a. 将插入位置的下标 -- 最后一个有效元素整体向后移动一个位置
post-1 --- last-1
- b. 在插入位置，放上数据
post-1得到插入的位置
- c. 有效元素个数+1

(2) 删除操作

- 将删除位置的下一个位置的下标 $post$ $---- last-1$
- 有效元素个数 -1

Day02 链表

线性表的链式存储结构，即链表

链表： 单向链表、单向循环链表、双向链表、双向循环链表

逻辑结构：线性结构 一对一

存储结构：链式存储结构

一、单向链表结构

将线性表 $L=(a_0, a_1, \dots, a_{n-1})$ 中各元素分布在存储器的不同存储块，称为结点，通过指针建立它们之间的联系，所得到的存储结构为链表结构。表中元素 a_i 的结点形式如图所示。



其中，结点的 **data** 域存放数据元素 a_i ，而 **next** 域是一个指针，指向 a_i 的直接后继 a_{i+1} 所在的结点。于是，线性表 $L=(a_0, a_1, \dots, a_{n-1})$ 的结构如图所示。



****单向链表又分为有头节点和无头节点两种 只是一种相对概念****

无头单向链表：链表的每个节点的数据域都是有效的

有头单向链表：链表的第一个节点的数据域是无效的



二、单向链表的节点定义

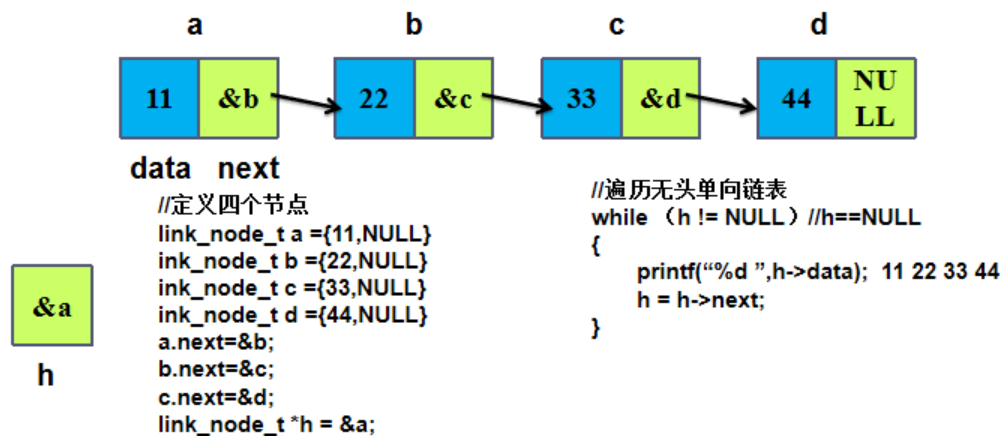
单向链表 节点的定义

```
typedef struct node
{
    int data;           //数据域
    struct node* next;  //指针域, next指针, 指向的是下一个节点的指针
}link_node_t;
```

node --> 节点

三、创建无头单向链表并遍历链表

无头单向链表



```
#include <stdio.h>

typedef struct node
{
    int data;           //数据域
    struct node* next;  //指针域 指向下一个节点的指针
}link_node_t;

int main(int argc, const char *argv[])
{
    //无头单向链表
    //定义4个节点
    link_node_t a = {11, NULL};
    link_node_t b = {22, NULL};
    link_node_t c = {33, NULL};
    link_node_t d = {44, NULL};
    //将4个节点,通过指针域连接在一起
    a.next = &b;
    b.next = &c;
    c.next = &d;
    //定义一个头指针,指向单向链表的第一个节点
    link_node_t* h = &a;
    //遍历无头的单向链表
    while(h != NULL)
```

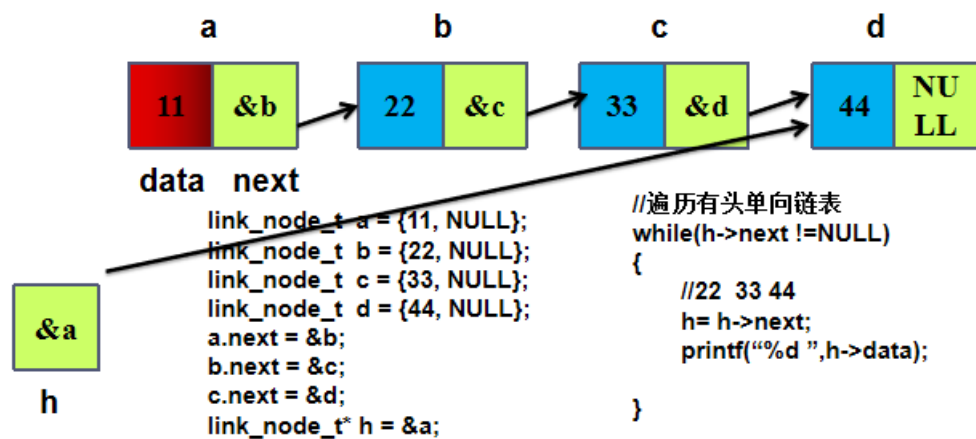
```

{
    printf("%d ",h->data);
    h = h->next;//头指针,指向下一个节点
}
printf("\n");
return 0;
}

```

四、创建有头单向链表并遍历链表

有头单向链表



```

#include <stdio.h>

typedef struct node
{
    int data;           //数据域
    struct node* next; //指针域 指向下一个节点的指针
}link_node_t;

int main(int argc, const char *argv[])
{
    //有头单向链表
    //定义4个节点
    link_node_t a = {11, NULL};
    link_node_t b = {22, NULL};
    link_node_t c = {33, NULL};
    link_node_t d = {44, NULL};
    //将4个节点,通过指针域连接在一起
    a.next = &b;
    b.next = &c;
    c.next = &d;
    //定义一个头指针,指向单向链表的第一个节点
    link_node_t* h = &a;
    //遍历有头的单向链表,第一个节点的数据域是无效的
    while(h->next != NULL)
    {

```

```

        h = h->next; //头指针,指向下一个节点
        printf("%d\n", h->data);
    }

    return 0;

}

```

五、单向链表的操作

链表可以解决 顺序表长度固定的问题，链表插入和删除方便

单向链表的操作(有头的单向链表)

- //1. 创建一个空的链表
- //2. 向链表的指定位置插入数据
- //3. 遍历链表的有效元素
- //4. 判断链表是否为空,空返回1, 未空返回0
- //5. 删除链表指定位置的数据
- //6. 求链表的长度
- //7. 查找指定数据, 出现在链表中的位置
- //8. 清空链表

5.1 创建一个空的链表

```

//1.创建一个空的链表(本质上只有一个头节点链表,将头节点的首地址返回)
link_node_t* createEmptyLinklist()
{
    //创建一个空的头节点
    link_node_t* p = malloc(sizeof(link_node_t));
    if(p == NULL)
    {
        printf("createEmptyLinklist malloc failed!!\n");
        return NULL; //表达申请失败
    }
    //申请空间,就是为了装东西
    p->next = NULL; //指针域初始化为NULL,数据域不用初始化,因为头节点数据域无效

    //将头节点的首地址返回
    return p;
}

```

5.2 遍历有头链表的有效元素

```

void showLinklist(link_node_t* p)
{
    while(p->next != NULL)
    {
        p = p->next;
        printf("%d ", p->data);
    }
    printf("\n");
}

```

5.3 求链表的长度

```

int getLengthLinklist(link_node_t* p)
{
    int len = 0; //统计长度
    while(p->next != NULL)
    {
        p = p->next;
        len++; //打印一次,就计数一次
    }
    return len;
}

```

5.4 判断链表是否为空,空返回1, 未空返回0

```

//6. 判断链表是否为空 空返回1,未空返回0
int isEmptyLinklist(link_node_t* p)
{
    //p指向头节点
    //p->next代表头节点的指针域,头节点的指针域为空,说明后面什么没有了
    return p->next == NULL ? 1 : 0;
}

```

5.5 向链表的指定位置插入数据

编程思想:

1. 创建新的节点,保存插入的数据x 也就是100
2. 将头指针移动到插入位置的前一个位置
3. 将新的节点插入(先连后面,再连前面)

```

int insertIntoLinklist(link_node_t* p, int post, int x)
{
    int i;
    //0. 容错判断
    if(post < 1 || post > getLengthLinklist(p)+1)
    {
        printf("insertIntoLinklist failed!!\n");
        return -1;
    }
}

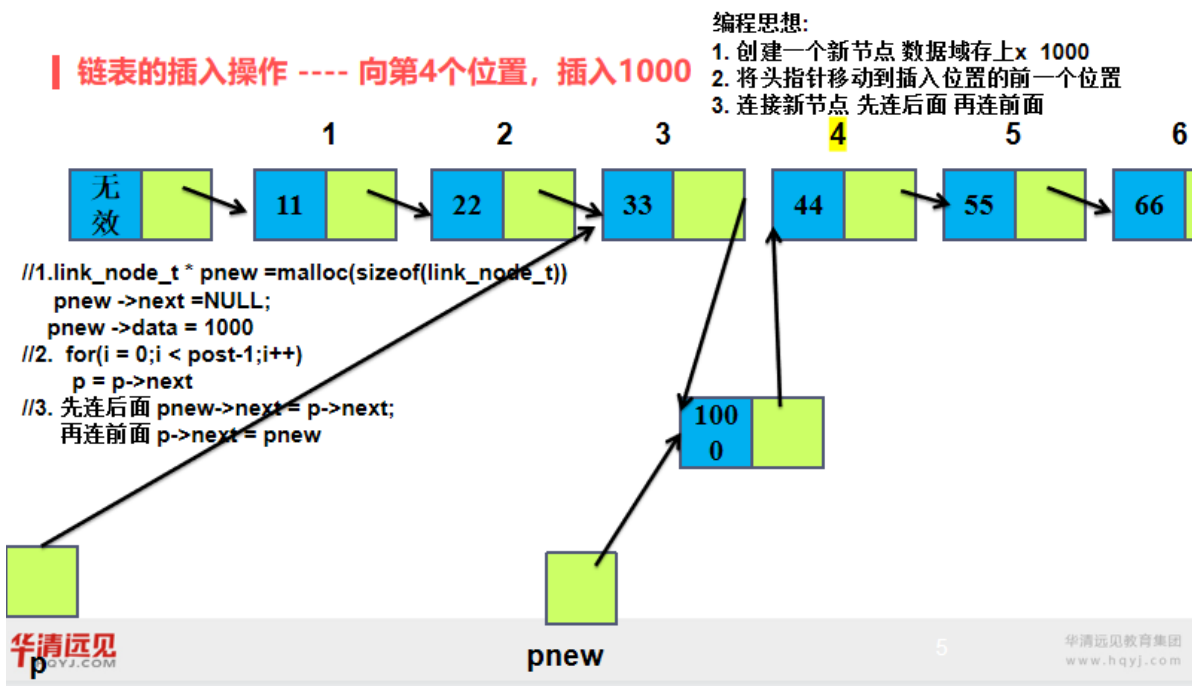
```



```

}
//1. 创建一个新的节点, 保存插入的数据x
link_node_t* pnew = malloc(sizeof(link_node_t));
if(pnew == NULL)
{
    printf("pnew malloc failed!!\n");
    return -1;
}
//申请空间, 就是为了装东西, 立刻装东西
pnew->data = x;
pnew->next = NULL;
//准备开始插入
//2. 将头指针指向(移动)插入位置的前一个位置
for(i = 0; i < post-1; i++)
    p = p->next;
//3. 将新节点插入链表(先连后面, 再连前面)
pnew->next = p->next; //连后面
p->next = pnew; //连前面
return 0;
}

```



5.6 删除链表指定位置的数据

编程思想:

1. 将头指针移动到删除位置的前一个位置
2. 定义一个指针, 指向被删除节点
3. 跨过被删除节点
4. 释放被删除节点

```

int deleteFromLinklist(link_node_t* p, int post)
{

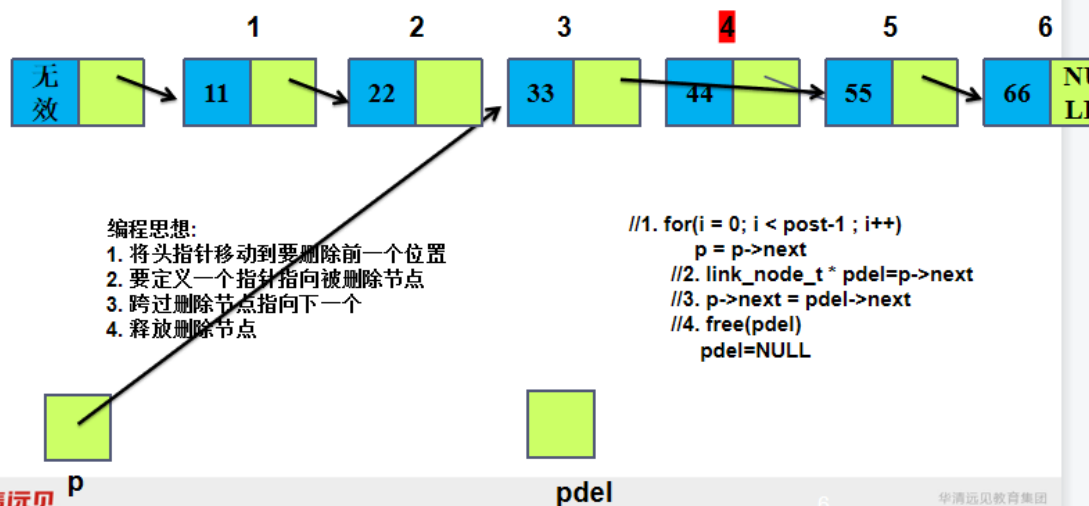
```

```

//0. 容错判断
if(post < 1 || post > getLengthLinklist(p) || isEmptyLinklist(p))
{
    printf("deleteFromLinklist failed!!\n");
    return -1;
}
//1. 将头指针移动到删除位置的前一个位置
int i;
for(i = 0; i < post-1; i++)
    p = p->next;
//2. 定义了指针变量4个字节, 指向被删除节点
link_node_t* pdel = p->next;
//3. 跨过被删除节点
p->next = pdel->next;
//4. 释放被删除节点
free(pdel)
pdel = NULL;
return 0;
}

```

单向链表的删除操作----删除第4个位置的元素



5.7 清空链表

编程思想:

1. 将头指针移动到删除位置的前一个位置 可以不写
2. 定义了指针变量4个字节, 指向被删除节点
3. 跨过被删除节点
4. 释放被删除节点

```

void clearLinklist(link_node_t* p)
{
    //砍头思想:每次删除的是头节点的下一个节点
    while(p->next != NULL) //只要链表不为空, 就执行删除操作 p->next == NULL 循环结束, 此时是空的表
    {
        //1. 将头指针移动到删除位置的前一个位置

```

```

//第一步可以省略不写,因为每次删除的是头节点的下一个节点,
//那么头节点就是每次被删除节点的前一个位置
//2.定义了指针变量4个字节,指向被删除节点
link_node_t* pdel = p->next;
//3.跨过被删除节点
p->next = pdel->next;
//4. 释放被删除节点
free(pdel);
pdel = NULL;
}
}

```

5.8 查找指定数据x出现的位置

```

int searchDataLinklist(link_node_t* p, int x)
{
    int post = 0;
    while(p->next != NULL)
    {
        p = p->next;
        post++; //用来记录第几个
        if(p->data == x)
            return post; //返回值是第几个
    }
    return -1; //不存在
}

```

5.9 尾插法

尾插法核心思想：有一个尾指针永远指向链表的尾巴(也就是最后一个节点)

```

link_node_t* h=malloc(sizeof(link_node_t));
link_node_t* ptail=h; //此时只有一个节点 即使头指针又是尾指针

```

写一个有头单向链表，一直输入学生成绩，存入链表中，直到输入-1 结束程序

每输入一个学生成绩，就malloc申请一个新的节点，将输入的成绩保存到数据域，并将该新节点链接到链表的尾巴

1. 创建一个新节点存学生成绩

```

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

typedef struct node
{
    int data; //数据域
    struct node* next; //指针域 指向下一个节点的指针
}link_node_t;

int main(int argc, const char *argv[])
{
    int score;
    //编程思想:尾插法,有一个尾指针永远指向当前链表的尾巴
}

```

```

//因为尾插法,每次都是将新节点插入在链表的最后一个节点的后面
//只需要将链表的最后一个节点的指针域指向新节点即可连接上
//创建一个头节点
link_node_t* h = malloc(sizeof(link_node_t));
if(h == NULL)
{
    printf("h malloc failed!!\n");
    return -1;
}
h->next = NULL; //指针域初始化为空,数据域无效
//创建头节点后,此时链表只有一个节点既是头节点也是尾节点
link_node_t* ptail = h; //让ptail尾指针,指向当前链表的尾巴
while(1)
{
    printf("请您输入要保存的学生成绩:\n");
    scanf("%d",&score);
    if(score == -1)
        break;
    //1.创建一个新的节点用数据域保存学生成绩
    link_node_t* pnew = malloc(sizeof(link_node_t));
    if(pnew == NULL)
    {
        printf("pnew malloc failed!!\n");
        return -1;
    }
    //2.申请空间就是为了装东西,立刻给成员变量赋值
    pnew->data = score; //数据域装成绩
    pnew->next = NULL; //指针域赋值为空
    //3.将新的节点连接在链表的尾巴上,即最后一个节点的指针域指向新的节点,ptail指向的就是最
    后一个节点
    ptail->next = pnew; //连新的节点成功
    //4.连接之后,链表变长,刚连接的新节点,是链表的最后一个节点
    //需要将尾指针指向当前链表的尾巴,需要向后移动一个位置
    ptail = pnew; //ptail = ptail->next;
}
//遍历有头单向链表

while(h->next != NULL)
{
    h = h->next;
    printf("%d ",h->data);
}
printf("\n");

```

六、单向循环链表

单向链表的最后一个节点的指针域指向链表的第一个节点,尾节点和首节点相连,此时就是一个 单向循环链表
 --- 环

结构体

5个结构体变量 link_node_t a = {11,NULL} a b c d e
 创建一个指向头结点的指针 h = &a;

5个节点形成链表

e节点的指针域指向头节点--环

遍历无头单向链表

`while()`

```
#include <stdio.h>

typedef struct node
{
    int data;          //数据域
    struct node* next; //指针域 指向下一个节点的指针
}link_node_t;

int main(int argc, const char *argv[])
{
    //无头单向链表
    //定义4个节点
    link_node_t a = {11, NULL};
    link_node_t b = {22, NULL};
    link_node_t c = {33, NULL};
    link_node_t d = {44, NULL};
    //将4个节点,通过指针域连接在一起
    a.next = &b;
    b.next = &c;
    c.next = &d;
    link_node_t* h = &a;

    //形成一个单向循环链表
    //最后一个节点的next指针,指向头节点
    d.next = &a; //形成环

    while(h != NULL)
    {
        printf("%d\n", h->data);
        h = h->next; //头指针,指向下一个节点
        sleep(1);
    }
    return 0;
}
```

```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
11
22
33
44
11
22
33
44
11
22
33
44
11
22
33
44
11
22
33
44
11
^C
linux@ubuntu:~$ vim test.c
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
```

死循环，遍历单向循环链表

6.1 约瑟夫问题

8只猴子围坐成一个圈，按顺时针方向从1到8编号。然后从1号猴子开始沿顺时针方向从1开始报数，报到m的猴子出局，再从刚出局猴子的下一个位置重新开始报数，如此重复，直至剩下一个猴子，它就是大王。

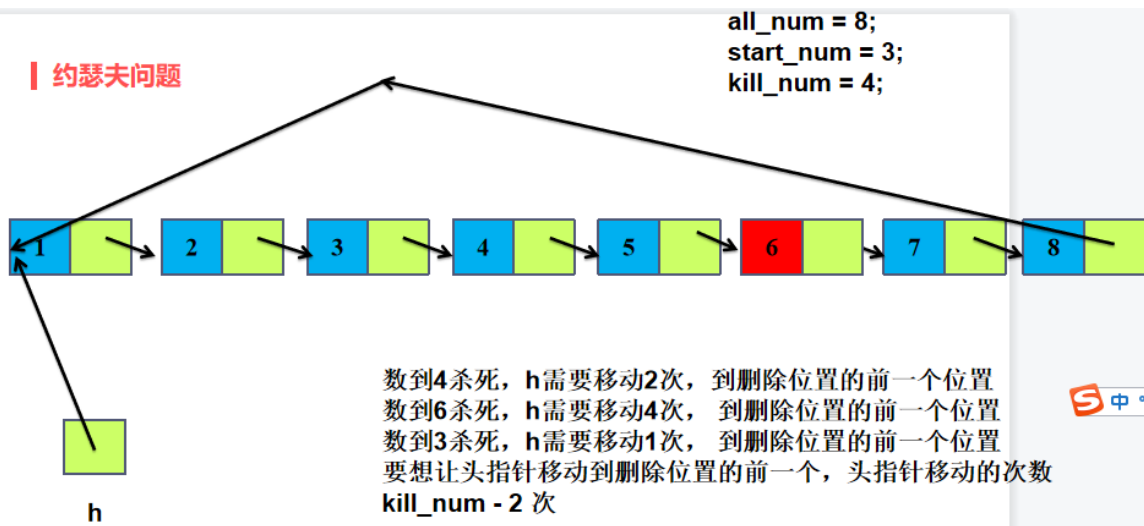
设计并编写程序，实现如下功能：

- (1) 要求由用户输入报的数m。
- (2) 给出当选猴王的编号。

1. 创建新链表 头指针h 数据域存上 1
2. 将2-8用尾插法 插入到链表上
3. 形成单向循环链表
4. 将头指针移动到开始数数的位置
5. 循环（当链表上只剩下一个节点时结束）
{
 循环杀猴 -- 删除节点
}

循环结束 猴王产生

约瑟夫问题



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

typedef struct node
{
    int data;          //数据域
    struct node* next; //指针域 指向下一个节点的指针
}link_node_t;

int main(int argc, const char *argv[])
{
    int all_num = 8; //猴子的总数
    int start_num = 3; //从几号猴子开始数
    int kill_num = 4; //数到几杀死猴子
    int i;
    printf("请您输入猴子的总数 开始报数号码 数到几杀死猴子:\n");
    scanf("%d%d%d", &all_num, &start_num, &kill_num);
    //1. 形成一个单向循环链表 8个节点, 数据域分别装的是 1 -- 8
    link_node_t* h = malloc(sizeof(link_node_t));
    if(h == NULL)
    {
        printf("h malloc failed!!\n");
        return -1;
    }
    h->data = 1; //代表的是1号猴子
    h->next = NULL;
    //尾插法
    link_node_t* ptail = h; //因为只有一个节点, 既是头节点也是尾节点
    //将2 --- 8号猴子, 用尾插法, 插入在链表的尾巴上
    for(i = 2; i <= all_num; i++)
    {
        link_node_t* pnew = malloc(sizeof(link_node_t));
        if(pnew == NULL)
        {
            printf("pnew malloc failed!!\n");
            return -1;
        }
        pnew->data = i; //i == 2 3 4 5 6 7
        pnew->next = NULL;
        ptail->next = pnew; //插入尾巴
        ptail = pnew; //ptail = ptail->next;
    }
```

```

}
//上面的程序,是一个无头的单向链表 8个节点
//形成一个单向循环链表 环
ptail->next = h;

//2. 循环杀猴
//循环杀猴之前,将头指针移动到开始报数的猴子号码节点上
for(i = 0; i < start_num-1; i++)
    h = h->next;

//循环杀猴
while(h != h->next)//当h == h->next的时候,循环结束,环上只剩下一个节点
{
    //1. 先将头指针移动到删除位置的前一个位置
    for(i = 0; i < kill_num-2; i++)
        h = h->next;
    //2. 定义一个pdel指向被删除节点
    link_node_t* pdel = h->next;
    //3. 跨过被删除节点
    h->next = pdel->next;
    printf("%d kill out!!\n",pdel->data);//调试程序,将出局猴子号码,打印删除
    //4. 释放被删除节点
    free(pdel);
    pdel = NULL;
    //删除之后,应该是从删除位置的后一个位置开始报数,需要将h指向开始报数的节点上
    //所以我们应该将头指针移动到开始报数位置,也就是删除位置的后一个
    h = h->next;
}

printf("monkey king is %d\n",h->data);
free(h);
return 0;
}

```

```

linux@ubuntu:~$ gcc linklist.c
linux@ubuntu:~$ ./a.out
请您输入猴子的总数 开始报数号码 数到几杀死猴子:
8 3 4
6 kill out!!
2 kill out!!
7 kill out!!
4 kill out!!
3 kill out!!
5 kill out!!
1 kill out!!
monkey king is 8

```

七、双向链表操作


```
//双向链表节点的定义
typedef struct node
{
    int data;
    struct node * next;//指向下一节点
    struct node * pri;//指向的前一个节点
}
```

7.1 插入操作

插入思想:

0. 容错判断
1. 创建新的节点保存插入的数据x
2. 将头指针移动到插入位置节点
3. 进行插入操作(先连前面, 再连后面)

//插入操作

//先连前面

```
p->pri->next = pnw;
```

```
pnw->pri = p->pri;
```

//再连后面

```
pnw->next = p;
```

```
p->pri = pnw;
```

7.2 删除操作

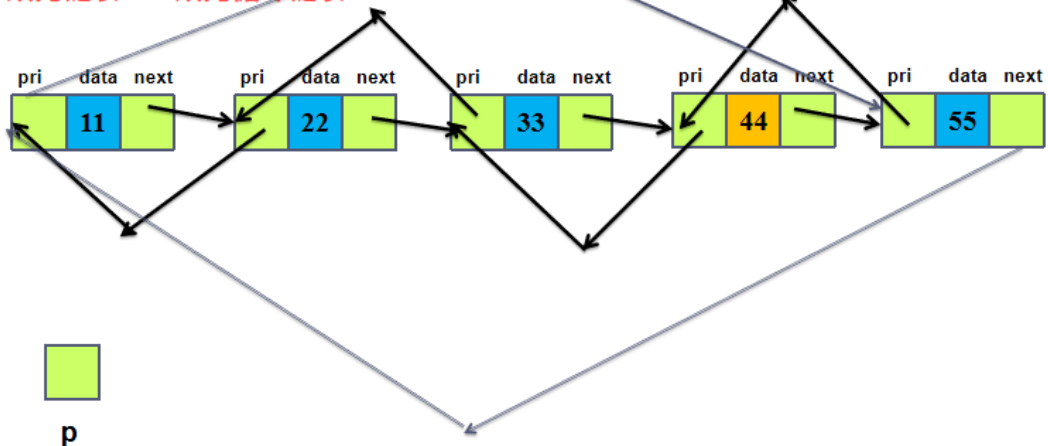
0. 容错判断
1. 将头指针移动到删除位置节点
2. 进行删除操作

```
p->pri->next = p->next;
```

```
p->next->pri = p->pri;
```

7.3 双向循环链表

双向链表 --- 双向循环链表



八、链表的总结

顺序表：缺点 长度固定 插入和删除麻烦 优点 查找方便 有下标

链表优点：

1. 插入和删除方便
2. 不必实现估计存储，因为链表不存在表满的情况

链表缺点：

1. 查找麻烦，不能实现随机访问

九、面试问题：

请你说一下，顺序表和链表的区别？？（等价于数组和链表的区别??）

相同：顺序表和链表，都属于线性表，逻辑结构是线性结构

不同：

1. 顺序表存储结构，内存当中是 连续 存储的，顺序存储结构
链表存储结构，内存当中是 不连续 存储的，链式存储结构，通过指针将节点连接在一起
2. 顺序表 长度是固定， 链表长度 不固定
3. 顺序表查找方便，插入删除麻烦 链表插入删除方便，查找麻烦

@作业1：单向链表的操作

建立一个空的.c文件 将有头单向链表的8个函数盲打

[单向链表的操作](#) (有头的单向链表)

- //1. 创建一个空的链表
- //2. 向链表的指定位置插入数据
- //3. 遍历链表的有效元素
- //4. 判断链表是否为空,空返回1, 未空返回0
- //5. 删除链表指定位置的数据
- //6. 求链表的长度
- //7. 查找指定数据，出现在链表中的位置
- //8. 清空链表

@复习

1. 链表结构体定义

```
typedef struct node
{
    int data; //数据域
    struct node* next; //指针域 指向下一个节点的地址
}link_node_t;
```

2. 有头链表 -- 第一个节点数据是无效

遍历有头链表

```
while(h->next != NULL)
{
    h = h->next;
    printf("%d ", h->data);
}
```

```
}
```

3. 无头单向链表 --所有数据域都是有效

```
while(h != NULL)
{
    printf("%d ",h->data);
    h = h->next;
}
```

4. 单向链表的操作

1. 是否为空
2. 遍历
3. 求长度
4. 创建空链表
5. 插入

编程思想:

创建一个新节点用来保存数据

将头指针移动到插入位置的前一个位置上

链接新节点 先连接后面 再连接前面

```
int insertLinklist(link_node_t * p,int post,int x)
{
    //0. 容错判断
    link_node_t * pnew = malloc(sizeof(link_node_t));
    if(pnew == NULL)
    {
        return -1;
    }
    //放数据
    pnew->data = x;
    pnew->next = NULL;

    //移动头指针到插入位置前一个位置
    int i;
    for(i = 0;i <post-1;i++)
        p = p->next;
    //先后再前

    pnew->next = p->next;
    p->next = pnew;
    return 0;
}
```

6. 删除

编程思想: 把头指针移动到删除位置的前一个位置

定义一个指针指向被删除节点

跨过删除节点

释放删除节点

```
int delete(link_node_t *p,int post)
{
    int i ;
    for(i = 0;i <post-1;i++)
        p= p->next;
    link_node_t * pdel = p->next;
    p->next = pdel->next;
    free(pdel);
    pdel=NULL;
    return 0;
}
```

7.清空 砍头思想

编程思想:

定义一个指针指向被删除节点

跨过删除节点

释放删除节点

```
void clearlinklist(link_node_t *p)
{
    while(p->next != NULL)
    {
        p = p->next;
        link_node_t * pdel = p->next;
        p->next = pdel->next;
        free(pdel);
        pdel=NULL;
    }
}
```

Day03

一. 栈

1. 什么是栈?

只能在一端插入和删除操作的线性表
插入和删除操作这端我们叫栈顶

2. 栈特点

栈特点:

先进后出 后进先出

FILO LIFO

F --frist

I -- in

L -- last

O -- out

记住栈特点: 往杯子里放大饼 1 --4 4 3 2 1

3. 栈实现方式

stack //栈

栈是解决问题的一种思想 先进后出

栈分为: 顺序栈和链式栈

顺序栈和链栈最大的区别是什么???

顺序栈长度固定, 链栈长度不固定

存储结构不同

用数组去实现顺序栈：本质上就是操作顺序表

4. 顺序栈的结构体定义

顺序表结构体定义

```
#define N 5
typedef struct
{
    int a[N];
    int last; // 数组有效元素个数
} seqlist_t;
```

顺序栈结构定义

```
#define N 5
typedef struct
{
    int a[N];
    int top; // 栈针 当数组下标用 top 的值时刻代表有效元素个数
} stack_t;

// 为什么 top 称为栈针
```

5. 顺序栈操作

- | | |
|------------|---|
| 1. 创建一个空的栈 | === 创建一个空的顺序表 <code>createEmptyStack</code> |
| 2. 入栈 | === 在顺序表的尾巴上插入一个数据 有效元素个数+1 <code>pushStack()</code> |
| 3. 出栈 | === 删除顺序表的尾巴 尾巴里面的数据返回 有效元素个数-1 <code>popStack()</code> |
| 4. 判断栈是否为满 | === 判断顺序表是否未满 <code>isFullStack()</code> |
| 5. 判断栈是否为空 | === 判断顺序表是否为空 <code>isEmptyStack()</code> |
| 6. 获取栈顶元素 | === 将顺序表的尾巴节点里面的数据域返回 <code>getTopStack()</code> |
| 7. 清空栈 | === 清空顺序表 <code>clearEmptyStack()</code> |

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

#define N 5
typedef struct
{
    int a[N];
    int top; // top 栈针, 当做数组下标使用, top 的值时刻代表栈中有效元素的个数
} stack_t;

// 1. 创建一个空的栈
stack_t* createEmptyStack()
{
    stack_t* p = malloc(sizeof(stack_t));
    if(p == NULL)
    {
        printf("createEmptyStack malloc failed!!\n");
        return NULL;
    }
}
```

```

    p->top = 0; //因为是空的栈,所以有效元素个数为0
    return p;
}
//2. 判断栈是否为满,满返回1,未满返回0
int isFullStack(stack_t* p)
{
    return p->top == N ? 1 : 0;
}
//3. 入栈,是插入,在数组尾巴插入一个数据
//push 推
int pushStack(stack_t* p, int x)
{
    //0. 容错判断
    if(isFullStack(p))
    {
        printf("isFullStack!!!\n"); //栈满,入栈失败
        return -1;
    }
    //1. 入栈
    p->a[p->top] = x;
    //2. 将入栈的元素变为有效,有效元素个数+1
    p->top++;
    return 0;
}
//4. 判断是否为空 1 代表空 0代表未空
int isEmptyStack(stack_t* p)
{
    return p->top == 0 ? 1 : 0;
}
//5. 出栈,删除,在数组的尾巴元素删除,同时将删除的数据值返回
int popStack(stack_t* p)
{
    //0. 容错判断
    if(isEmptyStack(p))
    {
        printf("isEmptyStack!!!\n"); //空栈,出栈失败
        return -1;
    }
    //1. 出栈,将数组的最后一个有效元素删除
    p->top--; //有效元素个数-1,相当于将栈顶元素删除
    //2. 将出栈元素的值返回
    //上面的p->top--之后,p->top里面保存的是出栈元素的下标
    return p->a[p->top];
}
//6. 获取栈顶元素的值
int getTopValue(stack_t* p)
{
    return p->a[p->top-1]; // p->top-1得到最后一个有效元素的下标,即栈顶元素的下标
}
//7. 清空栈
void clearStack(stack_t* p)
{
    p->top = 0; //有效元素赋值为0
}

//8. 显示一个数的二进制
void showBin(stack_t * p, int num)
{

```

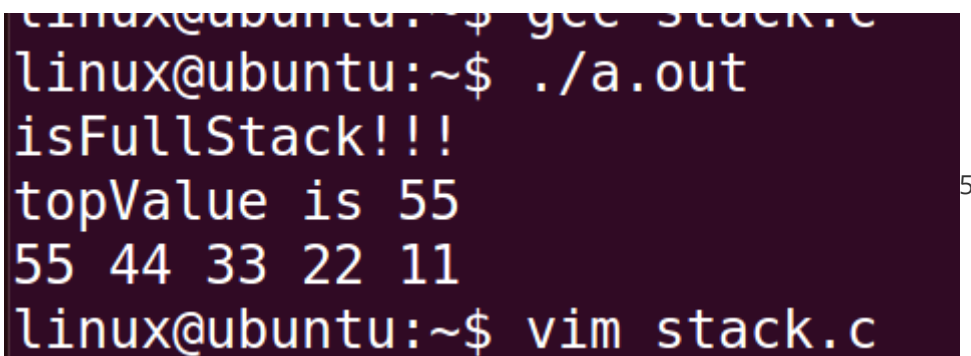
```

//1.求出的二进制全部入栈
while(num != 0)
{
    pushStack(p,num%2);
    num/=2;
}

//2.将所有的二进制全部出栈打印
while(!isEmpty(p))
{
    printf("%d",popStack(p));
}
printf("\n");
}

int main(int argc, const char *argv[])
{
    stack_t* p = createEmptyStack();
    pushStack(p, 11);
    pushStack(p, 22);
    pushStack(p, 33);
    pushStack(p, 44);
    pushStack(p, 55);
    pushStack(p, 66);//入栈失败,N 是 5, 栈满,第6个入栈失败
    printf("topValue is %d\n",getTopValue(p));//获取栈顶元素的值,并不是将栈顶元素删除
    while(!isEmptyStack(p))//只要栈不为空,就进行出栈操作
    {
        printf("%d ",popStack(p));//popStack函数返回值,就是出栈元素的值,全部出栈,打印
    }
    //入栈的顺序11 22 33 44 55,出栈顺序 55 44 33 22 11    先进后出
    printf("\n");
    return 0;
}

```



```

linux@ubuntu:~$ gcc stack.c
linux@ubuntu:~$ ./a.out
isFullStack!!!
topValue is 55
55 44 33 22 11
linux@ubuntu:~$ vim stack.c

```

#练习1:

定义一个函数,将其转换为二进制数,将二进制位存储到栈内,再出栈打印输出
 输入10, 打印输出1010
 输入15, 打印输出1111

```
linux@ubuntu:~$ vim stack.c
linux@ubuntu:~$ gcc stack.c
linux@ubuntu:~$ ./a.out
1010 10的二进制
linux@ubuntu:~$ vim stack.c
linux@ubuntu:~$ gcc stack.c
linux@ubuntu:~$ ./a.out
1111111 255的二进制
linux@ubuntu:~$ gedit stack.c
linux@ubuntu:~$
```

复习

栈: stack

只能在一端进行插入和删除操作的线性表 -- 栈顶

先进后出 后进先出

`typedef struct`

{

`int a[N];`

`int top`; 栈针 `a[top]` 时刻代表有效元素的个数

`}stack_t;`

1. 入栈 insert

`int pushStack(stack_t *p, int x)`

{

 // 栈满了不能入

`p->a[p->top] = x;`

`p->top++;`

}

2. 出栈 delete

`int outStack(stack_t *p)`

{

 // 栈空了不能出

`p->top--;`

`return p->a[p->top];`

}

二. 队列

1. 什么是队列


```
stack //栈
queue //队列
```

只能在两端进行插入和删除操作的线性表，在队头进行删除操作 在队尾进行插入操作

2. 队列的特点

先进先出，后进后出
FIFO LIFO

3. 队列实现方式

队列是解决问题的一种思想 先进先出

队列可以用数组实现，用数组实现，队列称为顺序队列(循环队列)，顺序存储结构
队列可以用链表实现，用链表实现，队列称为链式队列(链队列)，链式存储结构

顺序队列和链队列最大的区别是什么？
存储结构不同

4. 循环队列结构体定义

```
queue// 队列
#define N 5
typedef struct
{
    int a[N];
    int rear;// 后 队尾 入队 插入 用rear当下标 a[rear]
    int front;//前 队头 出队 删除 用front当下标 a[front]
}queue_t;
rear 指向尾巴的下一个位置
front 始终是指向队首元素
```

5. 循环队列操作

```
//rear永远指向队列的头
//front永远指向队列尾巴的后一个位置
typedef struct
{
    int a[N];
    int front; //队头，删除用front当下标
    int rear; //队尾，插入用rear当下标
}queue_t;
```

1. 创建一个空的队列 `createEmptyQueue()`
2. 入队 `inQueue()`
3. 出队 `outQueue()`
4. 判断队列是否为满 满返回值是1，未满是0 `isFullQueue()`
5. 判断队列是否为空，空返回值是1，未空是0 `isEmptyQueue()`

6. 求队列的长度 `getLengthQueue()`

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

#define N 5
typedef struct
{
    int a[N];
    int rear; //后 队尾,在队尾插入,入队的时候,用rear当做下标
    int front; //前 队头,在队头删除,出队的时候,用front当做下标
    //rear 永远指向队列尾元素的后一个位置,方便插入
    //front 永远指向对头元素,方便删除
}queue_t;

//1.创建一个空的队列
queue_t* createEmptyQueue()
{
    queue_t* p = malloc(sizeof(queue_t));
    if(p == NULL)
    {
        printf("createEmptyQueue malloc failed!!\n");
        return NULL;
    }
    //只要p->rear == p->front 就是空的队列,是几不重要
    //但是p->rear 和 p->front的值,要在数组的下标范围内 0 ---- N-1
    p->rear = p->front = 3; //将3赋值给front,在赋值给rear,让rear和front都得3
    return p;
}

//2.判断队列是否为满 满返回1,未满返回0
int isFullQueue(queue_t* p)
{
    //因为p->rear == p->front代表队列空,所以我们只能浪费一个存储位置来判断队列是否为满,
    //提前判断p->rear+1的位置是否 == p->front,来判断是否是满队列
    return (p->rear+1) % N == p->front ? 1 : 0;
}

//3.入队,在队列尾巴进行插入操作
int inQueue(queue_t* p, int x)
{
    //0.容错判断
    if(isFullQueue(p))
    {
        printf("isFullQueue!!\n");
        return -1;
    }
    //1.入队列,用rear当做下标
    p->a[p->rear] = x;
    //2.p->rear++,将出入的数据x,视为有效的元素
    p->rear = (p->rear+1) % N; // %N避免p->rear+1出现数组越界
    //p->rear = (p->rear+1) % N;此行代码等价于 p->rear++; p->rear = QQ p->rear % N;
}

//4.判断队列是否为空,空返回1,未空返回0
int isEmptyQueue(queue_t* p)
{
    return p->rear == p->front ? 1 : 0;
}

//5.出队列,在队列的头进行删除操作
```

```

int outQueue(queue_t* p)
{
    //0. 容错判断
    if(isEmptyQueue(p))
    {
        printf("isEmptyQueue!!\n");
        return -1;
    }
    //出队,用front当做数组的下标
    //1. 将即将出队的元素,临时存储到变量x中
    //因为front永远指向队头的元素
    int x = p->a[p->front];
    //2. 让出队的元素变为无效元素
    p->front = (p->front+1) % N;
    //3. 将出队元素的值返回
    return x;
}

//6. 求队列的长度
int getLengthQueue(queue_t* p)
{
    //按道理, rear值肯定大于front
    //方法一
    if(p->rear >= p->front)
        return p->rear - p->front;
    else // rear < front 按道理rear肯定大于front 为什么rear < front, 因为%N, 所以+N, 还原rear的值
        return p->rear + N - p->front;
    //方法二
    //return (p->rear + N - p->front) % N;
}

int main(int argc, const char *argv[])
{
    queue_t* p = createEmptyQueue();
    inQueue(p, 11);
    inQueue(p, 22);
    inQueue(p, 33);
    inQueue(p, 44);
    inQueue(p, 55); //入队失败, 因为数组的元素个数是N, 最多存储N-1个数据
    printf("len is %d\n", getLengthQueue(p));
    while(!isEmptyQueue(p)) //只要队列不为空, 就执行出队操作
    {
        printf("%d ", outQueue(p));
    }
    printf("\n");
    //入队的顺序是 11 22 33 44, 出队的顺序11 22 33 44 先进先出

    return 0;
}

```

```
linux@ubuntu:~$ ./a.out
isFullQueue!!
len is 4
11 22 33 44
```

Day04

一. 时间复杂度

时间复杂度只不过是对于算法运行时间和处理问题规模之间,关系的一种估算描述
一个算法的时间复杂度越高,那么也就说明这个算法在处理问题的时候所花费的时间也就越长

算法的可执行语句重复执行的频度和

语句频度: 算法中可执行语句重复执行的次数

通常时间复杂度用一个问题规模函数来表达

$T(n) = O(f(n))$

O 时间度量级 根据算法中语句执行的最大次数(频度)来 估算一个算法执行时间的数量级。

$f(n)$ 函数关系表达式

它表示随问题规模 n 的增大,算法执行时间的增长率和 $f(n)$ 的增长率相同。

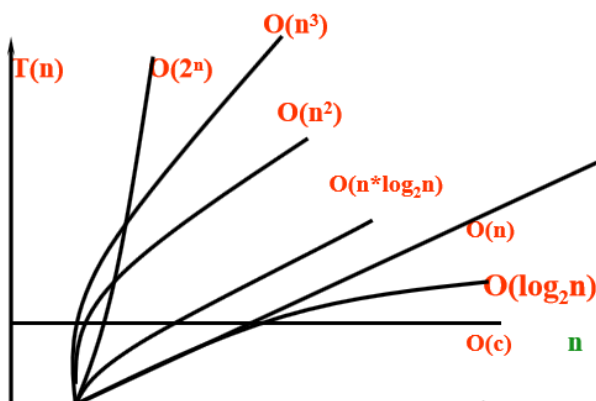
n 代表研究问题的规模 `int a[10000]`进行冒泡排序 算法 10000

► $T(n)$ 的量级通常有:

- $O(c)$ ——常数级,不论问题规模多大, $T(n)$ 一致,因而是理想的 $T(n)$ 量级;
- $O(n)$ ——线性级; $O(n^2), O(n^3)$ ——平方、立方级;
- $O(\log_2 n), O(n \cdot \log_2 n)$ ——对数、线性对数级;
- $O(2^n)$ ——指数级,时间复杂度最差。

以上几种常见的 $T(n)$ 随 n 变化的增长率如图7所示。

图7



时间复杂度排序

$O(c) > O(\log_2 n) > O(n) > O(n \cdot \log_2 n) > O(n^2) > O(n^3) > O(2^n)$

#案例1

求 $1+2+3+\dots+100$ 的和

//算法一

```
int sum = 0;
int n = 100;
for(i = 1; i <= n; i++)
    sum += i;
printf("sum is %d\n",sum);
```

//求和的时间复杂度： $T(n) = O(n)$

时间复杂度：运行时间和问题规模 n 之间的关系

$n = 100$ ----> 100次

$n = 88$ ----> 88次

$n = 10$ ----> 10次

找到问题规模 n 与重复执行次之间的关系式子

重复执行的次数 = n

$T(n) = O(f(n))$

$f(n) = n$; //先写出问题规模 n 与 重复执行次数的关系式子

$T(n) = O(n)$

//算法二 高斯定理

(首项+尾项) * 项数

2

```
int n = 100;
int sum = (1+n)*n / 2;
printf("sum is %d\n",sum);
n=100  1
n=88   1
n= 10  1
T(n) = O(1)
```

#案例2

```

int i,j;
for(i = 0; i < n; i++) //外循环 循环 n 次
{
    for(j = 0; j < n; j++)//内循环 循环 n 次
    {
        printf("hello world!!");
    }
}
n*n==n^2
T(n)=O(n^2)

```

#案例3

```

int i,j;
for(i = 0; i < n; i++)
{
    for(j = 0; j <= i; j++)
    {
        printf("hello world!!");
    }
}

```

i == 0; 循环 1次
i == 1; 循环 2次
i == 2; 循环 3次
 ...
 ...
i = *n*-1; 循环 *n*次

循环次数 = 1 + 2 + 3 + *n*

```

f(n) = (1+n)*n / 2;
f(n) = (1/2)*n^2 + (1/2)*n
//只保留最高项，其它项舍去， 只保留指数最高的项
(1/2)*n^2 ,因为最高项是n^2
//如果最高项系数不为1，将其置为1
T(n) = O(n^2)

```

时间复杂度简化方法

计算大O的方法

- (1)根据问题规模*n*写出表达式 $f(n)$
- (2)如果有常数项，将其置为1
- (3)只保留最高项，其它项舍去 //只保留指数最高的项
- (4)如果最高项系数不为1，将其置为1

二. 查找方法

1. 顺序查找

//查找x，出现在数组的位置的下标

```
int findByOrder(int* p, int n, int x)
{
}
```

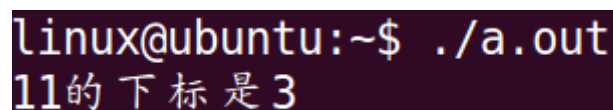
- (1) main函数中定义一个数组 `int a[8] = {12,34,2,11,32,123,1,2};`
- (2) 定义一个函数，查找指定数据
- (3) 如果找到了，返回它在数组中的位置下标即可，未找到返回-1
- (4) main函数中测试

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

int findByOrder(int* p, int n, int x)
{
    int i;
    for(i = 0; i < n; i++)
    {
        if(p[i] == x)
            return i;
    }

    return -1;
}

int main(int argc, const char *argv[])
{
    int a[8] = {12,34,2,11,32,123,1,2};
    int ret = findByOrder(a, 8, 11);
    if(ret == -1)
    {
        printf("没有找到!!\n");
    }
    else
    {
        printf("%d的下标是%d\n", a[ret], ret);
    }
    return 0;
}
```



```
linux@ubuntu:~$ ./a.out
11的下标是3
```

顺序查找缺点

当数据量大的时候，查找速度慢

2. 二分查找

二分查找(分半查找、拆半查找、二分搜索)

2.1 使用前提条件

二分查找的数组必须是一个有序序列(可以是递增、也可以是递减)

2.2 算法思想

假设被查找的数据是 x ，每次用 x 与中间位置做比较，判断 x 在中间位置的左侧还是右侧继续缩小查找范围，同样的操作

2.3 时间复杂度

二分查找时间复杂度： $T(n) = O(\log n)$

循环的次数 与 问题规模 n 的关系式子

问题规模 n

查找数据长度为 n ，每次查找后减半，

第一次 $n/2$ $n/2^1$

第二次 $n/2/2$ $n/2^2$

第三次 $n/2/2/2$ $n/2^3$

...

第 k 次 $n/2^k$

最坏的情况下第 k 次才找到，此时只剩一个数据，长度为1。

即 $n/2^k = 1$ // $2^k = n$; 求 $k = \log_2 n$

查找次数 $k = \log n$

2.4 案例代码

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

int findByHalf(int* p, int n, int x)
{
    int low = 0;
    int high = n-1;
    int middle; //用来保存中间位置下标
    while(low <= high) //low > high 循环结束
    {
        //1. 得到中间位置的下标
        middle = (low + high) / 2;
        //2. x与中间位置的元素做比较, 判断在middle位置的左边还是右边, 来缩小范围
        if(x == p[middle])
            return middle; //找到了
        else if(x > p[middle]) //说明在右侧, 需要移动low
            low = middle + 1;
        else //x < p[middle] //说明在左侧, 需要移动high
            high = middle - 1;
    }
    //上面的循环结束后, 如果都没有执行return middle, 说明没有找到
    return -1;
}

int main(int argc, const char *argv[])
```



```

{
    int a[8] = {11,22,33,40,55,60,70,89};
    int ret = findByHalf(a, 8, 55);
    if(ret == -1)
    {
        printf("没有找到!!\n");
    }
    else
    {
        printf("%d的下标是%d\n",a[ret],ret);
    }
    return 0;
}

```

```

linux@ubuntu:~$ ./a.out
55的下标是4

```

3. 哈希表

有一张表，保存了数据中关键字与对应的存储位置的关系

在选择key(关键字)的时候，要选择数据中不重复的关键字作为key

存时按照对应关系存

取时按照对应关系取

3.1 直接地址法

保存全国各个年龄的人口数

输入年龄和人口数，按照对应关系保存到哈希表中，

输入要查询的年龄，打印输出该年龄的人口数

2 100 //2 岁人口是 是 100

19 200 //19 岁人口数是 200

//哈希表是个数组

int hash_list[150]; //假设人能活的最大年龄150岁

//元素int类型，因为哈希表中保存的是 人口数

对应关系

1岁的人口数 -----> hash_list[0]

2岁的人口数 -----> hash_list[1]

...

...

100岁人口数 -----> hash_list[99]

关键字 age 存储位置 age-1

对应关系age - 1

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

```

//哈希函数：保存的是关键字与存储位置的关系
int hashFun(int key)
{
    int post = key - 1; //age - 1得到的是哈希表中的存储位置
    return post;
}

//存数据
//int* p,用来保存哈希表的首地址
void saveAgeNum(int* p, int age, int num)
{
    //存的时候按照对应关系存储
    //1.调用哈希函数,得到哈希表中的存储位置
    int post = hashFun(age);
    //2.有了存储位置,将人口数,保存到哈希表中
    p[post] = num;
}

//取数据
int getAgeNum(int* p, int age)
{
    //取的时候按照对应关系取
    //1.调用哈希函数,得到age对应的人口数哈希表中的存储位置
    int post = hashFun(age);
    //2.将人口数返回
    return p[post];
}

int main(int argc, const char *argv[])
{
    int hash_list[150] = { 0 };
    int age,num;
    int i;
    for(i = 0; i < 5; i++)
    {
        printf("请输入年龄 和 对应的人口数:\n");
        scanf("%d%d",&age,&num);
        saveAgeNum(hash_list ,age, num);
    }
    for(i = 0; i < 3; i++)
    {
        printf("请输入您要查找的年龄:\n");
        scanf("%d",&age);
        printf("%d年龄的人口数是%d万个!!\n",age, getAgeNum(hash_list,age));
    }
    return 0;
}

```

```
linux@ubuntu:~$ ./a.out
请输入年龄 和 对应的人口数：
19 190
请输入年龄 和 对应的人口数：
18 180
请输入年龄 和 对应的人口数：
17 170
请输入年龄 和 对应的人口数：
1 200
请输入年龄 和 对应的人口数：
100 5
请输入您要查找的年龄：
18
18年龄的人口数是180万个!!
请输入您要查找的年龄：
17
17年龄的人口数是170万个!!
请输入您要查找的年龄：
1
1年龄的人口数是200万个!!
```

3.2 叠加法

将图书馆的条形码叠加

```
int a[] = {321432543, 432321543, 654657345, 234456300, 213342, 123453333};
```

```
321432543
```

```
321
```

```
432
```

```
543
```

```
求和
```

```
hash_list[1000];
```

```
int hashFun(int key)
{
    int post = (key/1000000 + key/1000%1000 + key%1000) % 1000
    return post;
}
```

```

#include <stdio.h>

typedef struct
{
    int number; //用来保存图书条形码
    char book_name[20];
}book_t;

int hashFun(int key)
{
    int post = (key / 1000000 + key / 1000 % 1000 + key % 1000) % 1000;
    return post;
}

//保存图书信息
//book_t* p //保存哈希表的首地址
//book_t* q //保存一本图书的首地址
void saveBookInfo(book_t* p, book_t* q)
{
    //存的时候按照对应关系存
    //1.调用哈希函数,得到存储位置
    int post = hashFun(q->number);
    p[post] = *q; //q里面保存的是main函数中结构体变量b的地址,所以*q代表的就是main函数中的b
}

//通过条形码,查找图书信息
//book_t* p //用来保存哈希表的首地址
book_t* getBookInfo(book_t* p, int number)
{
    //取的时候按照对应关系取
    //1.调用哈希函数,得到存储位置
    int post = hashFun(number);
    //2.将图书信息返回
    return &p[post]; // return p+post;
}

int main(int argc, const char *argv[])
{
    book_t b;
    book_t* p = NULL;
    int number;
    book_t hash_list[1000]; //元素类型是book_t因为保存的是图书信息
    //图书条形码
    //输入3本图书信息,将图书信息保存到哈希表中
    int i;
    for(i = 0; i < 3; i++)
    {
        printf("请输入条形码 和 图书编号:\n");
        scanf("%d%s",&b.number, b.book_name);
        saveBookInfo(hash_list, &b);
    }
    //输入3个条形码,查找图书的信息
    for(i = 0; i < 3; i++)
    {

```

```

    printf("请输入条形码:\n");
    scanf("%d",&number);
    p = getBookInfo(hash_list, number);
    printf("条形码:%d 书名:%s\n",p->number, p->book_name);
}
return 0;
}

```

```

linux@ubuntu:~$ ./a.out
请输入条形码 和 图书编号:
123432452 阿三日记
请输入条形码 和 图书编号:
234234322 李四回忆录
请输入条形码 和 图书编号:
456785234 狂飙
请输入条形码:
234234322
条形码:234234322 书名:李四回忆录
请输入条形码:
123432452
条形码:123432452 书名:阿三日记
请输入条形码:
456785234
条形码:456785234 书名:狂飙

```

3.3 数字分析法

```

//k1 k2 k3 k4 k5 k6
//2 3 1 5 8 6
//2 4 2 3 4 6
//2 3 3 7 9 6
//2 3 9 8 8 6
//2 4 5 7 8 6
//2 3 4 2 9 6
//通过数字分析，只有中间两位数重复的次数最少，所以将中间两个字作为关键字key

```

```

//哈希函数
int hashFun(int key)
{
    int post = key % 10000 / 100;
    return post;
}
//将数据存储
void saveNum(int *hash_list,int key)
{

```

```

    int post = hashFun(key);
    hash_list[post] = key;
}
//将数据取出
int getNum(int *hash_list,int key)
{
    int post = hashFun(key);
    return hash_list[post];
}

int main(int argc, const char *argv[])
{
    //k1 k2 k3 k4 k5 k6
    //2  3  1  5  8  6
    //2  4  2  3  4  6
    //2  3  3  7  9  6
    //2  3  9  8  8  6
    //2  4  5  7  8  6
    //2  3  4  2  9  6
    int i;
    int a[] = {231586,242346,233796,239886,245786,234296};
    //创建哈希表
    int hash_list[100]; //100因为只取中间两位所以post为两位数
    //将数据存入哈希表
    for(i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        saveNum(hash_list,a[i]);
    //查找数据
    for(i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        printf("post:%d --- %d\n",hashFun(a[i]),getNum(hash_list,a[i]));
    return 0;
}

```

3.4 平方取中法

当取key中的某些值，不能是记录均匀分布时，根据数学原理，对key进行key的2次幂（取平方）
取key平方中的某些位可能会比较理想

key key的平方 H(key)

0100	00 100 00	100
0110	00 121 00	121
1010	10 201 00	201
1001	10 020 01	020
0111	00 123 21	123

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

```

// key    key的平方    H(key)
// 0100    00 100 00    100
// 0110    00 121 00    121
// 1010    10 201 00    201
// 1001    10 020 01    020
// 0111    00 123 21    123
//对key平方后，发现中间的三位重复次数最少

```

```

//哈希函数
int hashFun(int key)
{
    int post = key*key % 100000 / 100;
    return post;
}
//存储数据
void saveNum(int *hash_list, int key)
{
    int post = hashFun(key);
    hash_list[post] = key;
}
//取数据
int getNum(int *hash_list,int key)
{
    int post = hashFun(key);
    return hash_list[post];
}

int main(int argc, const char *argv[])
{
    int i;
    int a[] = {100,110,1010,1001,111}; //key
    int hash_list[1000] = { 0 };//哈希表
    for(i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        saveNum(hash_list,a[i]);
    for(i = 0; i < sizeof(a)/sizeof(a[0]); i++)
        printf("post:%3d --- %d\n",hashFun(a[i]),getNum(hash_list,a[i]));
    return 0;
}

```

3.5 保留余数法

```

int a[11] = {12,23,4,24,2,4,23,1,24,456,23}
int hash_list[15];//哈希表长度是15，最大的质数是13
数据个数为n
n = 11

a // 装填因子
哈希表的长度 m = n/a //n存储数据的个数 a的为装填因子，0.7-0.8之间最为合理
m = 11 / 0.75 == 15 0-15之间最大的质数为13
//prime 质数 为 不大于哈希表长的质数
//保留余数法
int hashFun(int key) //将余数作为存储位置
{
    int post = key % prime; //key % 13
    return post;
}

```

5. 哈希冲突解决

见PPT讲解

5.1 线性探查法

开放地址法

例9 设记录的key集合 $k=\{23, 34, 14, 38, 46, 16, 68, 15, 07, 31, 26\}$, 记录数 $n=11$ 。令装填因子 $\alpha=0.75$, 取表长 $m=\lceil n/\alpha \rceil=15$ 。用“保留余数法”选取Hash函数 ($p=13$):

$$H(\text{key})=\text{key}\%13$$

采用“线性探查法”解决冲突。依据以上条件, 依次取 k 中各值构造的Hash表HT, 如下图所示

(表HT初始为空) $k=\{23, 34, 14, 38, 46, 16, 68, 15, 07, 31, 26\}$

▶ $H(\text{key})=\text{key}\%13$; $H_i=(H(\text{key})+d_i)\%15$; $d_i=1, 2, 3, \dots, (m-1)$

HT:	26	14	15	16	68	31	^	46	34	07	23	^	38	^
H(key)	0	1	2	3	4	5	6	7	8	9	10	11	12	13

$H(68)=68\%13=3$ (冲突), 取 $H_1=(3+1)\%15=4$ (空), 故68存入4单元。

$H(07)=7\%13=7$ (冲突), 取 $H_1=(7+1)\%15=8$ (冲突), 取 $H_2=(7+2)\%15=9$ (空), 故07存入9单元。

若采用二次探测法: $d_i=1^2, -1^2, 2^2, -2^2, \dots$, 表为:

HT:	26	14	15	16	68	31	07	46	34	^	23	^	38	^
H(key)	0	1	2	3	4	5	6	7	8	9	10	11	12	13

其中, $H(07)=7\%13=7$ (冲突), 取 $H_1=(7+1^2)\%15=8$ (冲突), 取 $H_2=(7-1^2)\%15=6$ (空), 故07存入6单元。

华清远见

华清远见教育

5.2 链地址法

```
typedef struct node
{
    int data;
    struct node* next;
}link_node_t;

link_node_t* h1 = createEmptyLinklist();
link_node_t* h2 = createEmptyLinklist();
link_node_t* h3 = createEmptyLinklist();

link_node_t* hash_list[3];
hash_list[0] = createEmptyLinklist();
hash_list[1] = createEmptyLinklist();
hash_list[2] = createEmptyLinklist();
```

link_node_t* hash_list[13];

链地址法

发生冲突时, 将各冲突记录链在一起, 即同义词的记录存于同一链表。

设 $H(\text{key})$ 取值范围(值域)为 $[0, m-1]$, 建立头指针向量 $HP[m]$, $HP[i]$ ($0 \leq i \leq m-1$) 初值为空。凡 $H(\text{key})=i$ 的记录都链入头指针为 $HP[i]$ 的链表。

例10 设 $H(\text{key})=\text{key}\%13$,

其值域为 $[0, 12]$, 建立指针向量

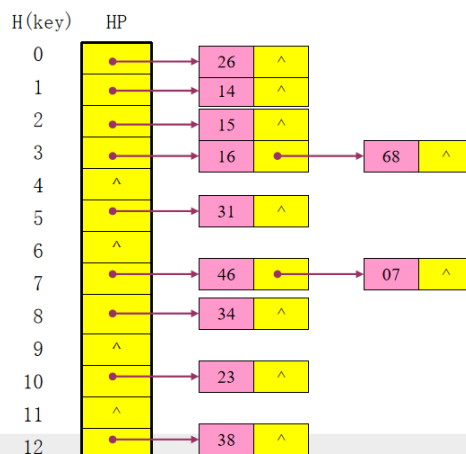
$HP[13]$ 。 对例9中:

$k=\{23, 34, 14, 38, 46, 16, 68, 15, 07, 31, 26\}$

依次取其中各值, 用链地址法

解决冲突时的Hash表如图:

链地址法解决冲突的优点: 无堆积现象; 删除表中记录容易实现。而开放地址法的Hash表作删除时, 不能将记录所在单元置空, 只能作删除标记。



华清远见

华清远见教育
www.hqvi.com

@作业

```
char a[] = "asdfasdlifadshjklrgeopaewrfpawedfoplhaqerqwrweqrterfgd";
```

统计字符串中出现字母最多的字母

```
int hash_list[26] = { 0 };
```

```
hash_list[0] = 5    //'a' 5次
```

```
hash_list[1] = 6    //'b' 6次
```

```
int post = key - 'a';
```

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

```
int main(int argc, const char *argv[])
```

```
{
```

```
    int i;
```

```
    int max;
```

```
    char a[] = "asddfsaffddkdasdffjadsfadsfdfsdfgladsfladsfkkfgdf";
```

```
    int b[26] = { 0 }; //26个元素,元素的值用来记录每个字母出现的次数
```

```
    //每个字符出现的次数,与在哈希表存储位置的下标关系
```

```
    //b[0] ---- b[25]
```

```
    // a ---- z
```

```
    //a ----> b[0]
```

```
    //b ----> b[1]
```

```
    //...
```

```
    //...
```

```
    //z ----> b[25]
```

```
    //存储位置 = 字符串中的字符 - 'a'
```

```
    //遍历字符串
```

```
    for(i = 0; a[i] != '\0'; i++)
```

```
    {
```

```
        int post = a[i] - 'a'; //得到a[i]这个字母出现的次数,在b数组中的存储位置下标
```

```
        b[post]++; //b[0] = b[0] + 1
```

```
    }
```

```
    #if 0
```

```
        for(i = 0; i < 26; i++)
```

```
        {
```

```
            //i + 'a' 存储位置 + 'a' 得到的对应的字母
```

```
            printf("%c出现的次数:%d\n", i + 'a', b[i]);
```

```
        }
```

```
    #endif
```

```
    max = b[0];
```

```
    for(i = 1; i < 26; i++)
```

```
    {
```

```
        max = b[i] > max ? b[i] : max;
```

```
    }
```

```
    //有可能出现并列第一的情况
```

```
    for(i = 0; i < 26; i++)
```

```
    {
```

```
        if(b[i] == max)
```

```
        {
```

```
            printf("%c出现次数最多%d次!!\n", i + 'a', max);
```

```
        }
```

```
}  
    return 0;  
}
```

```
linux@ubuntu:~$ gcc test.c  
linux@ubuntu:~$ ./a.out  
d出现次数最多13次!!  
f出现次数最多13次!!  
linux@ubuntu:~$ gedit test.c
```