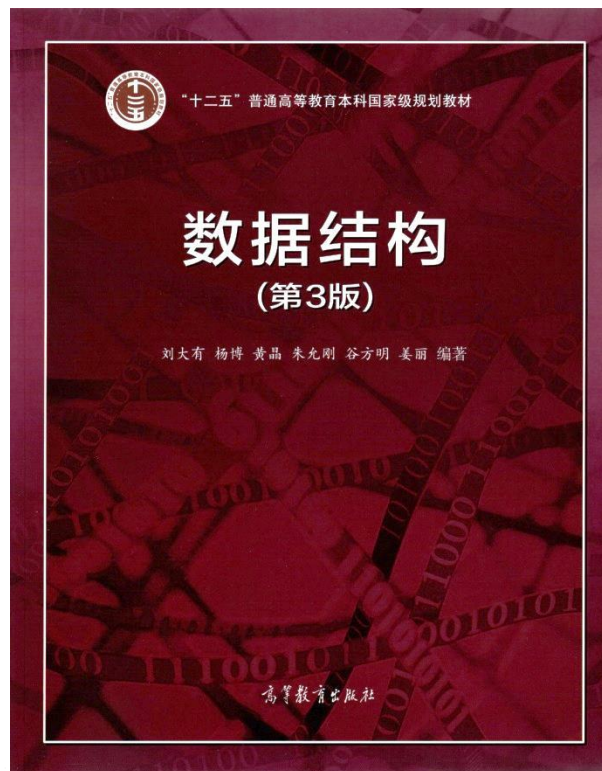




think.create.solve



线性表查找

顺序查找
对半查找
斐波那契查找
插值查找
分块查找
再论对半查找



数据之法
结构之美
算法之道



查找的基本概念

- **定义：**查找亦称**检索**。给定一个文件包含 n 个记录（或称元素、结点），每个记录都有一个关键词域。一个**查找算法**，就是对给定的值 **K** ，在文件中找关键词等于 K 的那个记录。
- **查找结果：**成功、失败。
- **平均查找长度：**查找一个元素所作的关键词平均比较次数，是衡量一个查找算法优劣的主要标准。



无序表的顺序查找

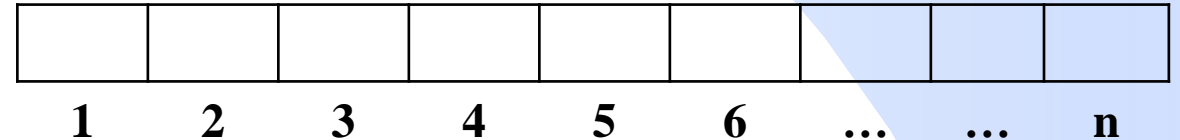
在线性表 R_1, R_2, \dots, R_n 中查找关键词等于 K 的元素：从线性表的起始元素开始，逐个检查每个元素 R_i ($1 \leq i \leq n$)，若查找成功，返回 K 在 R 中的下标，若查找失败，返回-1。

```
int Search(int R[], int n, int K){  
    for(int i=1; i<=n; i++)  
        if(R[i]==K) return i;  
    return -1;  
}
```

平均查找长度 (Average Search Length, ASL)

➤ 查找**成功**的平均查找长度:

$$\begin{aligned} ASL_{succ} &= \sum_{i=1 \dots n} P_i \times C_i \\ &= \frac{1}{n} \sum C_i \\ &= \frac{1}{n} \sum i = \frac{1}{n} \frac{n(n+1)}{2} \\ &= \frac{n+1}{2} \end{aligned}$$



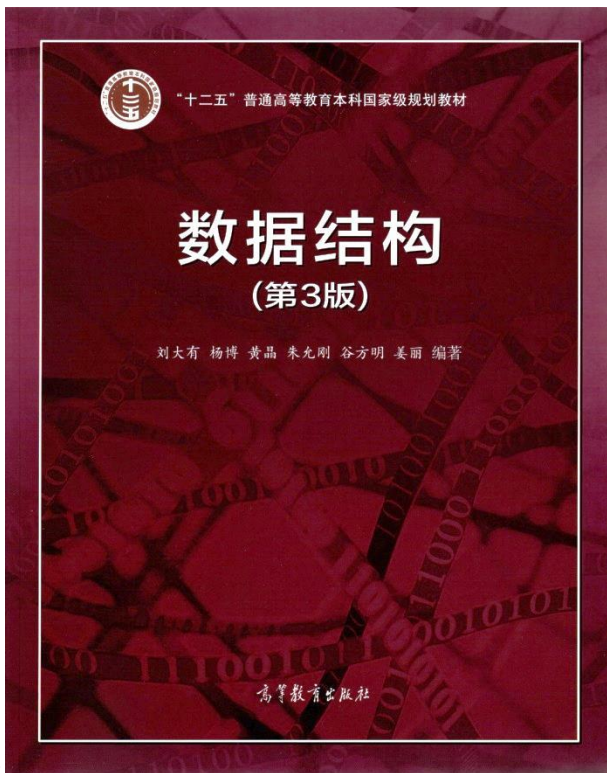
➤ 查找**失败**的查找长度: $ASL_{unsucc} = n+1$

➤ 顺序查找的时间复杂度: $O(n)$



think.create.solve

线性表查找



顺序查找

对半查找

斐波那契查找

插值查找

分块查找

再论对半查找

数据之法
结构之美
算法之道

有序表的二分查找

- R_1, R_2, \dots, R_n 按照关键词递增有序。
- 选取一个位置 i ($1 \leq i \leq n$), 比较 K 和 R_i , 若:
 - ✓ $K < R_i$, [K 只可能在 R_i 左侧, 不必考虑 R_i, \dots, R_n]
 - ✓ $K = R_i$, [查找成功结束]
 - ✓ $K > R_i$, [K 只可能在 R_i 右侧, 不必考虑 R_1, \dots, R_i]
- 使用不同的规则确定 i , 可得到不同的二分查找方法: 对半查找、斐波那契查找、插值查找等。

R_1, R_2, \dots, R_{i-1}	R_i	R_{i+1}, \dots, R_N
----------------------------	-------	-----------------------

对半（折半）查找

➤ K 与待查表的中间记录的关键词 $R_{\lfloor n/2 \rfloor}$ 比较，有三种可能结果：

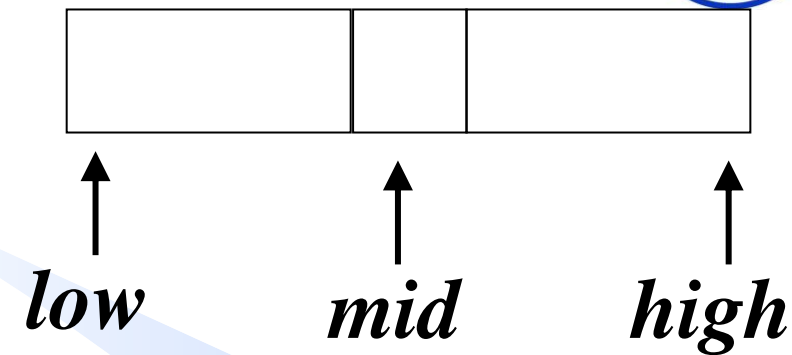
① $K < R_{\lfloor n/2 \rfloor}$ ， ② $K = R_{\lfloor n/2 \rfloor}$ ， ③ $K > R_{\lfloor n/2 \rfloor}$

由情况 ① 和 ③ 能确定下一次应到表的哪一半中去查找，即将查找范围缩小一半，由情况 ② 知查找成功结束；

➤ 对确定的那一半重复上述过程，直至找到所查记录或确定该记录不在表中。

对半查找

```
int BinarySearch(int R[],int n, int K){  
    //在数组R中对半查找K, R中关键词递增有序  
    int low = 1, high = n, mid;  
    while(low <= high){  
        mid=(low+high)/2;  
        if(K<R[mid]) high=mid-1;  
        else if(K>R[mid]) low=mid+1;  
        else return mid;  
    }  
    return -1;    //查找失败  
}
```

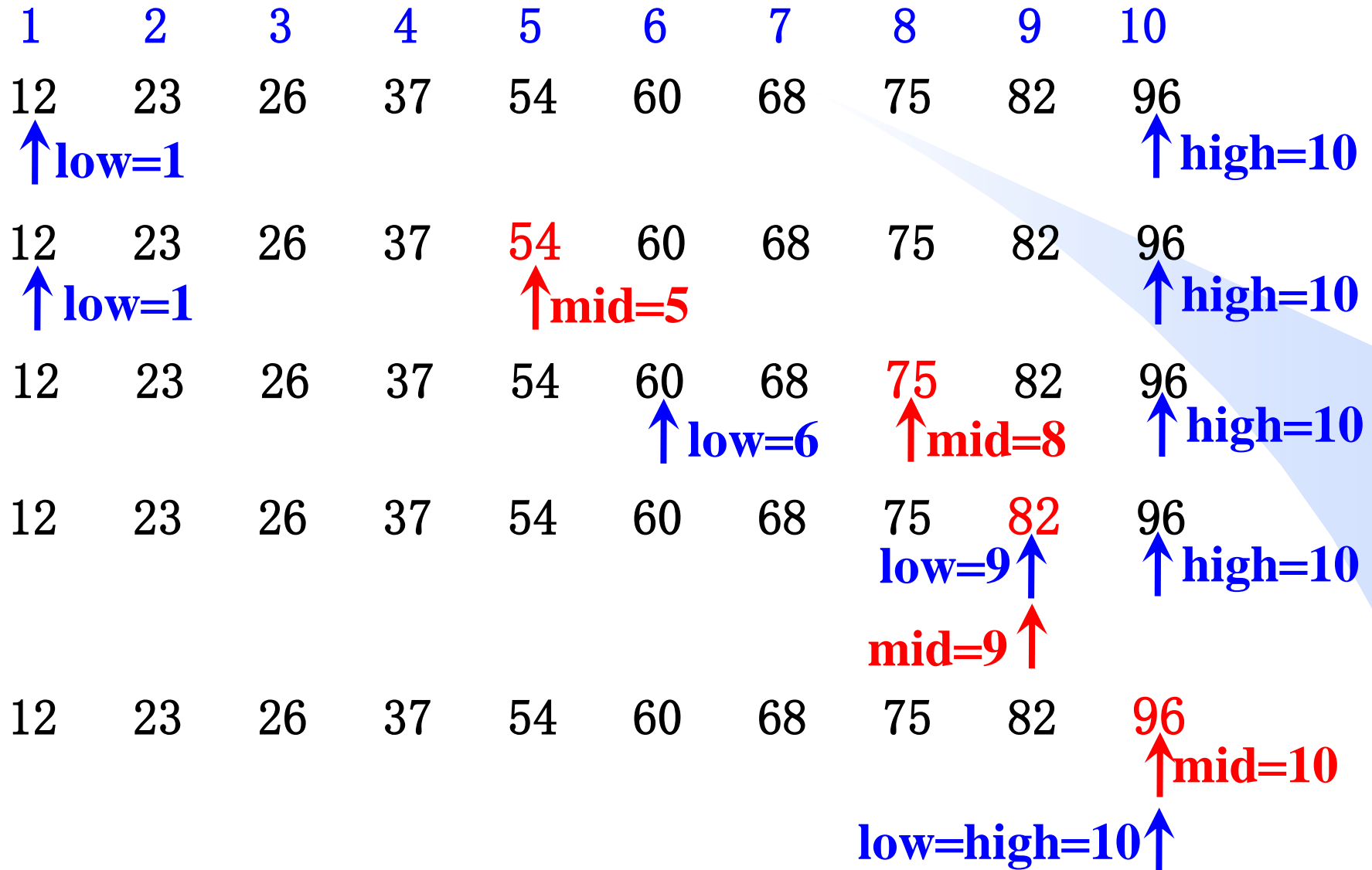


//在左半部分查找
//在右半部分查找
//查找成功

时间复杂度
 $O(\log n)$

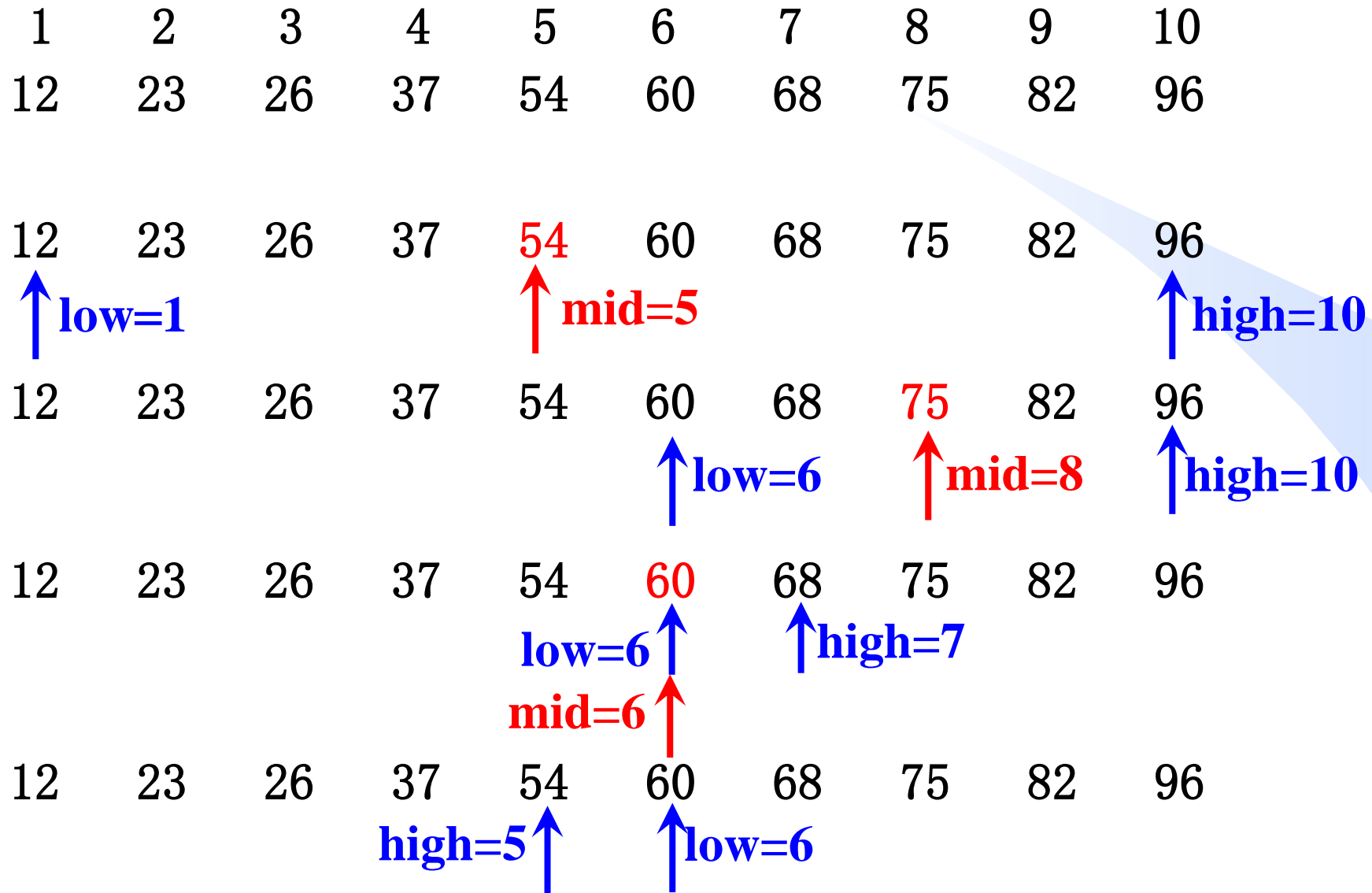


例：查找 $K=96$ 时二分查找过程（4次比较成功）





查找 $K=58$ 时二分查找过程 (3次比较, 查找失败)



二叉判定树

为便于分析算法的时间复杂度，采用二叉树表示查找过程

对于有序表 $R_{low}, R_{low+1}, \dots, R_{high}$ ，对半查找的二叉判定树 $T(low, high)$ 的 **递归定义** 如下：

- 当 $high - low + 1 \leq 0$ 时： $T(low, high)$ 为空；
- 当 $high - low + 1 > 0$ 时：
 - ✓ $T(low, high)$ 的根结点是 $\lfloor (low + high) / 2 \rfloor$ ；
 - ✓ 根结点的左子树是 $R_{low}, \dots, R_{\lfloor (low + high) / 2 \rfloor - 1}$ 对应的二叉判定树；
 - ✓ 根结点的右子树是 $R_{\lfloor (low + high) / 2 \rfloor + 1}, \dots, R_{high}$ 对应的二叉判定树。

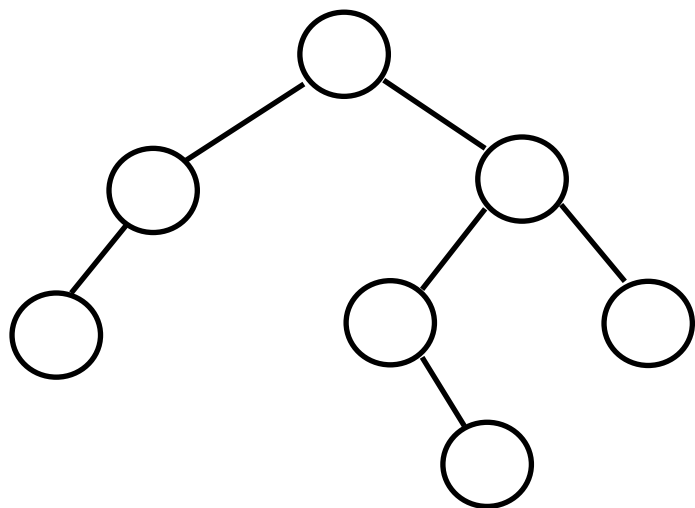
$low, \dots, \lfloor (low + high) / 2 \rfloor - 1$	$\lfloor (low + high) / 2 \rfloor$	$\lfloor (low + high) / 2 \rfloor + 1, \dots, high$
--	------------------------------------	---

回顾：扩充二叉树

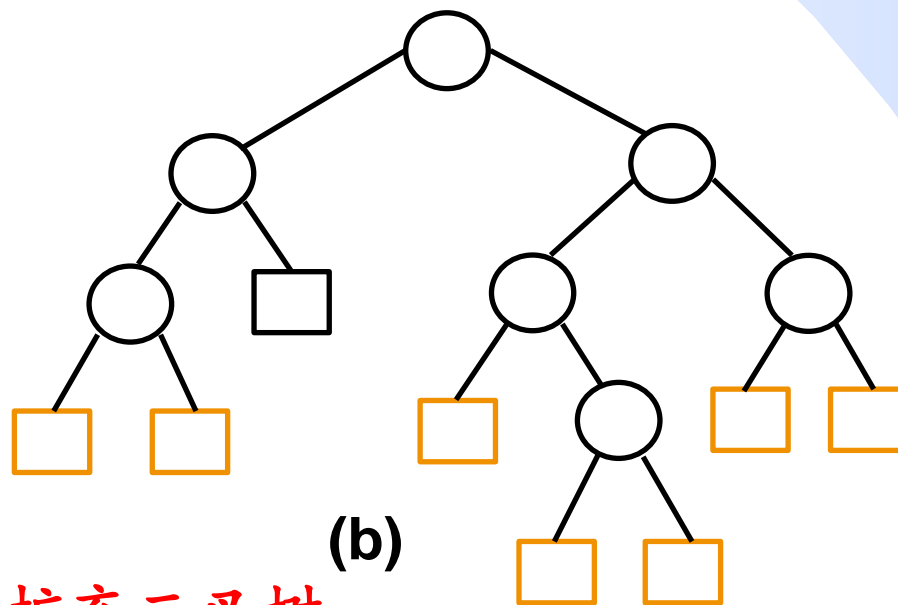
当二叉树中出现一个空的子树时，就增加特殊的结点，由此生成的二叉树称为**扩充二叉树**。称圆形结点为内结点，方形结点为外结点。

图(a)所示的二叉树的扩充二叉树如图(b)所示。

二叉判定树是一棵**扩充二叉树**。



(a)

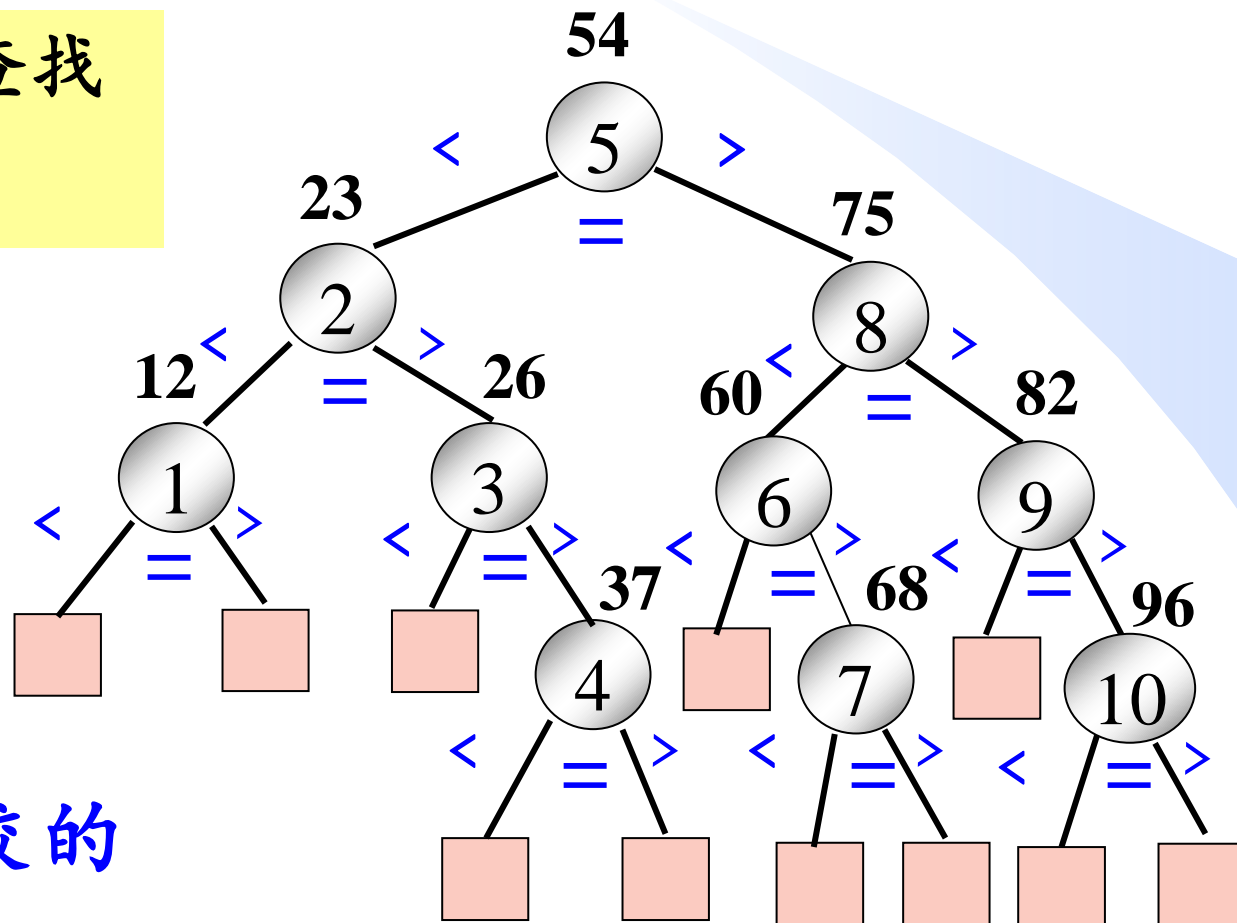


(b)

二叉树和它的扩充二叉树

每个圆圈结点表示
与一个关键词的比较

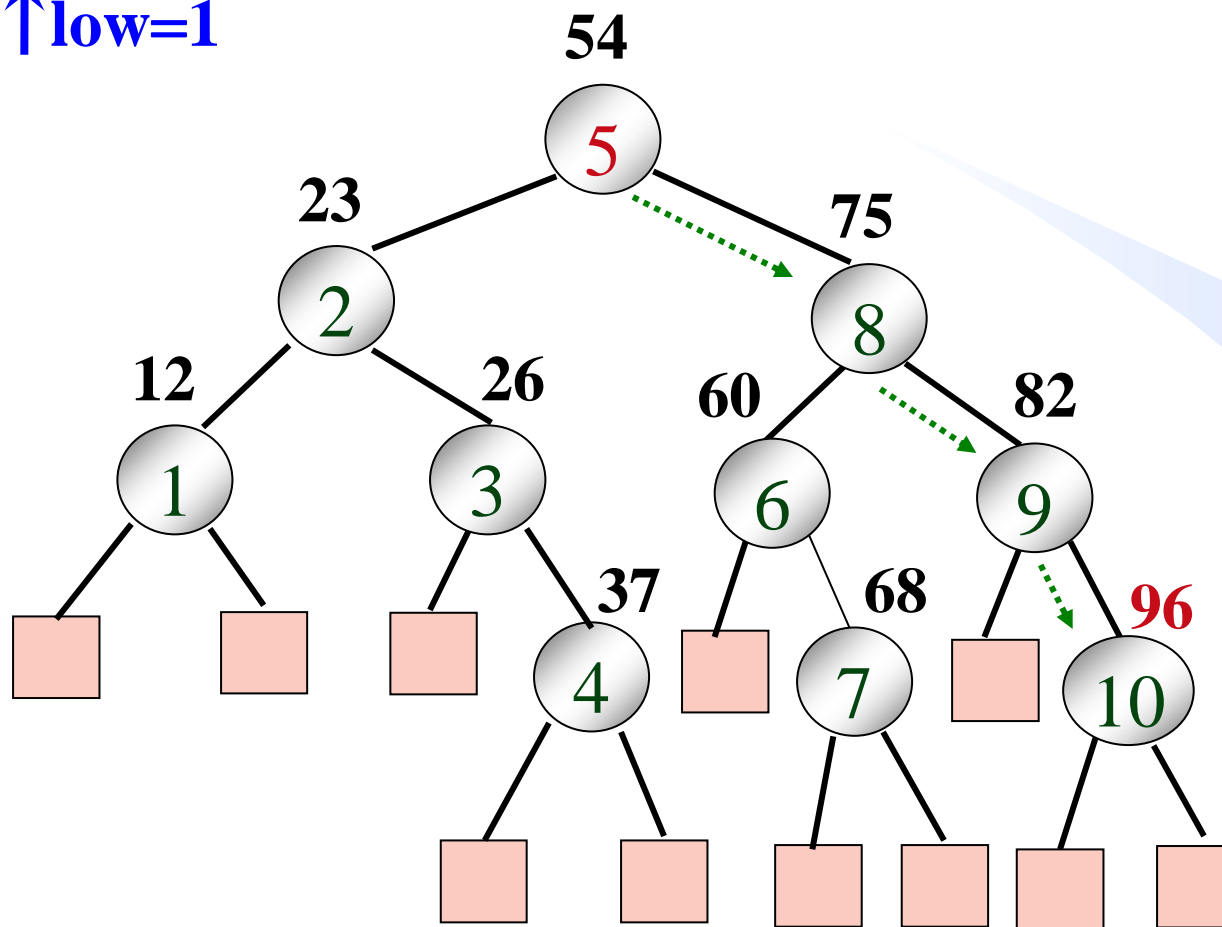
结点值：关键词比较的位置（下标）



查找**K=96**
成功情况

1	2	3	4	5	6	7	8	9	10
12	23	26	37	54	60	68	75	82	96
									↑
									high=10

↑ low=1



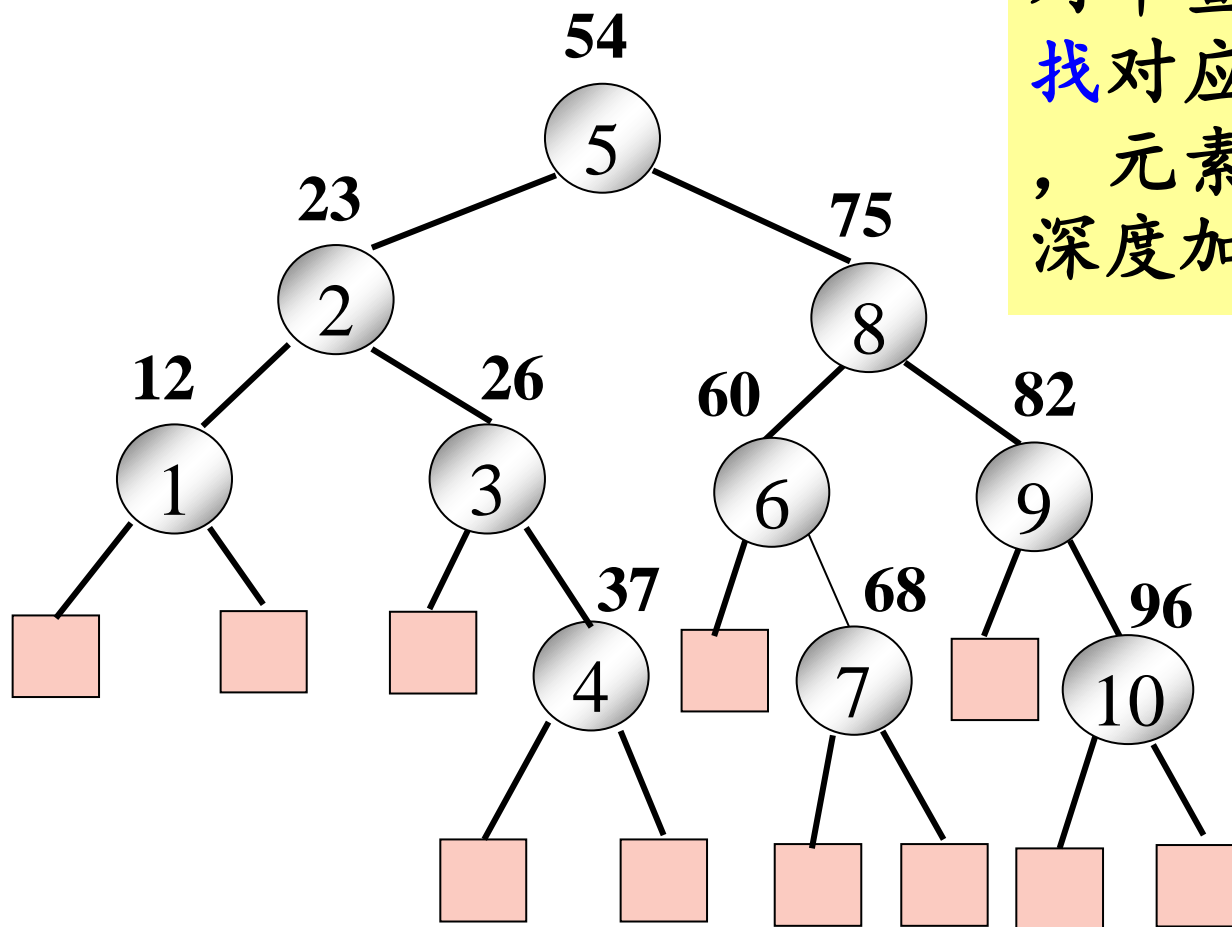
对半查找算法的**每次成功查找**正好对应判定树的一个**内结点**，元素比较次数为**该结点的深度加1**，即从根到该结点所经过的结点数。



每次不成功的查找对应判定树的一个外结点，元素比较次数恰好为该结点深度，即根到该节点所经过的内结点数。

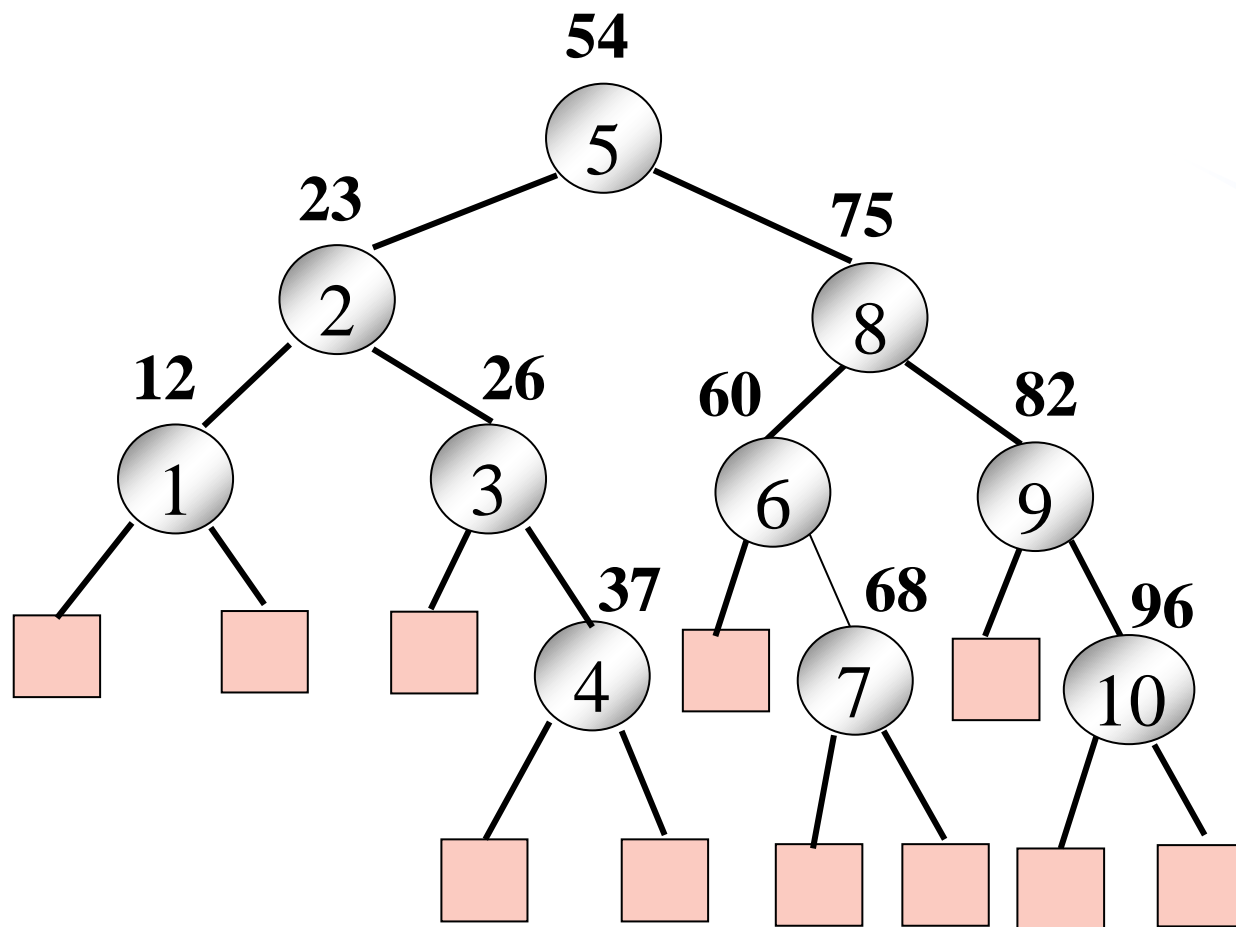
查找成功的平均查找长度

对半查找算法的每次成功查找对应判定树的一个内结点，元素的比较次数为该结点深度加1。



$$ASL_{SUCC} = (1*1 + 2*2 + 4*3 + 3*4) / 10 = 29 / 10 = 2.9$$

查找失败的平均查找长度



每次不成功的查找对应判定树的一个外结点，关键词的比较次数恰好为该结点深度。

$$ASL_{UNSUCC} = (5*3+6*4)/11 = 39/11$$

$$K_1 \ K_2 \ K_3 \ \dots \ K_{n-1} \ K_n$$

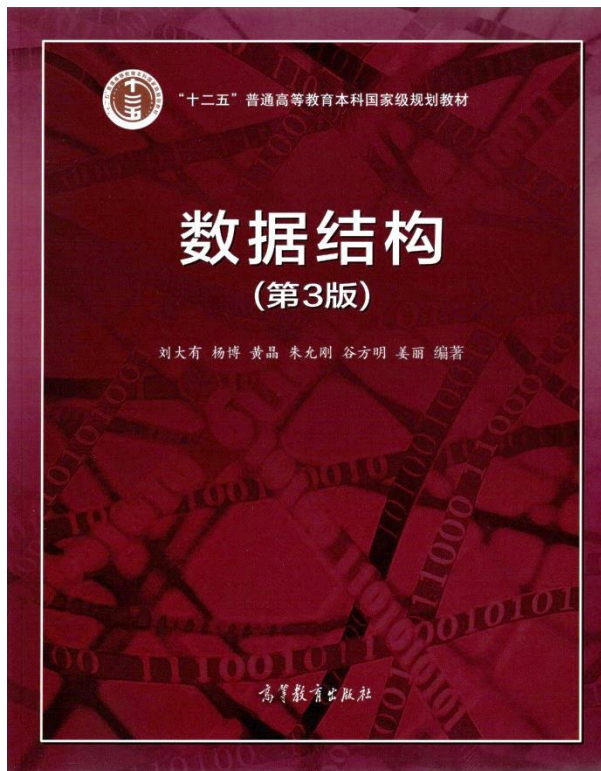


对半查找总结

- 优点：查找效率为 $O(\log n)$ ，比顺序查找高。
- 缺点：只适用于有序表，且限于顺序存储结构，对线性链表难以进行对半查找。



think.create.solve



线性表查找

查找的基本概念

顺序查找

对半查找

斐波那契查找

插值查找

分块查找

再论对半查找



数据之法
结构之美
算法之道



Fibonacci查找

➤ Fibonacci 序列：0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

$$F_0=0, F_1=1,$$

$$F_k=F_{k-1}+F_{k-2}, j \geq 2$$

➤ 斐波那契（Fibonacci）查找：折半查找的改进，以 Fibonacci 序列的分划代替对半查找的均匀分划。

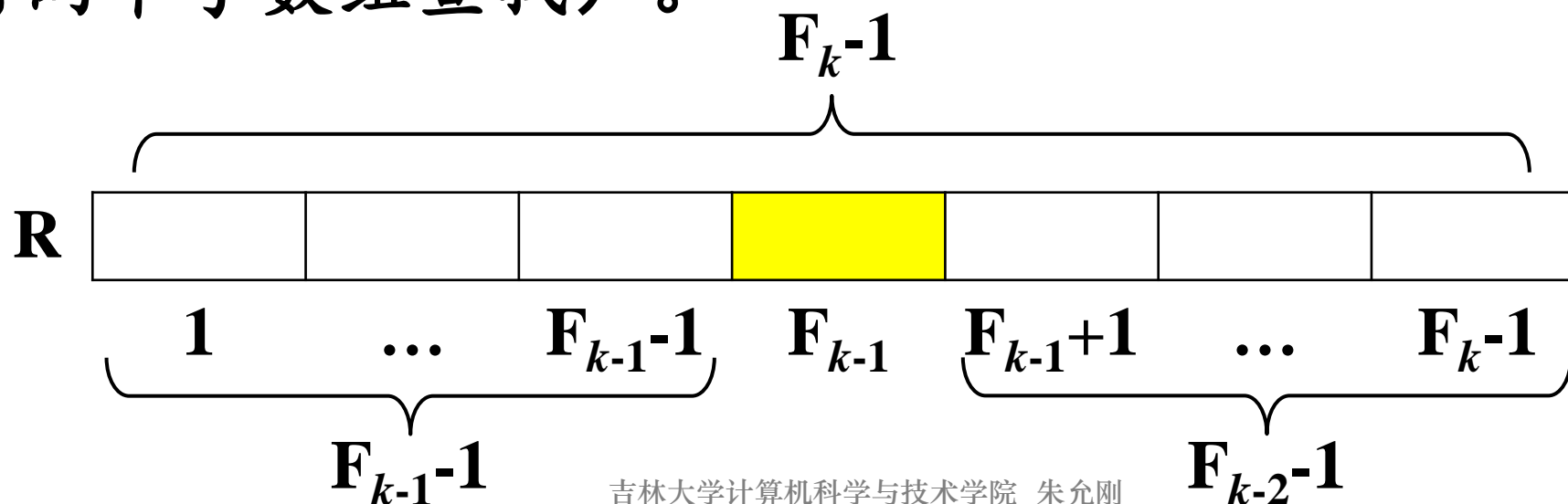
➤ 设有一个长度为 $F_k - 1$ 的有序数组 R 。下标 F_{k-1} 将数组分为三部分：

✓ 左子数组 $R[1] \dots R[F_{k-1} - 1]$ ； 长度为 $F_{k-1} - 1$

✓ F_{k-1} ；

✓ 右子数组 $R[F_{k-1} + 1] \dots R[F_k - 1]$ ； 长度为 $F_k - 1 - F_{k-1} = F_{k-2} - 1$

➤ 原数组长度和左右两个子数组长度均为某个 Fibonacci 数 - 1，即把大问题（对大数组查找）分解为两个结构相同的子问题（对两个子数组查找）。



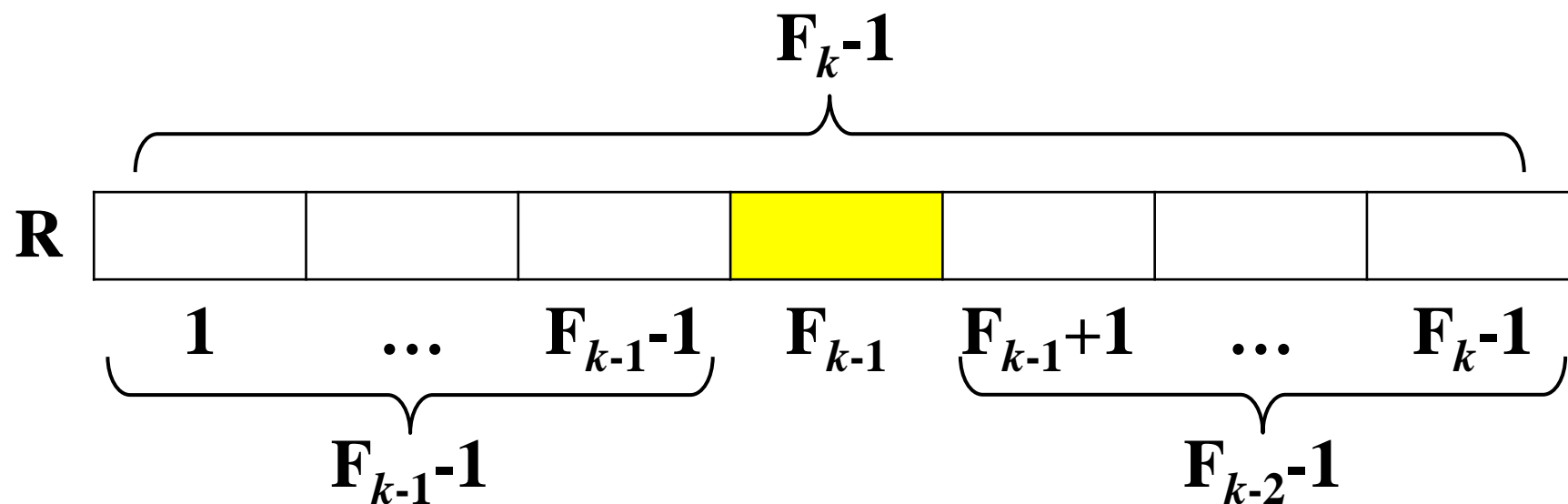
Fibonacci查找

假定数组中元素个数 n 是某个Fibonacci数减1，即 $n=F_k-1$ 。把 K 与 $R[F_{k-1}]$ 比较，若：

- $K == R[F_{k-1}]$ ：则查找成功；
- $K < R[F_{k-1}]$ ：在 $R[1]...R[F_{k-1}-1]$ 内继续查找；
- $K > R[F_{k-1}]$ ：在 $R[F_{k-1}+1]...R[F_k-1]$ 内继续查找。

$k-1$ 阶Fibonacci查找

$k-2$ 阶Fibonacci查找



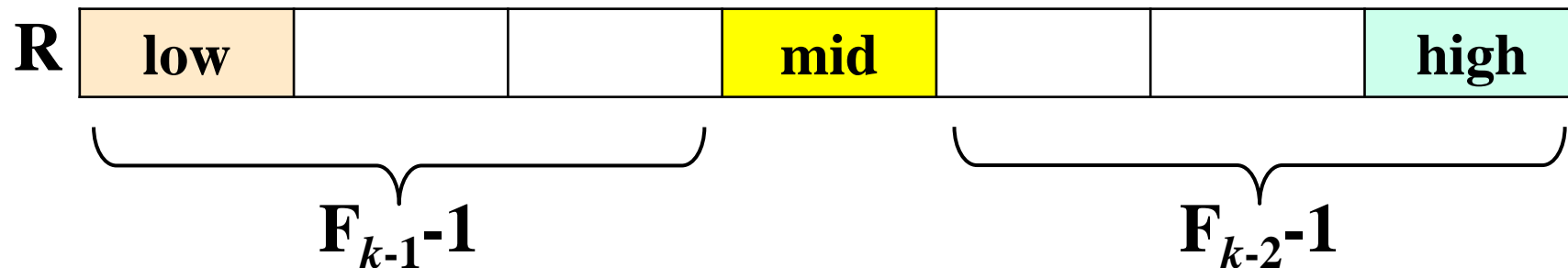
对长度为 F_k-1 的数组进行Fibonacci查找，不妨简称为 k 阶Fibonacci查找

```
int Fib_Search(int R[], int n, int K, int F[], int k){
//对有序文件 $R_1, \dots, R_n$ 进行Fibonacci查找, 假定Fibonacci数列存于数组F, 且
//数组长度 $n=F[k]-1$ , 若查找成功, 返回K在R中下标, 否则返回-1
```

对左侧子数组进行
 $k-1$ 阶Fibonacci查找

```
    int low=1, high=n;
    while(low <= high){
        int mid=low+F[k-1]-1;
        if(K<R[mid]) {high=mid-1; k--;}
        else if(K>R[mid]) {low=mid+1; k-=2;}
        else return mid;
    }
    return -1;
}
```

对右侧子数组进行
 $k-2$ 阶Fibonacci查找





```
int Fib_Search(int R[], int n, int K, int F[], int k){  
//对有序文件 $R_1, \dots, R_n$ 进行Fibonacci查找, 假定Fibonacci数列存于数组F
```

```
    int low=1, high=n;  
    while(low <= high){  
        int mid=low+F[k-1]-1;  
        if(K<R[mid]) {high=mid-1; k--;}  
        else if(K>R[mid]) {low=mid+1; k-=2;}  
        else return (mid<n)? mid:n;  
    }  
    return -1;  
}
```

对左侧子数组进行
 $k-1$ 阶Fibonacci查找

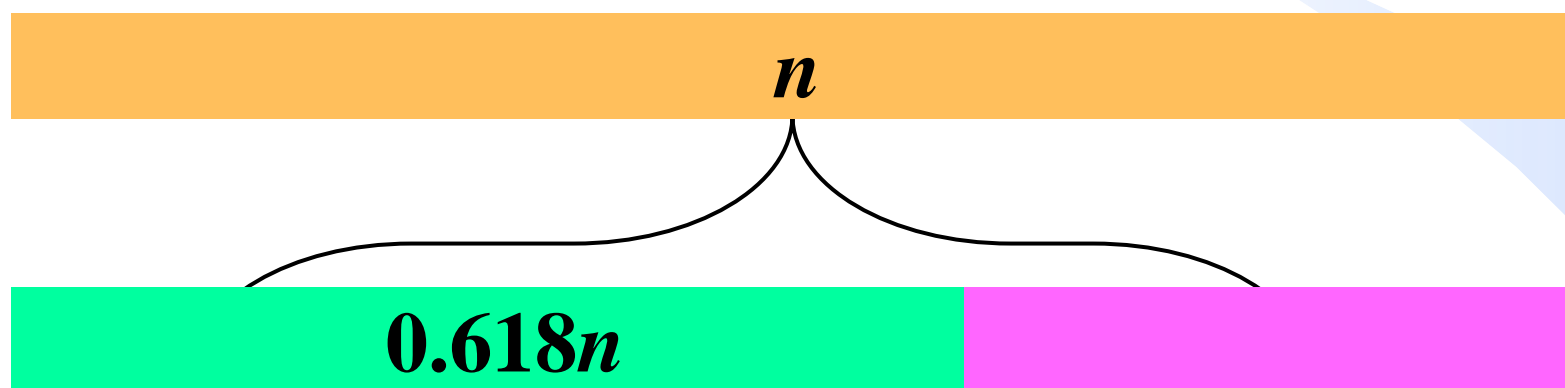
对右侧子数组进行
 $k-2$ 阶Fibonacci查找

若数组长度 n 不等于
Fibonacci 数减 1, 则令
 $k = \min_k \{F_k - 1 \geq n\}$, 把
数组长度扩充至 $F_k - 1$, 将
 $R[n+1]$ 至 $R[F_k - 1]$ 用 $R[n]$ 补全

R	15	16	23	56	98	98	98
	1	n	...	$F_k - 1$

Fibonacci查找

- 本质：在黄金分割点处对数组划分。



$$\lim_{k \rightarrow \infty} \frac{F_{k-1}}{F_k} = 0.618 \dots$$

F_k 为第 k 个斐波那契数



Fibonacci查找

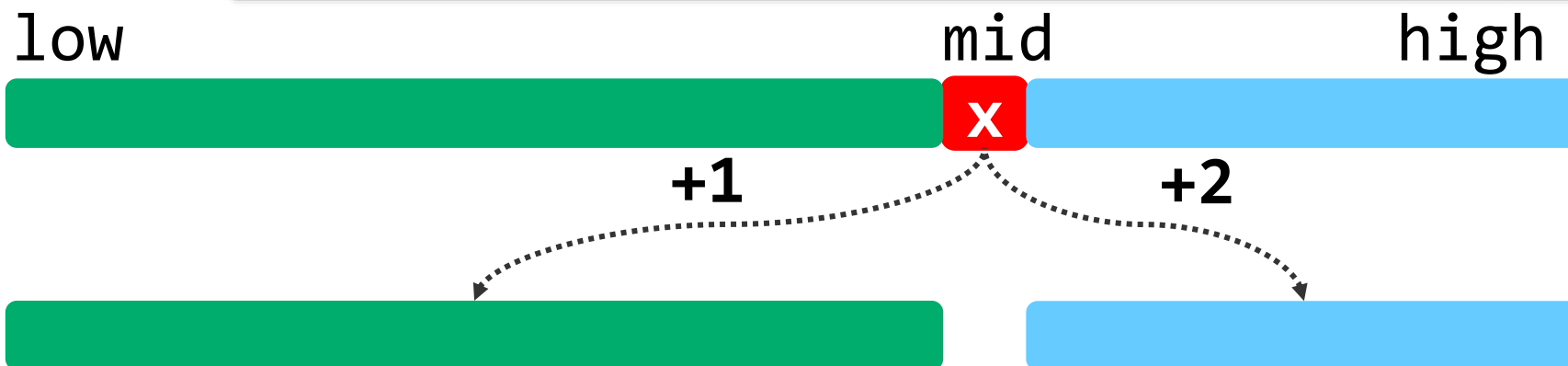
- 平均和最坏情况下的时间复杂度为 $O(\log_2 n)$ 。
- 总体运行时间略快于对半查找算法。
- 算法不涉及乘除法，而只涉及加减法。

```
int Fib_Search(int R[], int n, int K, int F[], int k){  
    int low=1,high=n;  
    while(low <= high){  
        int mid=low+F[k-1]-1;  
        if(K<R[mid]) {high=mid-1; k--;}  
        else if(K>R[mid]) {low=mid+1; k-=2;}  
        else return mid;  
    }  
    return -1;  
}
```

Fibonacci查找

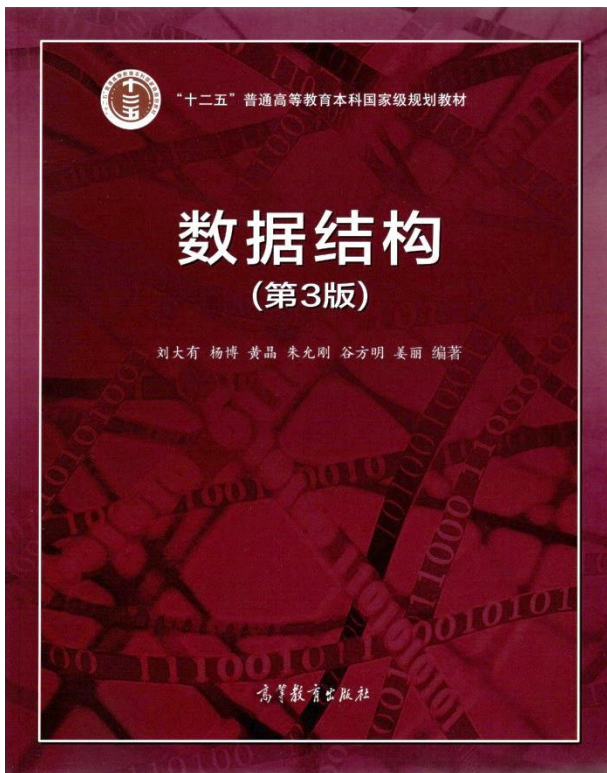
- 左区间较右区间长，使查找过程中进入左区间概率更大。而转向左区间前所做的关键词比较次数更少，从而使查找过程中关键词比较的总次数更少。

```
while(low <= high){  
    int mid=low+F[k-1]-1;  
    if(K<R[mid]) {high=mid-1; k--;}  
    else if(K>R[mid]) {low=mid+1; k-=2;}  
    else return mid;  
}
```





think.create.solve



数据之法
结构之美
算法之道

线性表查找

查找的基本概念

顺序查找

对半查找

斐波那契查找

插值查找

分块查找

再论对半查找



zhuyungang@jlu.edu.cn

插值查找

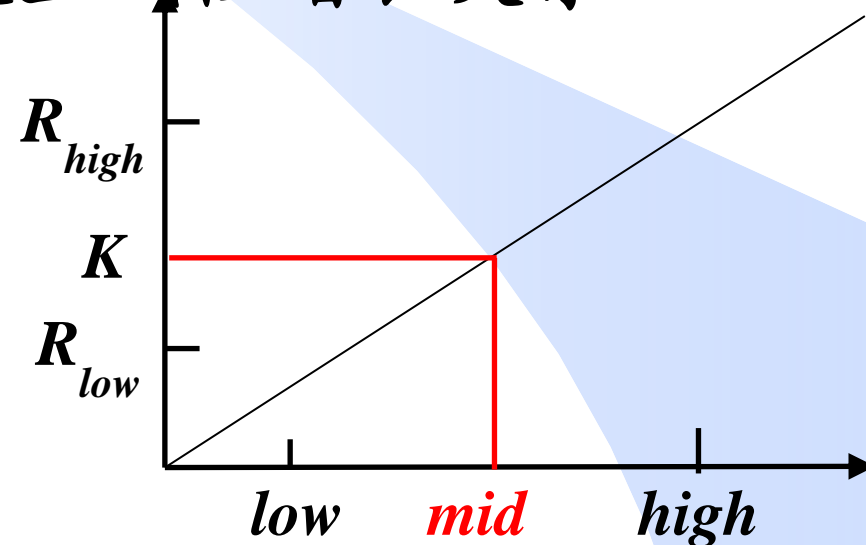
➤ 假设：有序数组 R 中元素 **均匀** 随机分布，例如

10	20	30	40	50	60	70	80
1	2	3	4	5	6	7	8

➤ 于是， $R[low]...R[high]$ 内各元素应大致呈线性增长关系

$$\frac{mid - low}{high - low} \approx \frac{K - R_{low}}{R_{high} - R_{low}}$$

$$mid \approx low + \frac{K - R_{low}}{R_{high} - R_{low}} (high - low)$$

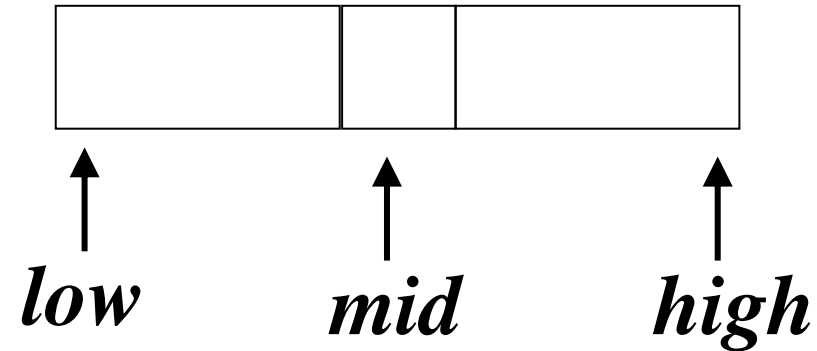


➤ 每次迭代过程中，通过线性插值预测 K 的期望位置 mid 。

➤ 回顾对半查找： $mid = \frac{low + high}{2} = low + \frac{1}{2} (high - low)$

插值查找

```
int Interpolation_Search(int R[], int n, int K){  
    int low=1, high=n, mid;  
    while(low<=high && K>=R[low] && K<=R[high]){  
        if(low==high) return low;  
        mid=ceil(low+(K-R[low])*(high-low)/(R[high]-R[low]));  
        if(K<R[mid]) high = mid-1;  
        else if(K>R[mid]) low = mid+1;  
        else return mid;  
    }  
    return -1;  
}
```



在英文词典查找单词
“zoo”，你为什么不用
对半查找法，而直接从
字典的后面找？



插值查找时间复杂度



姚期智

图灵奖获得者

中国科学院院士

美国科学院外籍院士

哈佛大学博士

加州大学伯克利分校教授 (1981-1982)

斯坦福大学教授 (1982-1986)

普林斯顿大学教授 (1986-2004)

清华大学教授 (2004-现在)

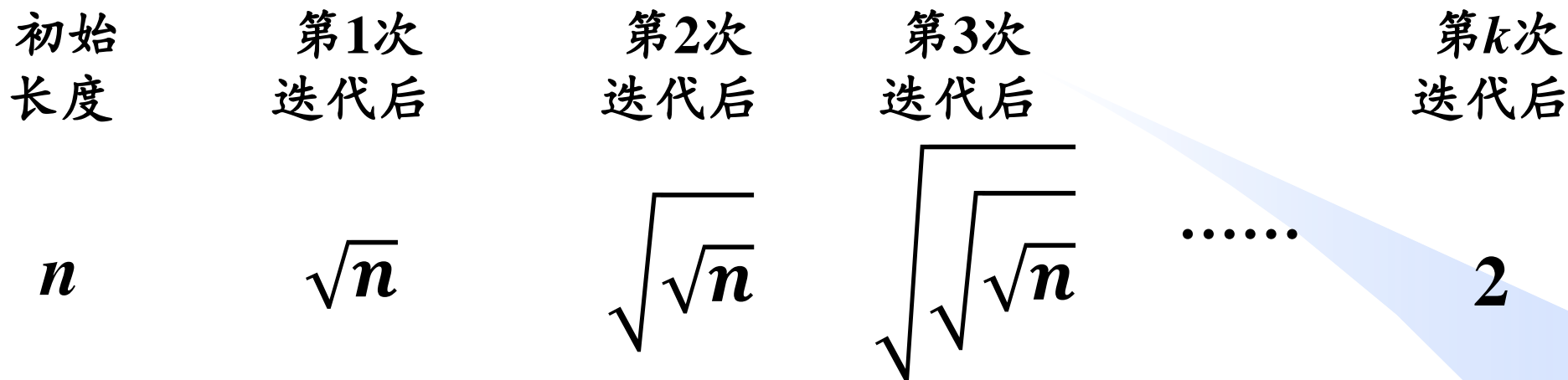
储枫

麻省理工学院博士

清华大学教授

姚期智及夫人储枫证明：插值查找算法每经一次迭代，平均情况下待查找区间的长度由 n 缩至 \sqrt{n} 。

插值查找时间复杂度



➤ 元素比较的次数 = 迭代的次数。

➤ 假定 k 次迭代，则 $n^{(\frac{1}{2})^k} = 2$

➤ 平均时间复杂度 $O(\log \log n)$ 。

➤ 最坏情况：元素分布极不均匀；
最坏时间复杂度 $O(n)$ 。

1	2	3	5	6	9999
1	2	3	4	5	6

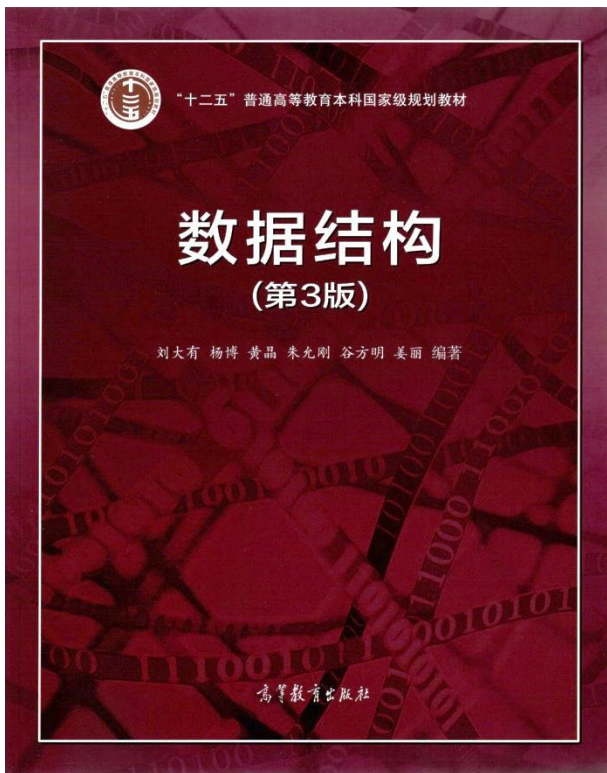


插值查找总结

- 从 $O(\log n)$ 到 $O(\log \log n)$ 优势并不明显（除非查找表极长，或比较操作成本极高）。
比如 $n=2^{32} \approx 42.9$ 亿
 $\log n = \log 2^{32} = 32$
 $\log \log n = \log 32 = 5$
- 需引入乘除法运算。
- 元素分布不均匀时效率受影响。
- 实际中可行的方法：首先通过插值查找迅速将查找范围缩小到一定的范围，然后再进行二分查找或顺序查找。



think.create.solve



线性表查找

查找的基本概念

顺序查找

对半查找

斐波那契查找

插值查找

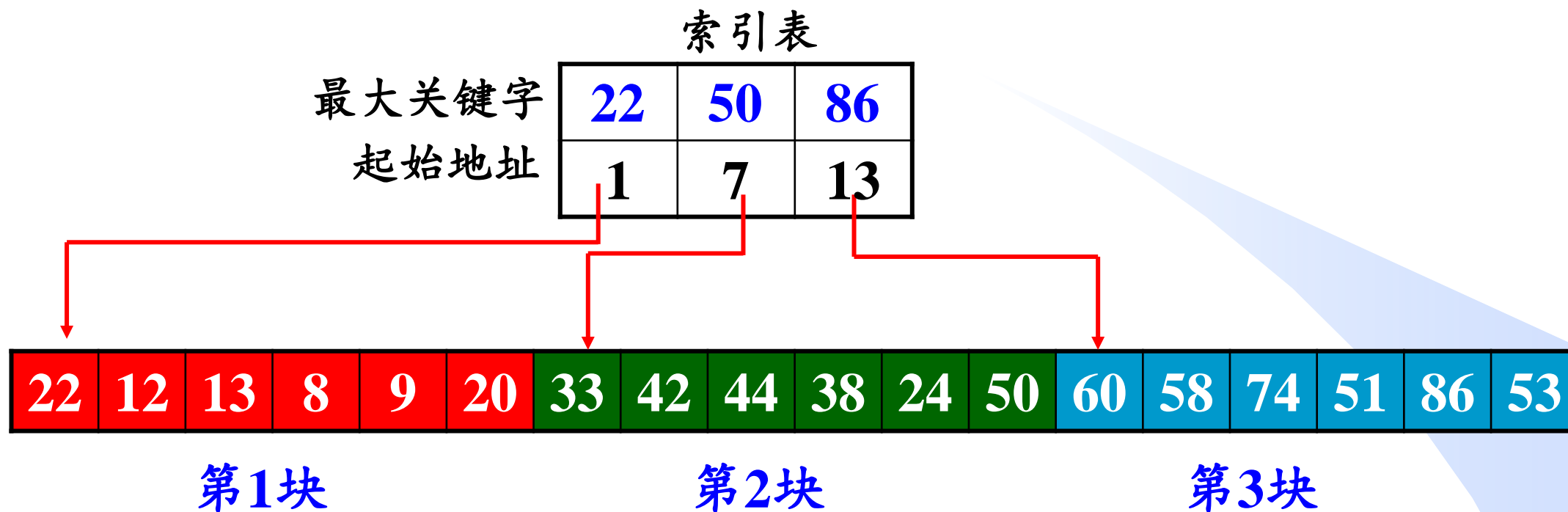
分块查找

再论对半查找



数据之法
结构之美
算法之道

分块查找



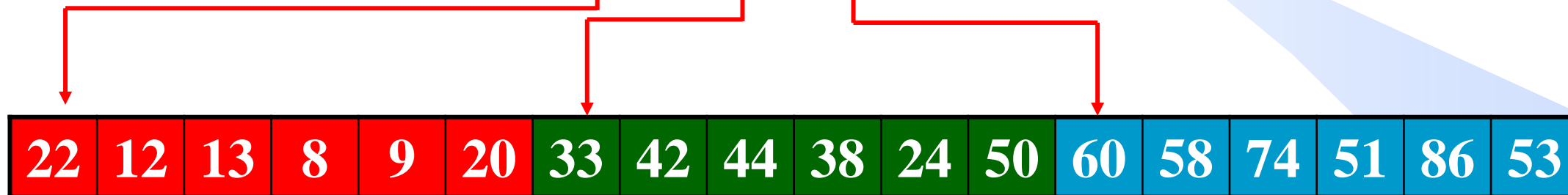
将大数组分成若干子数组（块），每个块中的数值都比后一块中数值小（块内不要求有序），建一个索引表记录每个子表的起始地址和各块中的最大关键字

分块查找

索引表

最大关键字
起始地址

22	50	86
1	7	13



第1块

第2块

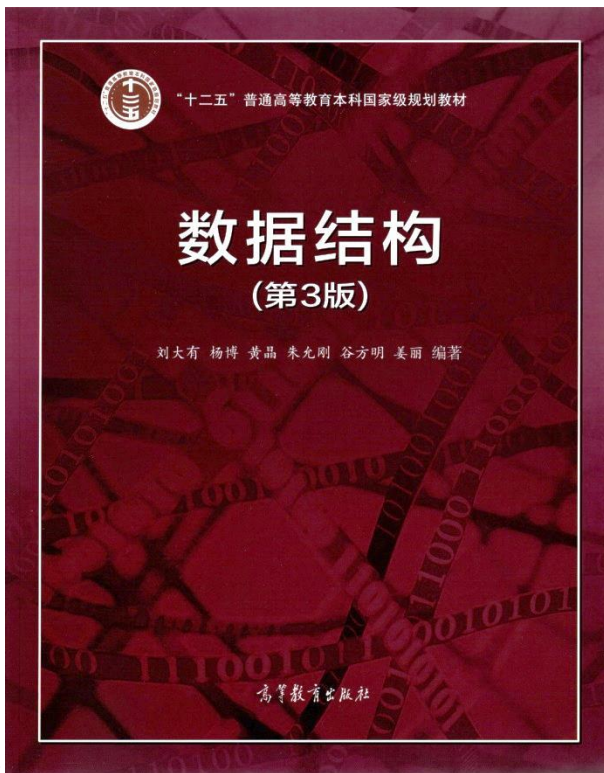
第3块

查找过程

- ① 对索引表使用对半查找（因为索引表是有序表）
- ② 确定了关键字所在的块后，在块内采用顺序查找



think.create.solve



线性表查找

查找的基本概念

顺序查找

对半查找

斐波那契查找

插值查找

分块查找

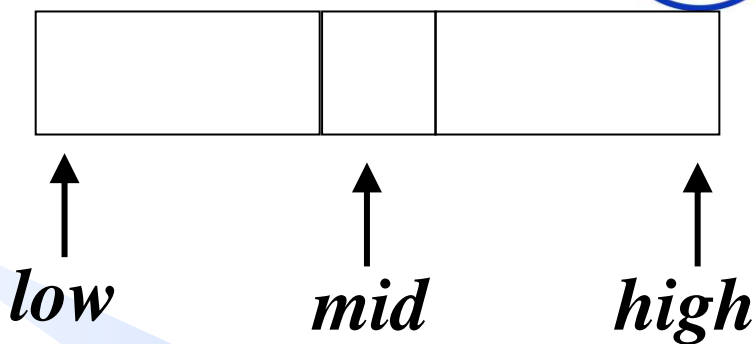
再论对半查找



数据之法
结构之美
算法之道

回顾——传统对半查找

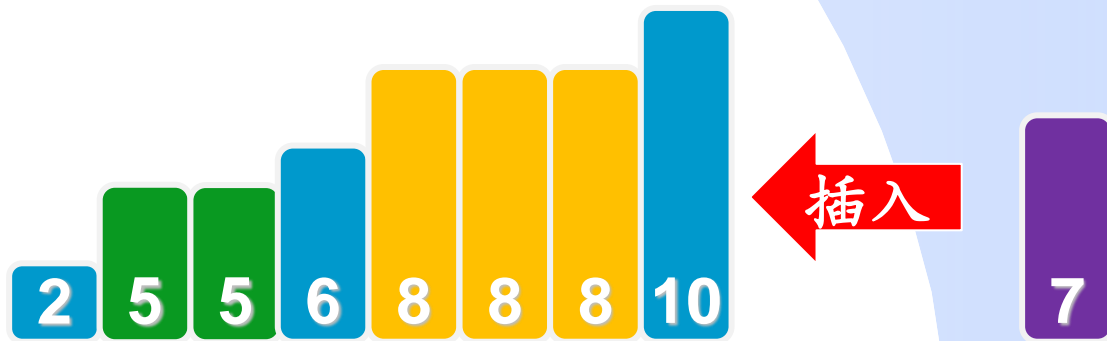
```
int BinarySearch(int R[], int n, int K){
    //在数组R中对半查找K, R中关键词递增有序
    int low = 1, high = n, mid;
    while(low <= high){
        mid=(low+high)/2;
        if(K<R[mid]) high=mid-1;
        else if(K>R[mid]) low=mid+1;
        else return mid;
    }
    return -1; //查找失败
}
```



//在左半部分查找
 //在右半部分查找
 //查找成功

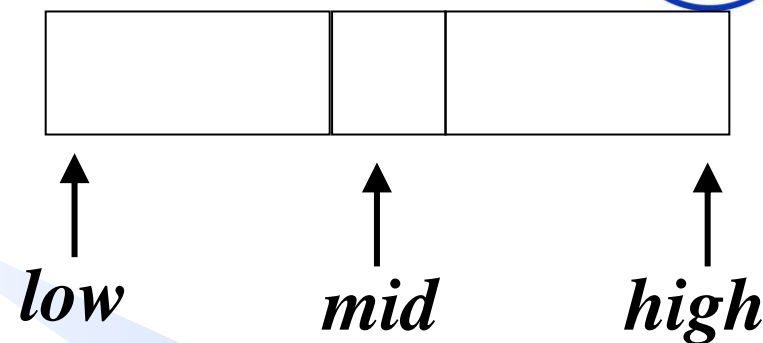
更好的方案：返回更多信息

- 若查找失败，能给出查找失败的位置，便于新元素插入



回顾——传统对半查找

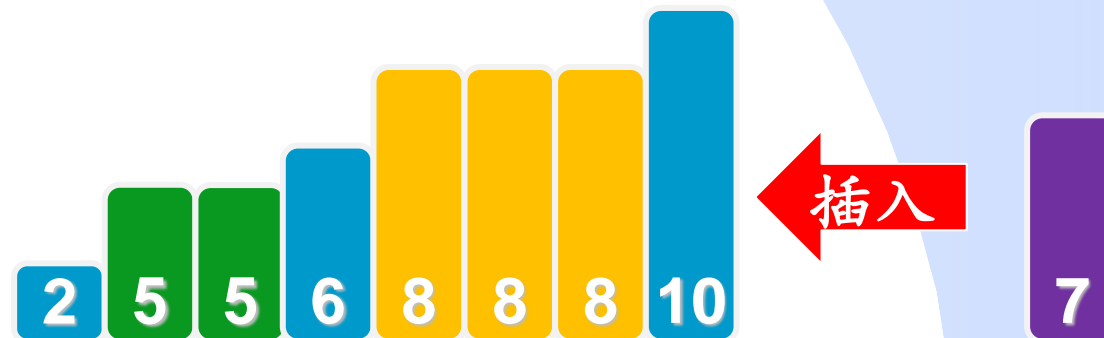
```
int BinarySearch(int R[], int n, int K){
    //在数组R中对半查找K, R中关键词递增有序
    int low = 1, high = n, mid;
    while(low <= high){
        mid=(low+high)/2;
        if(K<R[mid]) high=mid-1;
        else if(K>R[mid]) low=mid+1;
        else return mid;
    }
    return -1; //查找失败
}
```



//在左半部分查找
 //在右半部分查找
 //查找成功

返回:

- (1) 大于等于K的第一个位置
- (2) 小于等于K的最后一个位置

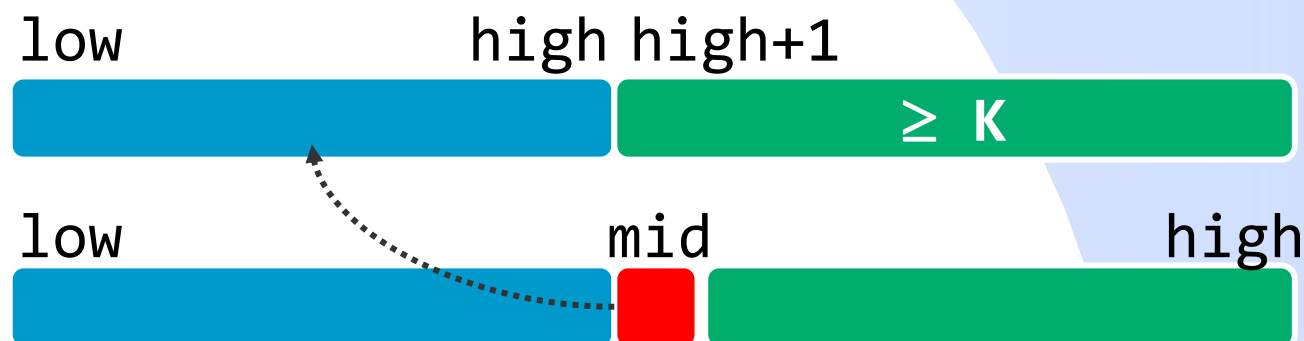


大于等于K的第一个位置

```
int BinarySearch2(int R[], int n, int K){
    int low = 1, high = n - 1;
    while(low <= high){
        int mid = (low + high)/2;
        if(K <= R[mid]) high = mid-1;
        else low = mid + 1;
    }
    return low;
}
```

算法终止时
low=high+1为 $\geq K$ 的第一个位置

high+1是目前确定的 $\geq K$ 的第一个位置



high low

< K

$\geq K$

小于等于K的最后一个位置

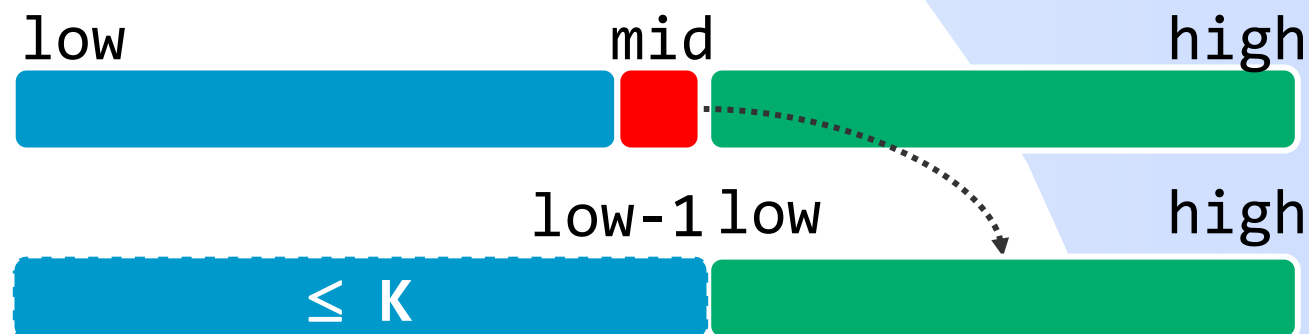
```
int BinarySearch3(int R[], int n, int K){
    int low = 1, high = n - 1;
    while(low <= high){
        int mid = (low + high)/2;
        if(K < R[mid]) high = mid-1;
        else low = mid + 1;
    }
    return high;
}
```

算法终止时
 $high = low - 1$ 为 $\leq K$ 的最后一个位置

high low

$\leq K$

$> K$



$low - 1$ 是目前确定的 $\leq K$ 的最后一个位置

例题：第一个/最后一个等于K的位置

给定有序整型数组 R 和一个整数 K ，找出 K 在数组中开始位置和结束位置，若 K 不在 R 中则返回-1，数组下标从0开始。【华为、字节跳动、百度、阿里、美团、小米、360、谷歌、微软、苹果面试题】

2	3	6	6	6	6	6	7	8
0	1	2	3	4	5	6	7	8

第一个等于K的位置

策略：先找 $\geq K$ 的第一个位置，再看该位置的元素是否等于K

```
int LeftBound(int R[], int n, int K){  
    int low = 0, high = n - 1;  
    while(low <= high){ //先找  $\geq K$  的第一个位置  
        int mid = (low + high)/2;  
        if(K <= R[mid]) high = mid-1;  
        else low = mid + 1;  
    } //此时low为  $\geq K$  的第一个位置  
    if(low < n && R[low] == K) return low;  
    return -1;  
}
```

时间复杂度
 $O(\log n)$

2	3	6	6	6	8	9
---	---	---	---	---	---	---

最后一个等于K的位置（右边界）

策略：先找 $\leq K$ 的最后一个位置，再看该位置元素是否等于K

```
int RightBound(int R[], int n, int K){  
    int low = 0, high = n - 1;  
    while(low <= high){ //找 $\leq K$ 的最后一个位置  
        int mid = (low + high)/2;  
        if(K < R[mid]) high = mid-1;  
        else low = mid + 1;  
    } //此时high为 $\leq K$ 的最后一个位置  
    if(high >= 0 && R[high] == K) return high;  
    return -1;  
}
```

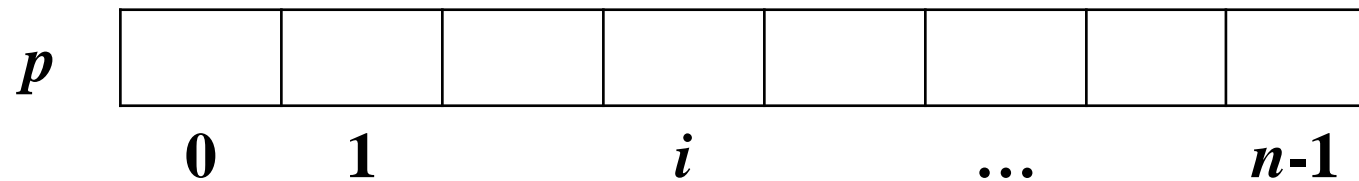
时间复杂度
 $O(\log n)$

2	3	6	6	6	8	9
---	---	---	---	---	---	---



二分搜索答案

珂珂喜欢吃香蕉。有 n 堆香蕉，第 i 堆中有 $p[i]$ 根香蕉。假定她吃香蕉的速度 s ，即每个小时她将会选择一堆香蕉，从中吃掉 s 根。如果这堆香蕉少于 s 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。编写程序计算她可以在 H 小时内吃掉所有香蕉的最小速度 s 。【华为、字节跳动、招商银行、中国移动、谷歌、苹果面试题】



常规（暴力）解法

➤ s 的最小值1, s 的最大值 $\max\{p[i]\}$

```
for(s=1; s<=max{p}; s++){
    if(以速度s能在H时间内吃完香蕉) return s;
```

```
bool CanEat(int p[], int n, int s, int H){
    long time = 0;
    for(int i=0; i<n; i++){
        time += ceil(1.0*p[i]/s);
    }
    return time<=H;
}
```

时间复杂度
 $O(nm)$
 $m=\max\{p[i]\}$

相当于有一个新的数组，下标是 s ，元素值是以速度 s 能否在 H 小时内吃完香蕉。暴力方案相当于从左往右扫描数组，找第一个元素值 ≥ 1 的位置

以速度 s 能否在 H 小时内吃完香蕉
0 表示不能；1 表示能

0	0	1	1	1
1	...	s	...	$\max\{p\}$

新策略——二分搜索答案

- 可通过**二分查找**来确定 s 。
- 找满足条件的最小 s （在下面递增有序的数组里找元素值 ≥ 1 的第一个位置）。

以速度 s 能否在 H 小时内吃完香蕉
0表示不能；1表示能

0	0	1	1	1
1	...	s	...	$Max\{p\}$

二分搜索答案

```
int MinEatingSpeed(int p[], int n, int H) {  
    int low=1, high=-1; //high存max{p[i]}  
    for(int i=0;i<n;i++) if(p[i]>high) high=p[i];  
    while(low<=high){  
        int mid = (low+high)/2;  
        if(CanEat(p,n,mid,H)) high = mid-1;  
        else low = mid+1;  
    }  
    return low;  
}
```

时间复杂度
 $O(n \log m)$
 $m = \max\{p[i]\}$

0	0	1	1	1
1	...	mid	...	$Max\{p\}$



自愿性质OJ练习题

- ✓ [LeetCode 34](#) (找第一/最后等于K的位置)
- ✓ [LeetCode 35](#) (找 $\geq K$ 的第一个位置)
- ✓ [LeetCode 875](#) (珂珂吃香蕉)