# 线性结构查找

顺序查找

对半查找

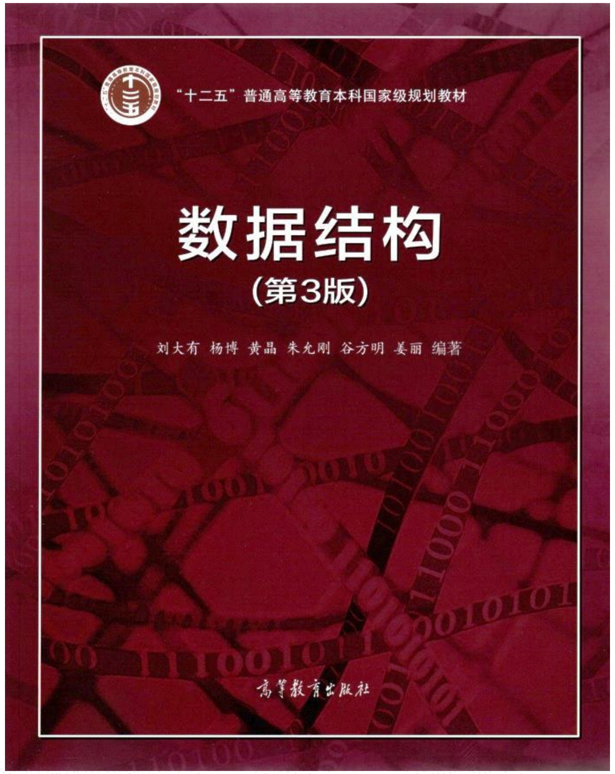
斐波那契查找

插值查找

分块查找

再谈对半查找

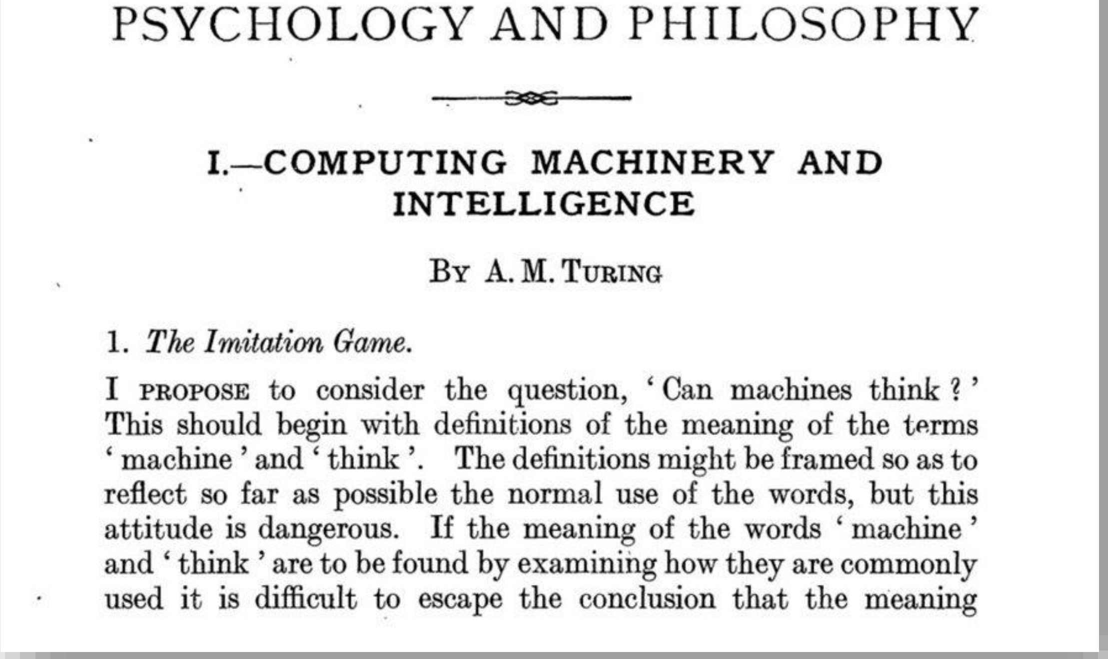
跳跃表




数据之法
结构之美
算法之道

Last updated on 2023.12

zhuyungang@jlu.edu.cn



PSYCHOLOGY AND PHILOSOPHY

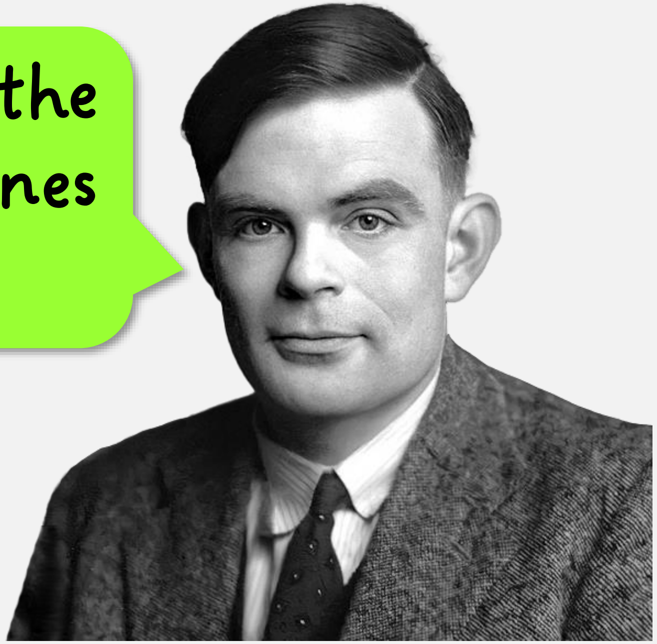
I.—COMPUTING MACHINERY AND INTELLIGENCE

By A. M. Turing

1. *The Imitation Game.*

I propose to consider the question, 'Can machines think?' This should begin with definitions of the meaning of the terms 'machine' and 'think'. The definitions might be framed so as to reflect so far as possible the normal use of the words, but this attitude is dangerous. If the meaning of the words 'machine' and 'think' are to be found by examining how they are commonly used it is difficult to escape the conclusion that the meaning

I propose to consider the question "Can machines think?"



Alan Turing

The question of whether computers can think is like the question of whether submarines can swim.

Edsger Dijkstra

# 查找的基本概念

A

- **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

A

在线性表$R_1,R_2,\ldots,R_n$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$R_i$ ($1\le i \le n$)，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;
}
```

时间复杂度
$O(n)$

# 线性结构查找

顺序查找

**对半查找**

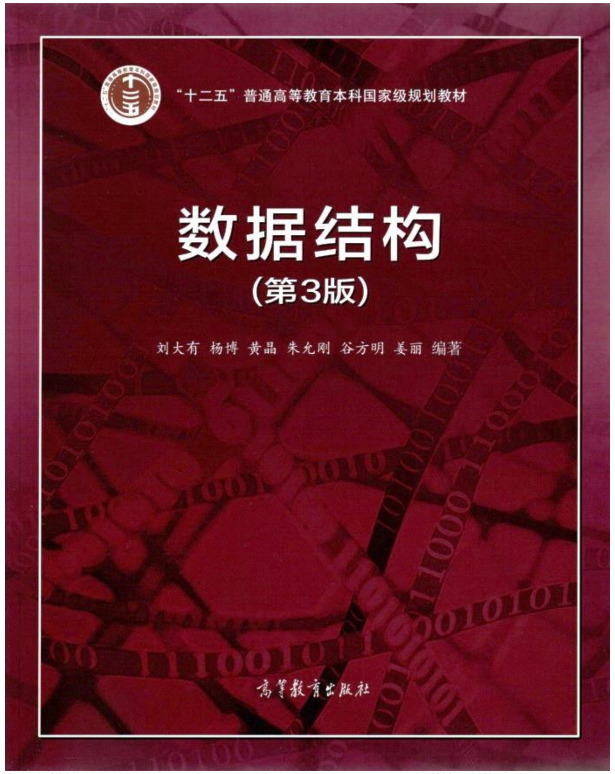
斐波那契查找

插值查找

分块查找

再谈对半查找

跳跃表



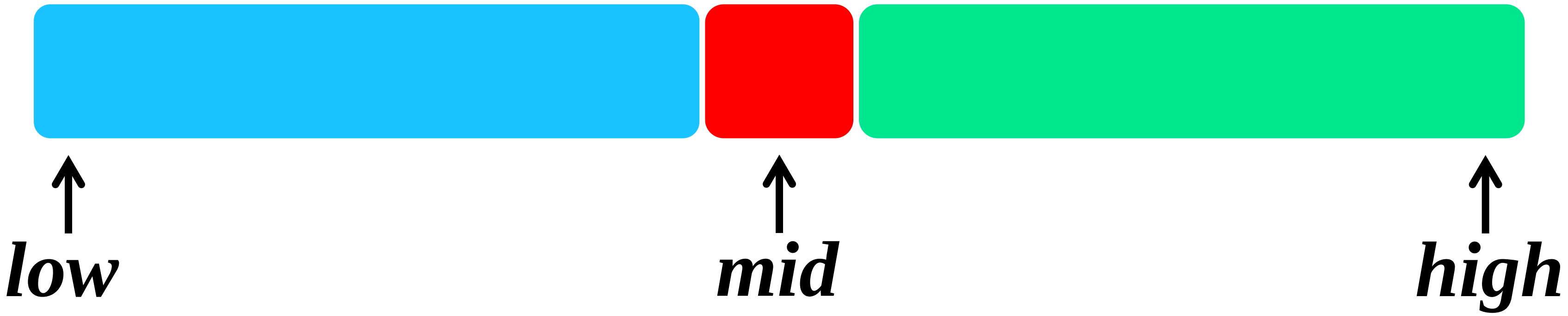

数据之法
结构之美
算法之道



zhuyungang@jlu.edu.cn

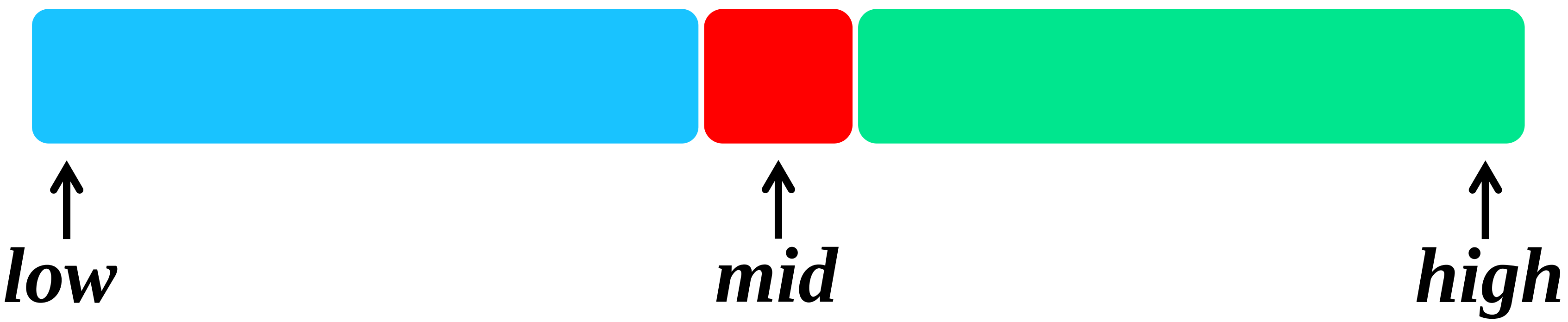
# 有序表的二分查找

- 有序表$R_{low}, R_{low+1}, \ldots, R_{high}$按照关键词递增有序。
- 选取一个**位置mid** ($low \le mid \le high$)，比较$K$和$R_{mid}$，若:
  - ✓ $K < R_{mid}$，[ $K$只可能在$R_{mid}$左侧]
  - ✓ $K = R_{mid}$，[ 查找成功结束 ]
  - ✓ $K > R_{mid}$，[$K$只可能在$R_{mid}$右侧 ]
- 使用不同的规则确定mid，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。



# 对半（折半）查找

- K与待查表的中间记录进行比较，即$mid \leftarrow \lfloor (low+high)/2 \rfloor$
- 每次迭代可将查找范围缩小一半。

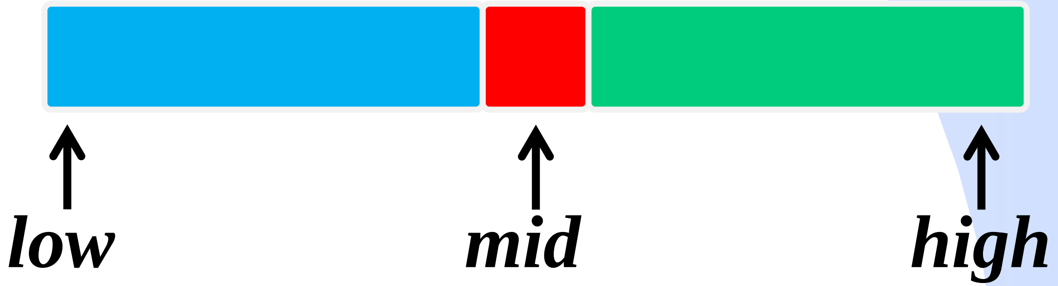


# 对半查找

A

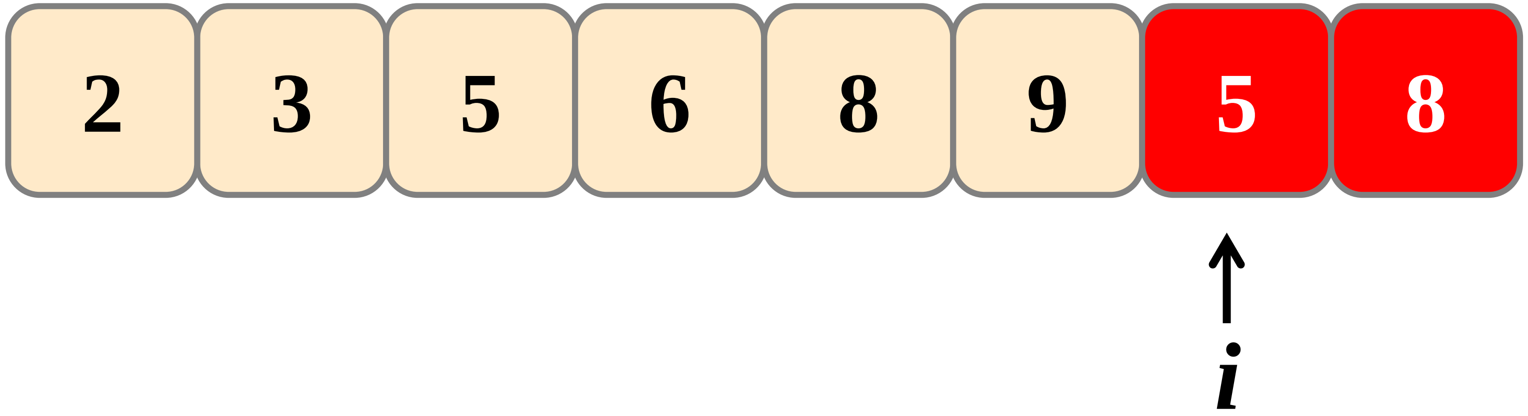
```
int BinarySearch(int R[],int n, int K){
   //在数组R中对半查找K，R中关键词递增有序
   int low = 1, high = n, mid;
   while(low <= high){
      mid=(low+high)/2;
      if(K<R[mid]) high=mid-1;         //在左半部分查找
      else if(K>R[mid]) low=mid+1;     //在右半部分查找
      else return mid;                 //查找成功
   }
   return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

时间复杂度
O(log*n*)



# 对半插入排序

- 插入$R_i$时，基于对半查找确定插入的位置，可将每次插入的关键词比较次数降为O(log*n*)。
- 由于元素移动次数仍为O(*n*)，故排序算法总时间复杂度仍为O($n^2$)，但常数更低。



# 练习

A

对同一序列分别进行对半插入排序和直接插入排序，两者之间可能的不同之处是_______。【考研题全国卷】

A.排序的总趟数
B.元素的移动次数
C.使用辅助空间的数量
D.元素的比较次数

# 例：查找 $K$=96 时对半查找过程（第1次比较）

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***R*** | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |
| | ↑ *low* | | | | ↑ *mid* | | | | | ↑ *high* |

# 例：查找 $K$=96 时对半查找过程（第2次比较）

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***R*** | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |
| | | | | | | ↑ *low* | | ↑ *mid* | | ↑ *high* |

# 例：查找 $K$=96 时对半查找过程（第3次比较）

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***R*** | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |
| | | | | | | | | | ↑ *low* | ↑ *high* |
| | | | | | | | | | ↑ *mid* | |

# 例：查找 *K*=96 时对半查找过程（第4次比较）

*A*

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***R*** | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |

↑ *high*
↑ *low*
↑ *mid*

查找成功

# 例：查找 $K$=58 时对半查找过程（第1次比较）

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $R$ | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |
| | ↑ | | | | ↑ | | | | | ↑ |
| | *low* | | | | *mid* | | | | | *high* |

# 例：查找 $K$=58 时对半查找过程（第2次比较）

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $R$ | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |
| | | | | | | ↑ | | ↑ | | ↑ |
| | | | | | | *low* | | *mid* | | *high* |

# 例：查找 *K*=58 时对半查找过程（第3次比较）

A

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***R*** | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |
| | | | | | | *low* | *high* | | | |
| | | | | | | *mid* | | | | |

# 例：查找 *K*=58 时对半查找过程（第3次比较）

A

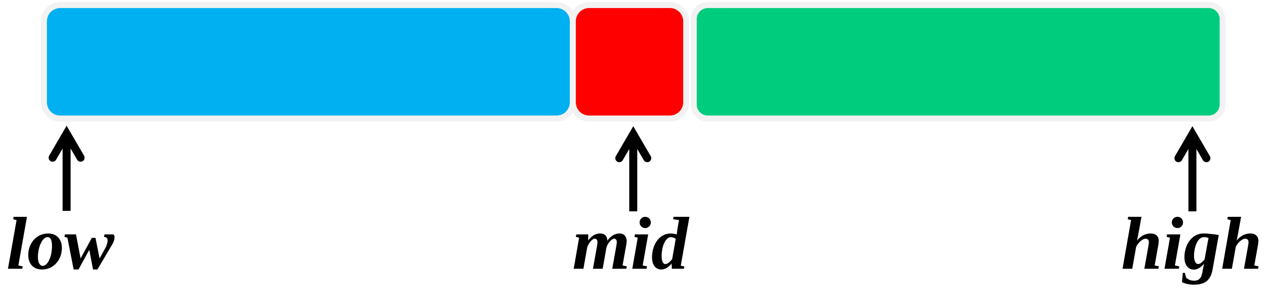
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***R*** | 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |
| | | | | | ↑ *high* | ↑ *low* | | | | |

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$R_{low}$，$R_{low+1}$，…，$R_{high}$，对半查找的二叉判定树$T(low, high)$的是按如下递归定义的扩充二叉树：

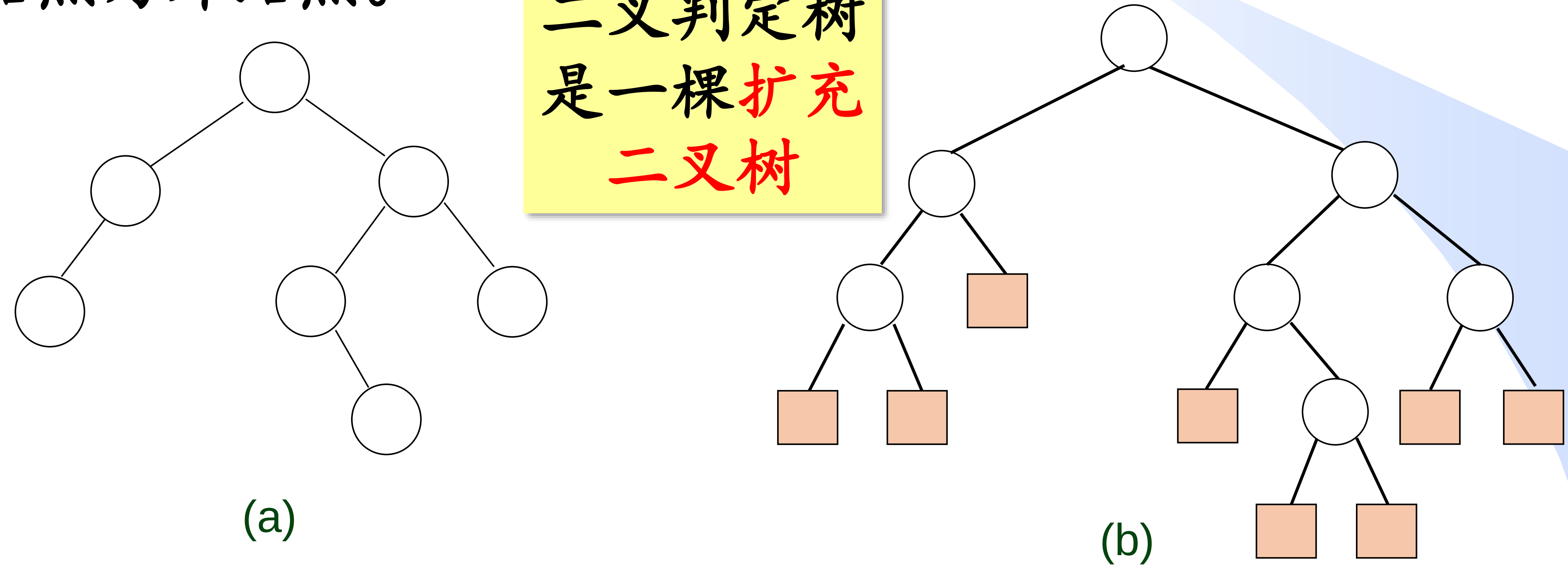
- 当$high-low+1 \le 0$时：$T(low, high)$为空；
- 当$high-low+1 > 0$时，令$mid = \lfloor (low+high)/2 \rfloor$
  - ✓$T(low, high)$的根结点是$mid$；
  - ✓根结点的左子树是$R_{low},\ldots,R_{mid-1}$对应的二叉判定树；
  - ✓根结点的右子树是$R_{mid+1},\ldots,R_{high}$对应的二叉判定树。



# 回顾：扩充二叉树

在二叉树中空指针的位置，都增加特殊的结点（空叶结点），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。

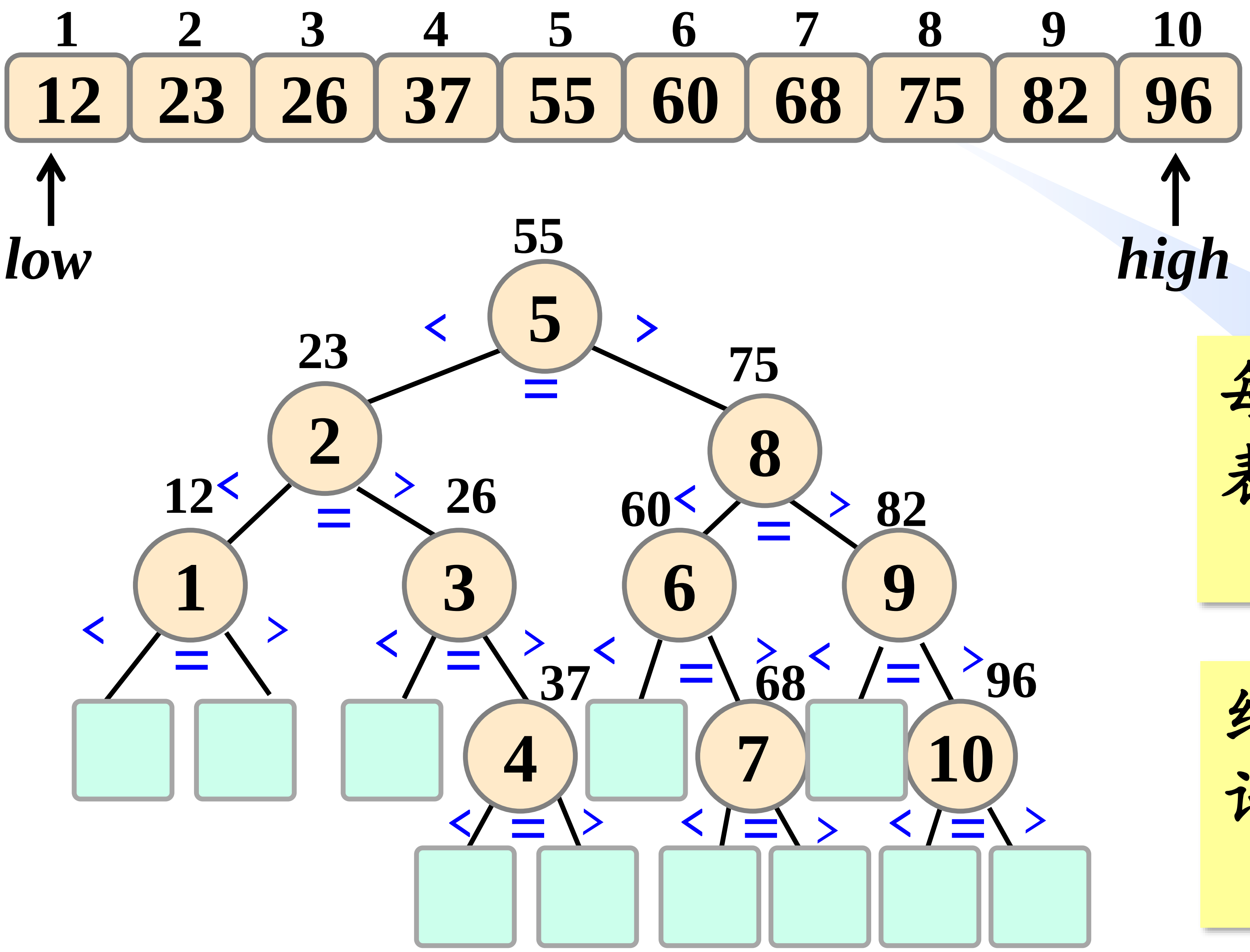
二叉判定树是一棵**扩充二叉树**




二叉树及其对应的扩充二叉树

# 对半查找二叉判定树*T*(1,10)

A




每个圆圈结点表示与关键词的比较

结点值：关键词比较的位置（下标）

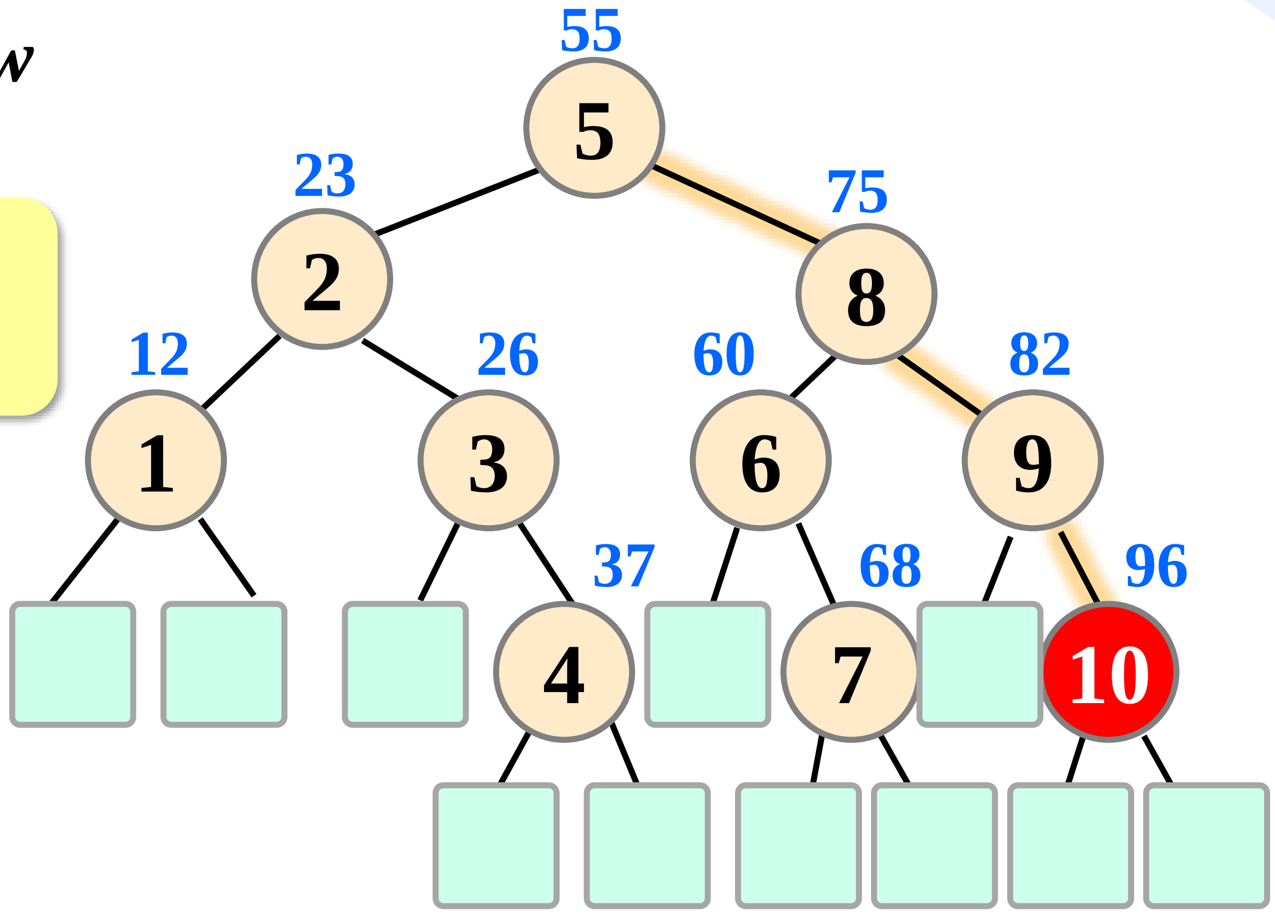
# 二叉判定树——查找成功情况示例

A

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |

↑ *low* ↑ *high*

查找*K*=96
成功情况




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。**

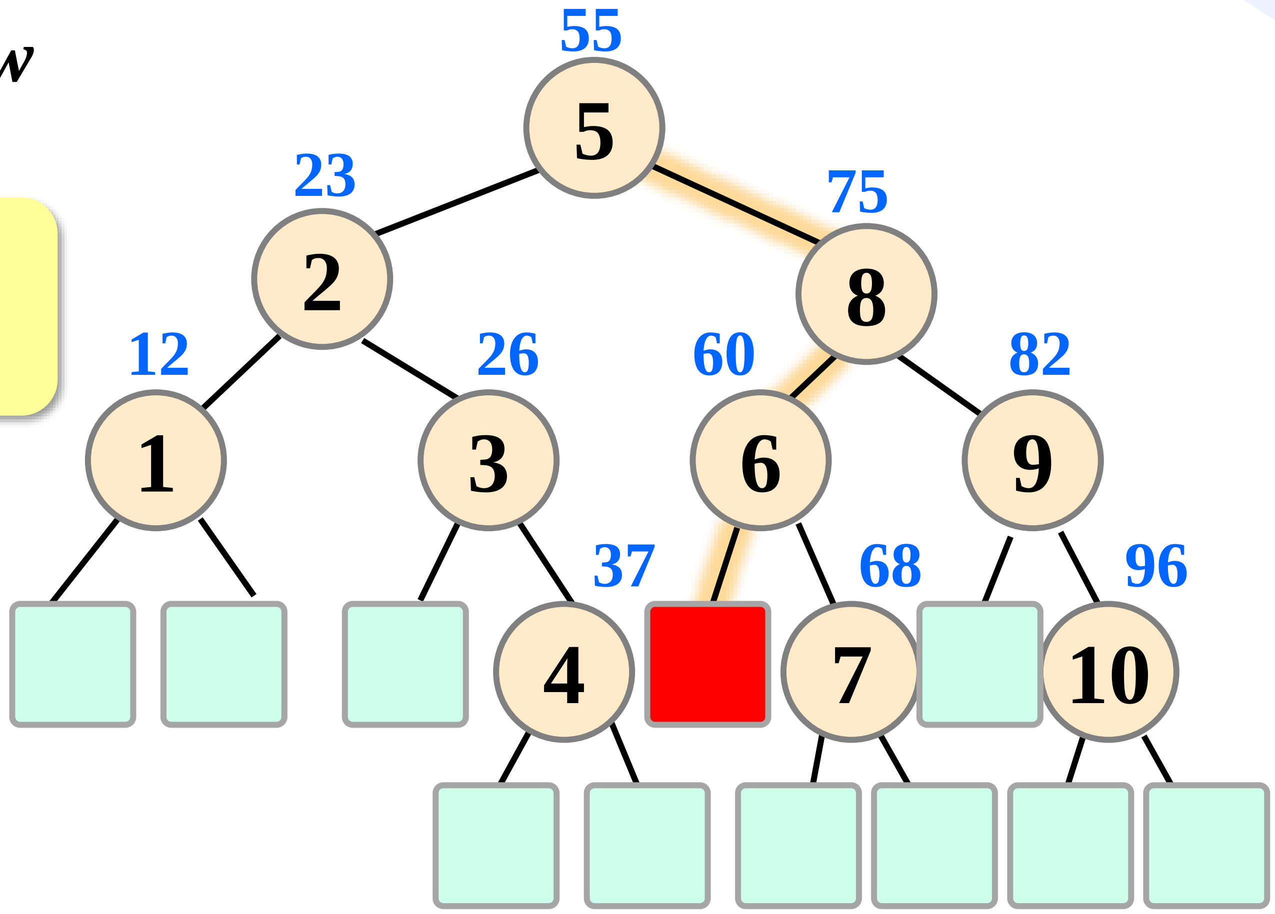
# 二叉判定树——查找成功情况示例

A

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 96 |

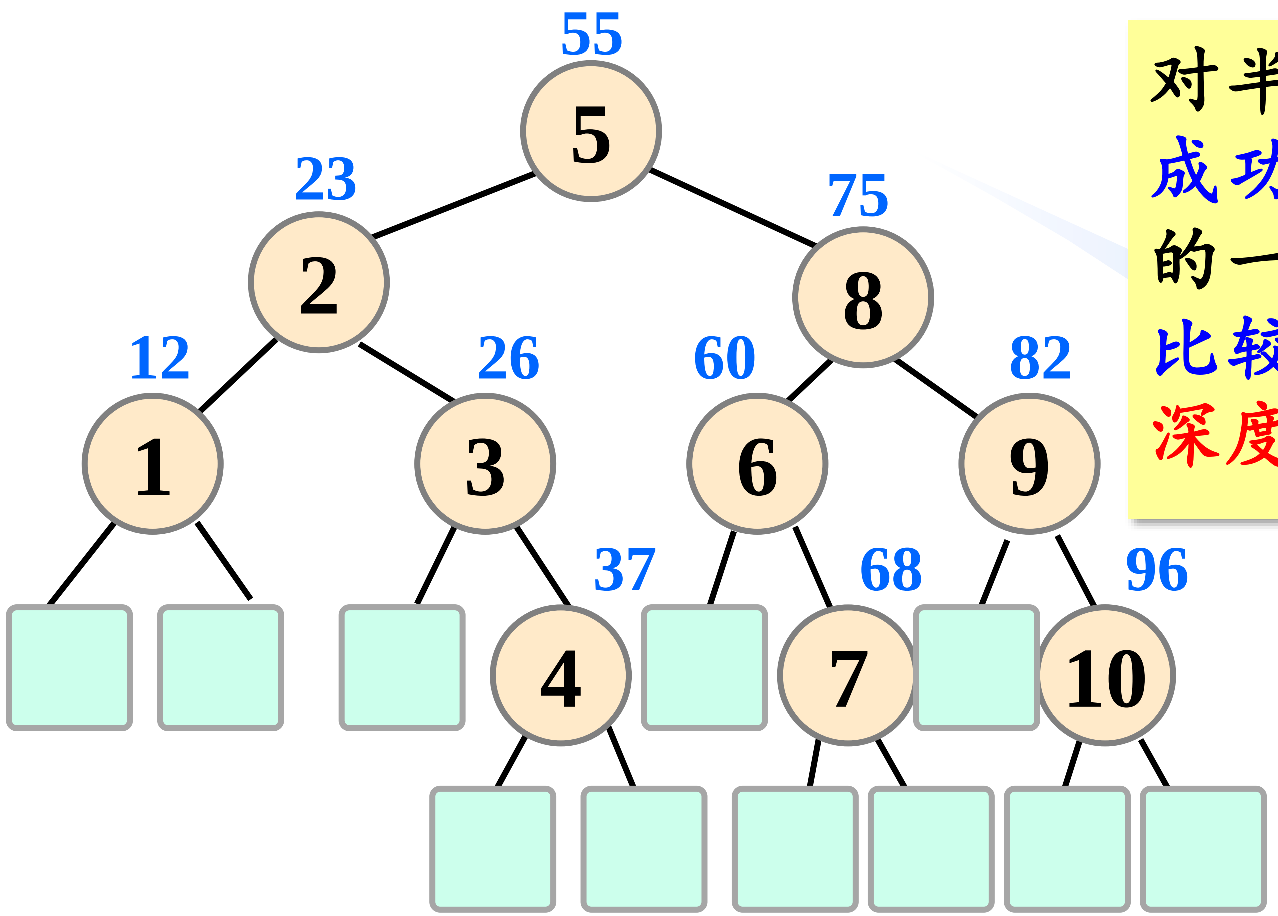
*low* *high*

查找*K*=58
失败情况




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。**

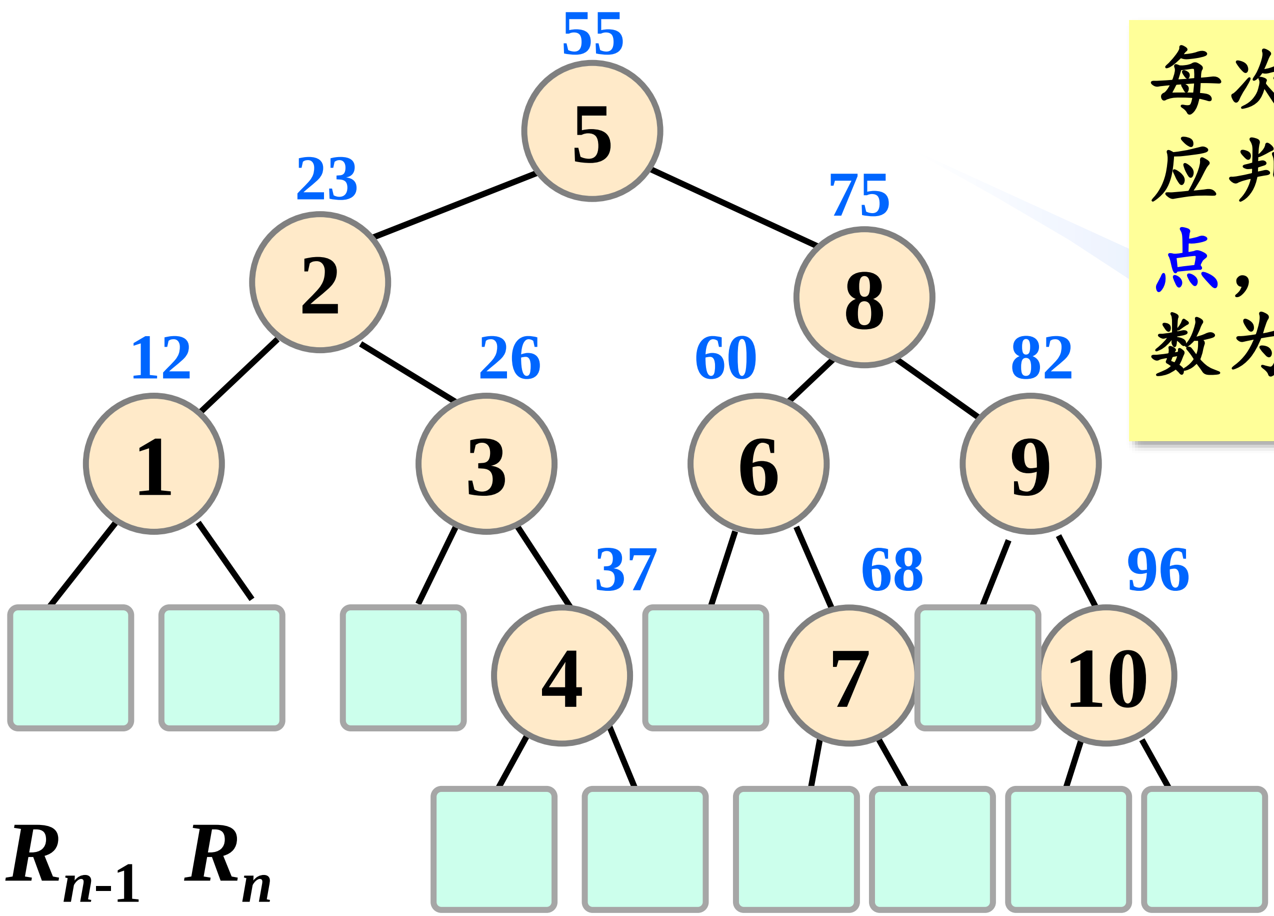
# 查找成功的平均查找长度(Average Search Length, ASL) A




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

$$ASL_{succ}=\frac{1}{10}(1\times1\ +2\times2\ \ +4\times3\ +3\times4)=2.9$$

# 查找失败的平均查找长度(Average Search Length, ASL) A



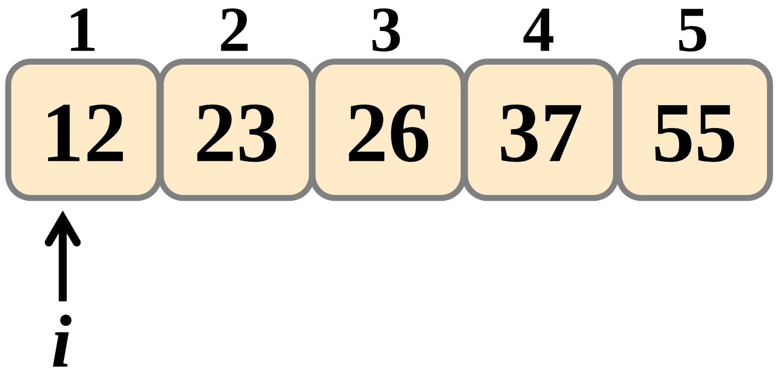

每次不成功的查找对应判定树的一个外结点，关键词的比较次数为该结点的深度。

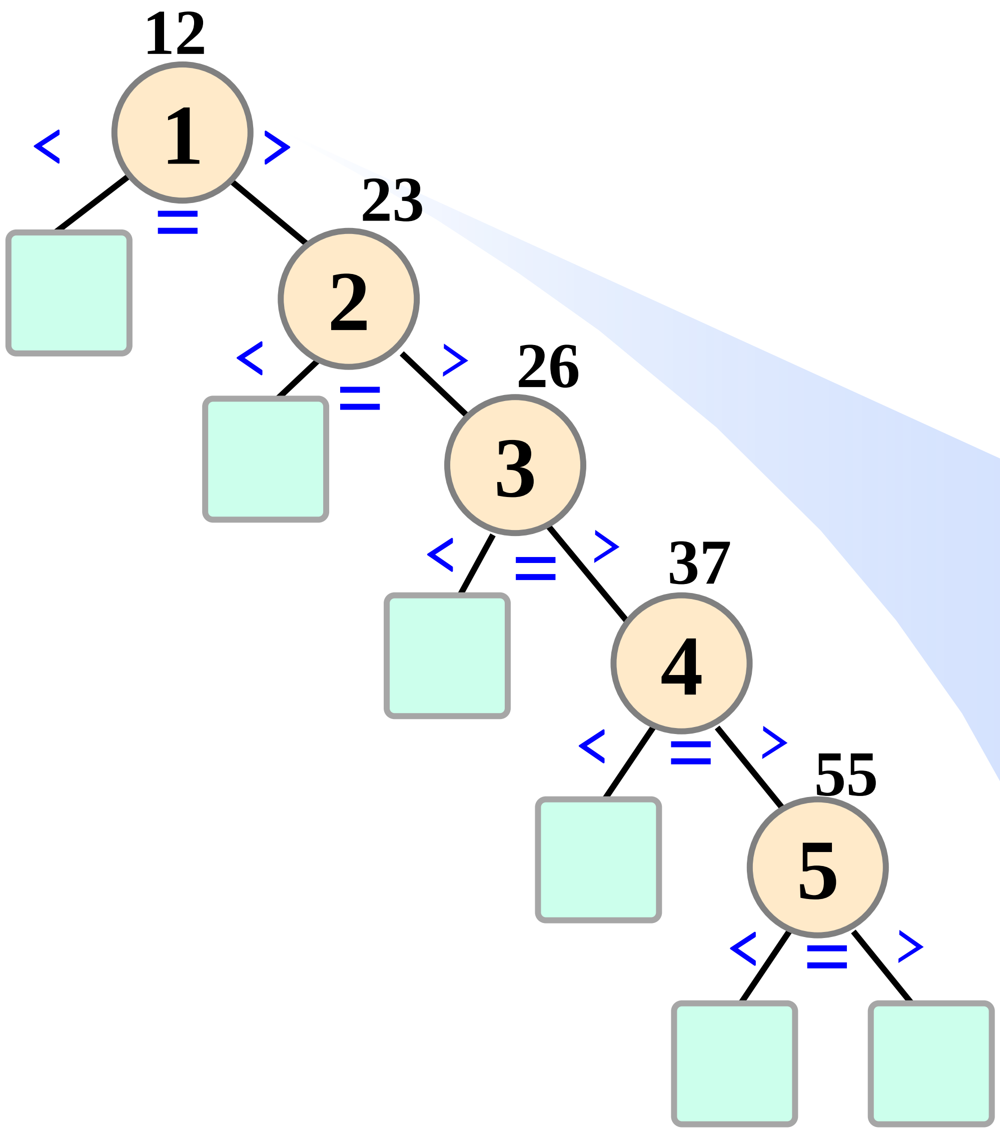
$R_1$ $R_2$ $R_3$ … $R_{n-1}$ $R_n$

$$ASL_{unsucc} = \frac{1}{11}(5\times3 + 6\times4) = 39/11 \approx 3.5$$

# 课下思考：对有序数组进行顺序查找的二叉判定树

A



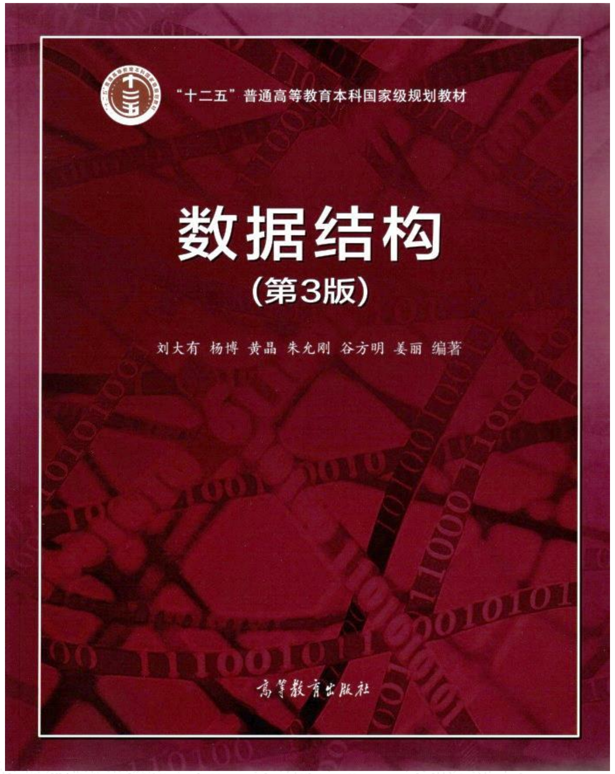



# 以下哪个分支语句执行效率更高?

D

```
switch(a){
    case 1000: f1(); break;
    case 2000: f2(); break;
    case 2500: f3(); break;
    case 5000: f5(); break;
    case 7000: f6(); break;
    case 7500: f7(); break;
    case 8000: f8(); break;
    case 9000: f9(); break;
    default: f10();
}
```

```
if(a==1000) f1();
else if(a==2000) f2();
else if(a==2500) f3();
else if(a==5000) f5();
else if(a==7000) f6();
else if(a==7500) f7();
else if(a==8000) f8();
else if(a==9000) f9();
else f10();
```




# 线性结构查找

顺序查找

对半查找

**斐波那契查找**

插值查找

分块查找

再谈对半查找

跳跃表

数据之法
结构之美
算法之道

Last updated on 2023.12

zhuyungang@jlu.edu.cn

B

# 斐波那契（Fibonacci）查找

➢ 斐波那契序列： $F_0=0$，$F_1=1$，$F_k=F_{k-1}+F_{k-2}$ , $k \geq 2$

| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | … |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | … |

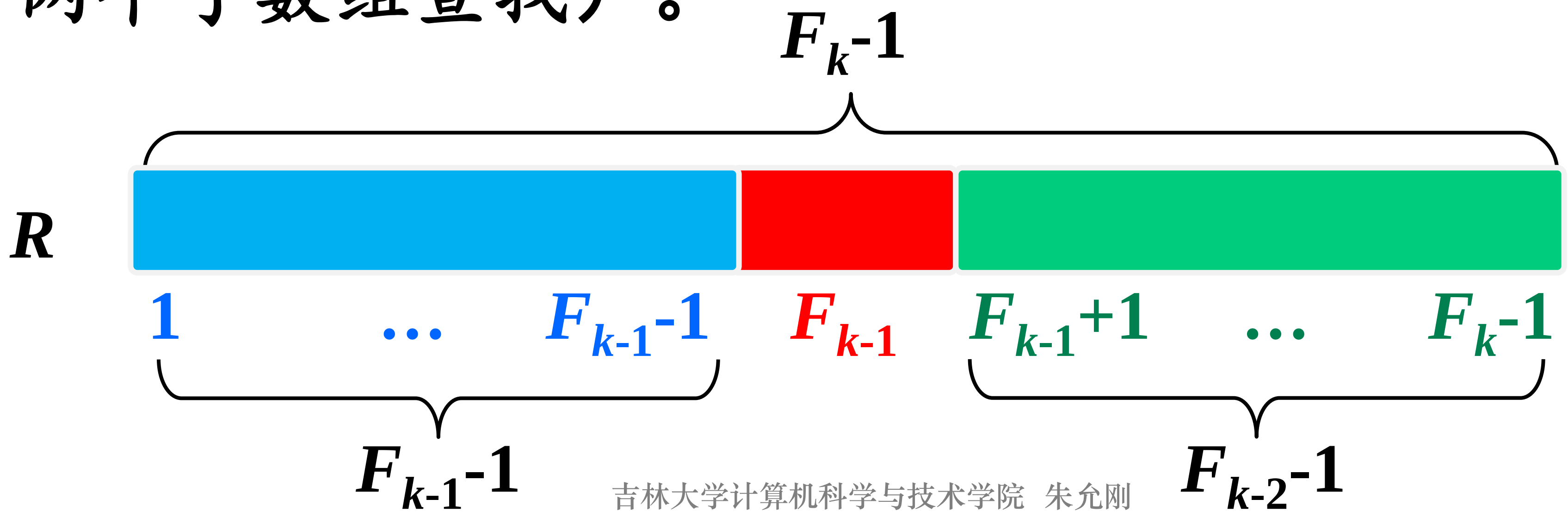
➢ 斐波那契查找 ：折半查找的改进，以斐波那契序列的划分代替对半查找的均匀划分。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 23 | 26 | 37 | 55 | 60 | 68 | 75 | 82 | 86 | 90 | 96 |
| ↑ *low* | | | | | | | | | | | ↑ *high* |

B

# 斐波那契查找

- 设有序数组$R$长度为$F_k$-1。下标 $F_{k-1}$ 将数组分为三部分：
  - ✓ 左子数组$R[1]\ldots R[F_{k-1}-1]$； 长度为$F_{k-1}-1$
  - ✓ $R[F_{k-1}]$；
  - ✓ 右子数组$R[F_{k-1}+1]\ldots R[F_k-1]$；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斐波那契数减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）。



# 斐波那契查找

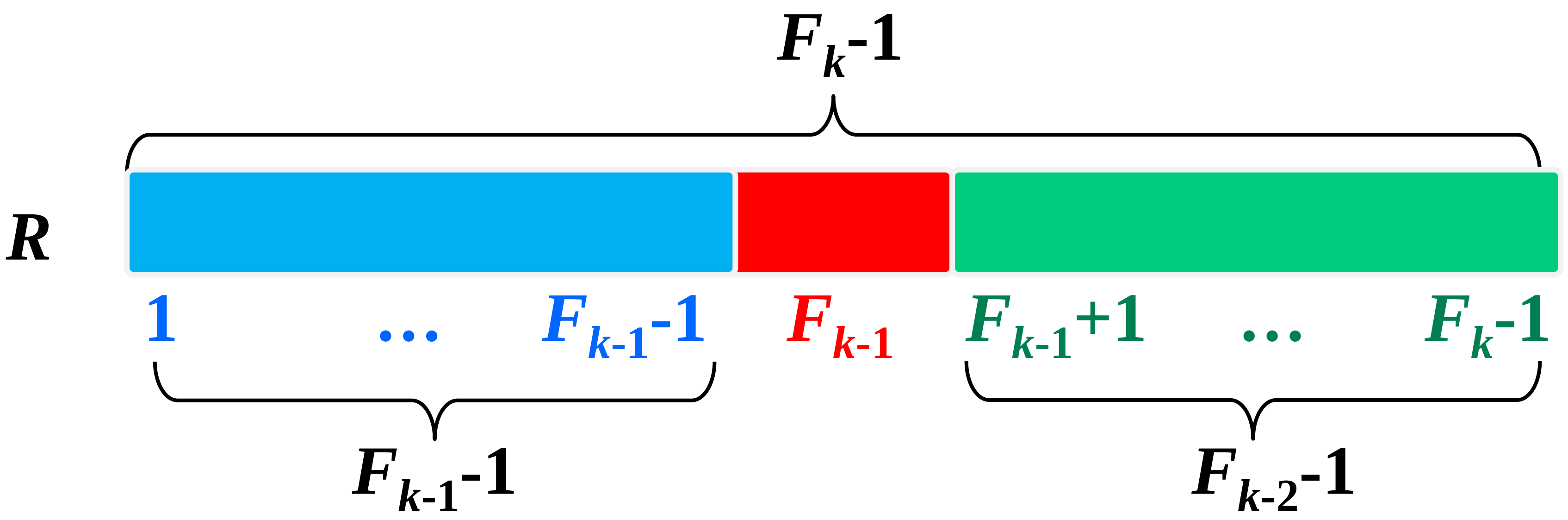
B

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

- $K < R[F_{k-1}]$：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；
- $K = R[F_{k-1}]$：则查找成功。

$k$-1阶斐波那契查找

$k$-2阶斐波那契查找




对长度为$F_k$-1的数组进行斐波那契查找，不妨简称为$k$阶斐波那契查找

# 例：查找26

**B**

- 斐波那契序列

| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | … |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | … |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 23 | 26 | 37 | 56 | 60 | 68 | 75 | 82 | 86 | 90 | 96 |
| ↑ *low* | | | | | | | ↑ *mid* | | | | ↑ *high* |

# 例：查找26

B

- 斐波那契序列

| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | … |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | … |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 23 | 26 | 37 | 56 | 60 | 68 | 75 | 82 | 86 | 90 | 96 |
| ↑ low | | | | ↑ mid | | ↑ high | | | | | |

# 例：查找26

B

➢ 斐波那契序列

| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | … |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | … |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 23 | 26 | 37 | 56 | 60 | 68 | 75 | 82 | 86 | 90 | 96 |

↑ *low* ↑ *mid* ↑ *high*

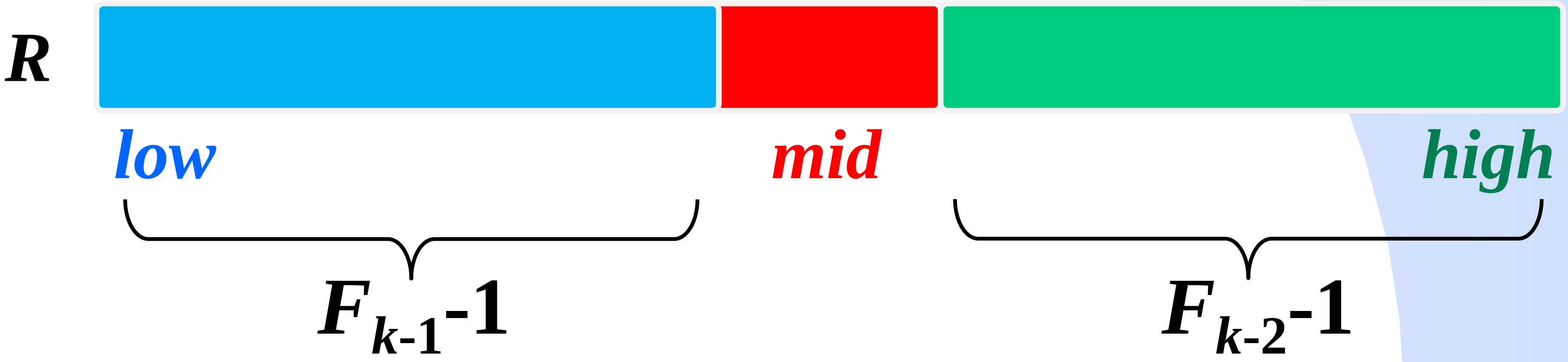
# 斐波那契查找

B

```
int FibSearch(int R[], int n, int K, int F[], int k){
//对有序文件R1,…,Rn进行k阶斐波那契查找，假定斐波那契数列存于数组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
    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;
}
```

对左侧子数组进行 $k$-1阶斐波那契查找

对右侧子数组进行 $k$-2阶斐波那契查找



# 斐波那契查找

B

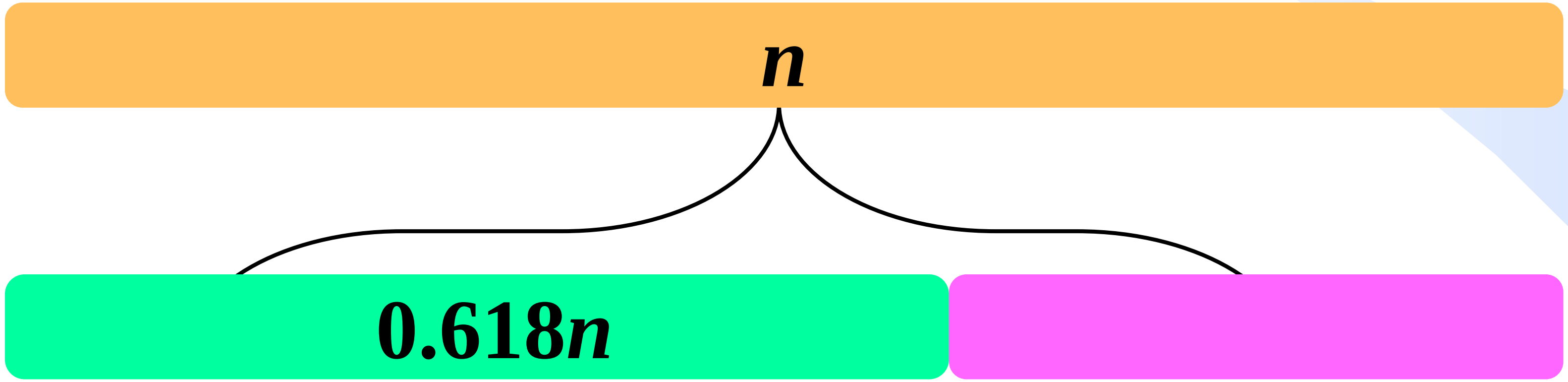
```
int FibSearch(int R[], int n, int K, int F[], int k){
//对有序文件R1,…,Rn进行k阶斐波那契查找，假定斐波那契数列存于数组F
//若查找成功，返回K在R中下标，否则返回-1
    int low=1,high=n;
    while(low <= high){
        int mid=low+F[k-1]-1;
        if(K<R[mid]) {high=mid-1; k--;}
        else if(K>R[mid]) {low=mid+1; k-=2;}
        else (mid<n)? mid:n;
    }
    return -1;
}
```

若数组长度$n$不等于斐波那契数减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$R[F_k\text{-}1]$用$R[n]$补全

| $R$ | 15 | 16 | 23 | 56 | 98 | 98 | 98 |
|---|---|---|---|---|---|---|---|
| | 1 | … | … | … | $n$ | … | $F_k$-1 |

# 斐波那契查找

- 本质：在黄金分割点处对数组划分。




$$\lim_{k\to\infty}\frac{F_{k-1}}{F_k}=0.618\ldots$$

$F_k$为第$k$个斐波那契数

# 斐波那契查找总结

B

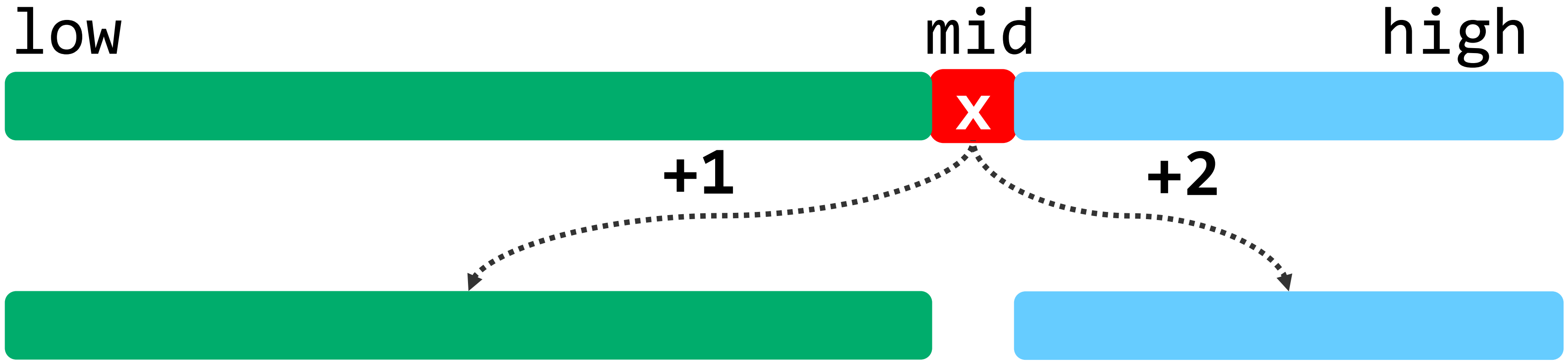
- 平均和最坏情况下的时间复杂性为$O(\log_2 n)$。
- 总体运行时间略快于对半查找算法。
- 算法不涉及乘除法，而只涉及加减法。

```
int FibSearch(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;
}
```

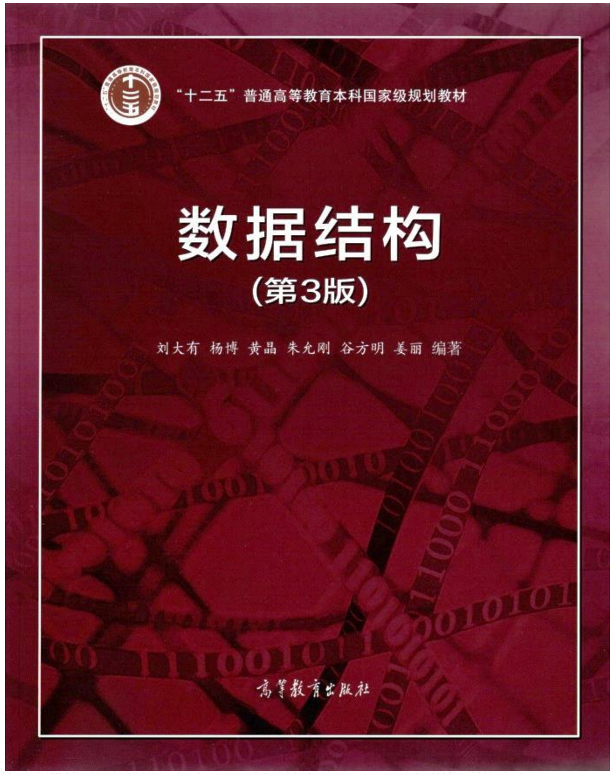
# 斐波那契查找总结

- **左区间**比**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;
}
```



# 线性结构查找




- 顺序查找
- 对半查找
- 斐波那契查找
- **插值查找**
- 分块查找
- 再谈对半查找
- 跳跃表

数据之法
结构之美
算法之道

Last updated on 2023.12

zhuyungang@jlu.edu.cn

# 插值查找

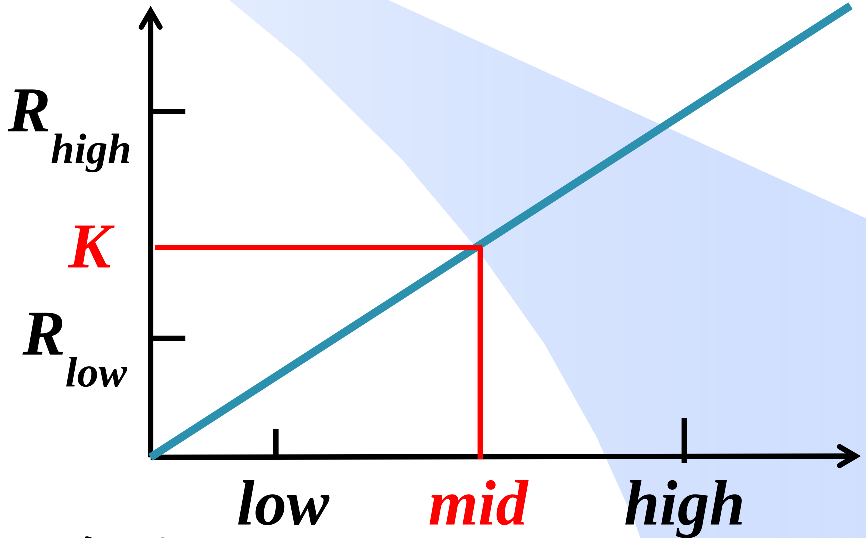
- 假设：有序数组*R*中元素均匀随机分布，例如

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

- 于是，$R_{low}\ldots R_{high}$内各元素应大致呈线性增长关系

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

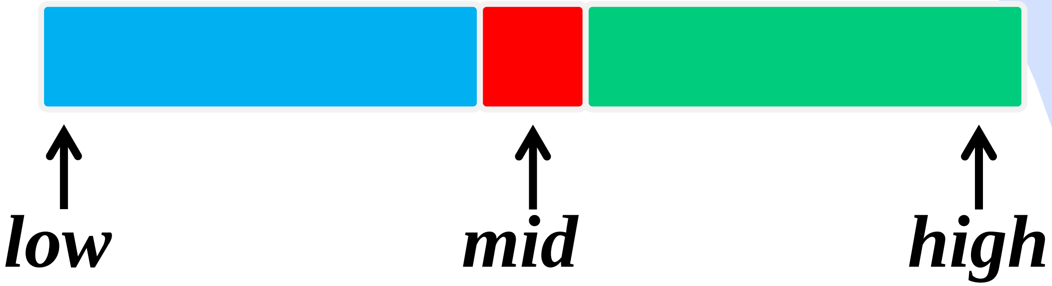
$$mid\approx low+\frac{K-R_{low}}{R_{high}-R_{low}}(high-low)$$




- 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：$mid=\dfrac{low+high}{2}=low+\dfrac{1}{2}(high-low)$

# 插值查找

```c
int InterpolationSearch(int R[], int n, int K){
    int low=1, high=n, mid;
    while(low<=high && K>=R[low] && K<=R[high]){
        if(R[low]==R[high]) return low;
        mid=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
哈佛大学博士
图灵奖获得者
中国科学院院士
美国科学院外籍院士
加州伯克利教授（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$\sqrt{n}$。**

AC Yao and FF Yao. The Complexity of Searching an Ordered Random Table. *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*. 173-177, 1976.

# 插值查找时间复杂度

A

| 初始长度 | 第1次迭代后 | 第2次迭代后 | 第3次迭代后 | | 第$k$次迭代后 |
|---|---|---|---|---|---|
| $n$ | $\sqrt{n}$ | $\sqrt{\sqrt{n}}$ | $\sqrt{\sqrt{\sqrt{n}}}$ | …… | 2 |

- 元素比较的次数 = 迭代的次数。
- 假定k次迭代，则 $n^{\left(\frac{1}{2}\right)^k} = 2$
- 平均时间复杂度O(loglog$n$).
- 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

A

- 从O(log$n$)到O(loglog$n$)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

  $\log n = \log 2^{32} = 32$

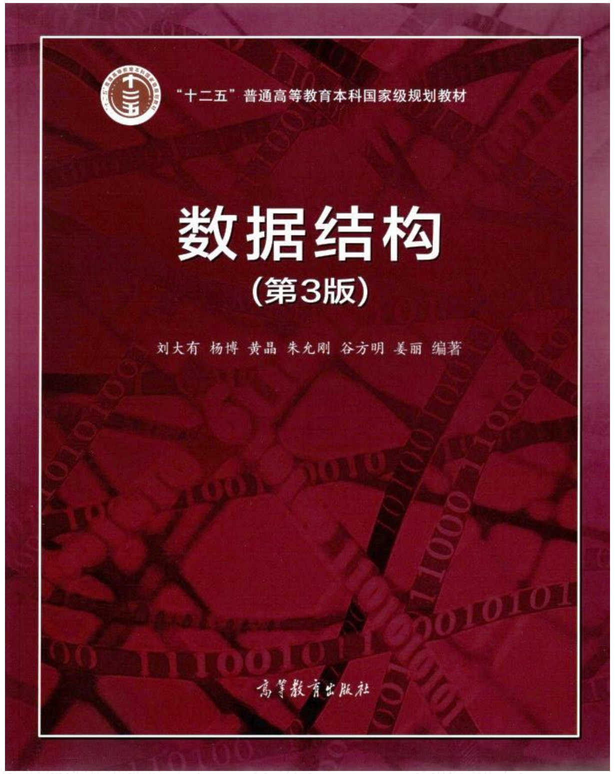
  $\log\log n= \log 32 = 5$

- 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，然后再进行对半查找或顺序查找。

# 二分查找总结

- 优点：平均查找效率不超过O(log*n*)，比顺序查找高。
- 缺点：
  - ✓适用于有序数组，对有序链表难以进行二分查找。
  - ✓适用于静态查找场景，若元素动态变化（频繁增删）后，为了维持数组有序，需要O(*n*)时间调整，与顺序查找相比，就没有优势了。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 12 | 23 | 26 | 37 | 55 | 60 | 68 |




# 线性结构查找

顺序查找

对半查找

斐波那契查找

插值查找

**分块查找**

再谈对半查找

跳跃表

数据之法
结构之美
算法之道

Last updated on 2023.12

zhuyungang@jlu.edu.cn

# 分块查找

索引表

| | | | |
|---|---|---|---|
| 最大关键字 | 22 | 50 | 86 |
| 起始地址 | 1 | 7 | 13 |

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 12 | 13 | 8 | 9 | 20 | 33 | 42 | 44 | 38 | 24 | 50 | 60 | 58 | 74 | 51 | 86 | 53 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

第1块　　第2块　　第3块

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

C

# 分块查找

索引表

| | | | |
|---|---|---|---|
| 最大关键字 | 22 | 50 | 86 |
| 起始地址 | 1 | 7 | 13 |

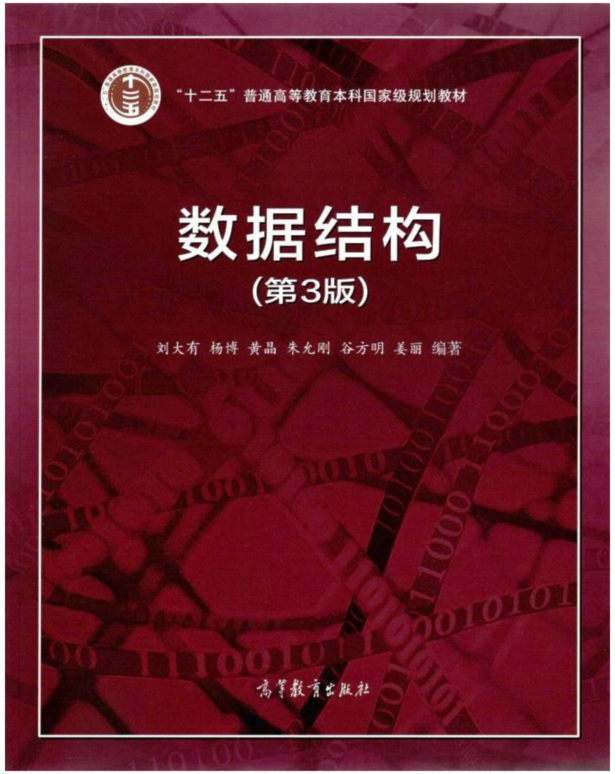
| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 12 | 13 | 8 | 9 | 20 | 33 | 42 | 44 | 38 | 24 | 50 | 60 | 58 | 74 | 51 | 86 | 53 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

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




# 线性结构查找

顺序查找

对半查找

斐波那契查找

插值查找

分块查找

**再谈对半查找**

跳跃表

数据之法
结构之美
算法之道



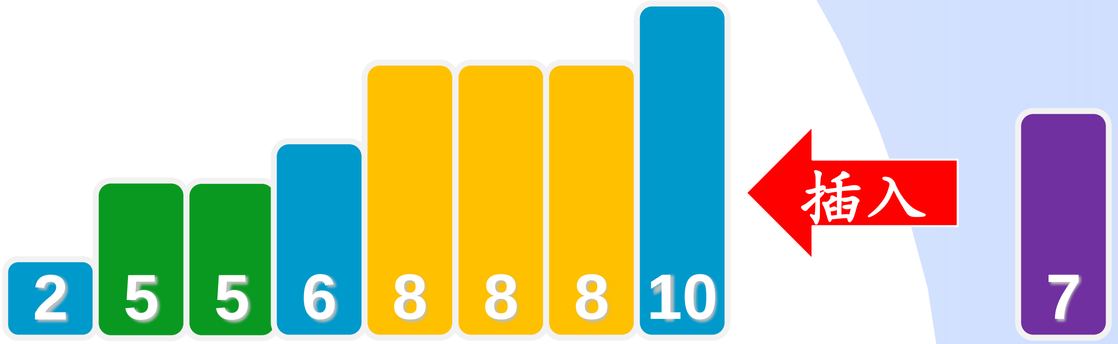
zhuyungang@jlu.edu.cn

# 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;        //在左半部分查找
        else if(K>R[mid]) low=mid+1;    //在右半部分查找
        else return mid;                //查找成功
    }
    return -1; //查找失败
}
```

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

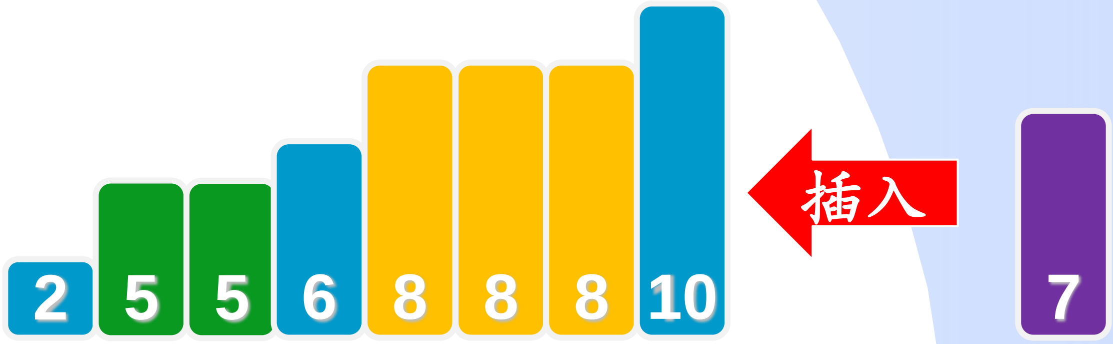
C

```
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;          //在左半部分查找
      else if(K>R[mid]) low=mid+1;      //在右半部分查找
      else return mid;                  //查找成功
   }
   return -1; //查找失败
}
```

返回：

（1）大于等于*K*的第一个位置

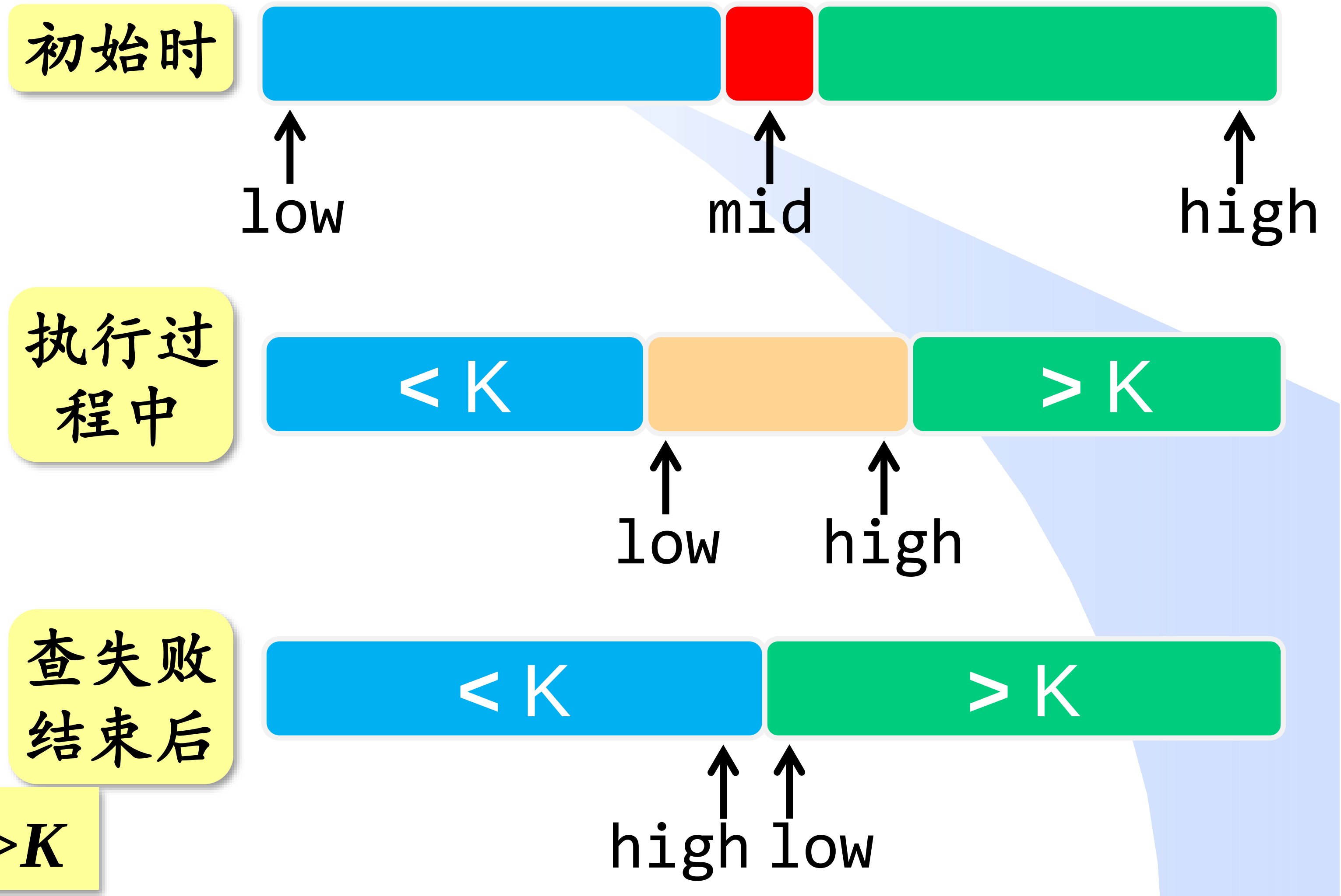
（2）小于等于*K*的最后一个位置



# 进一步审视对半查找过程

C

```
int BinarySearch(int R[],int n, int K){
    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
}
```

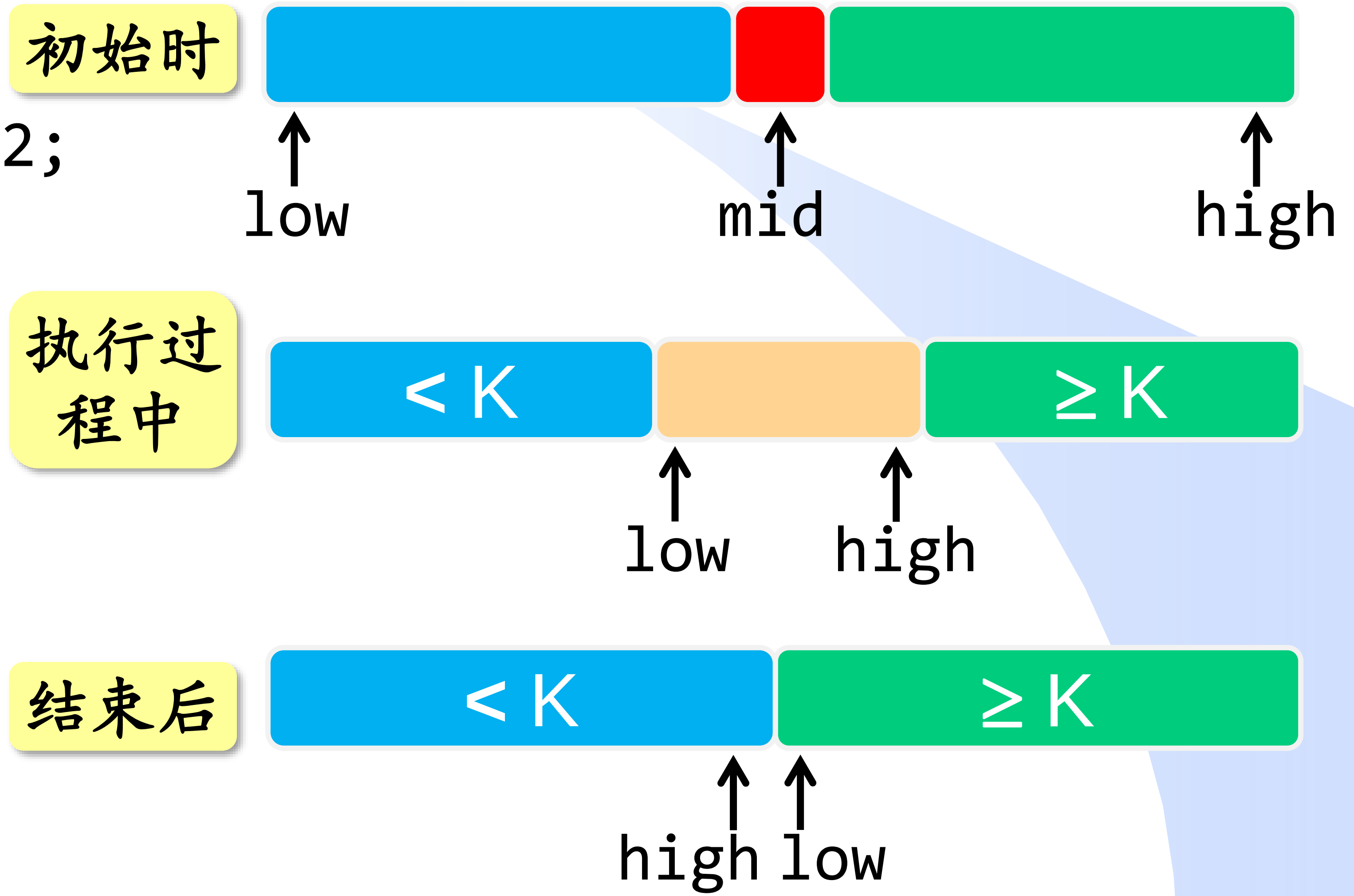



**low的左边<*K*，high的右边>*K***

# 大于等于K的第一个位置

C

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

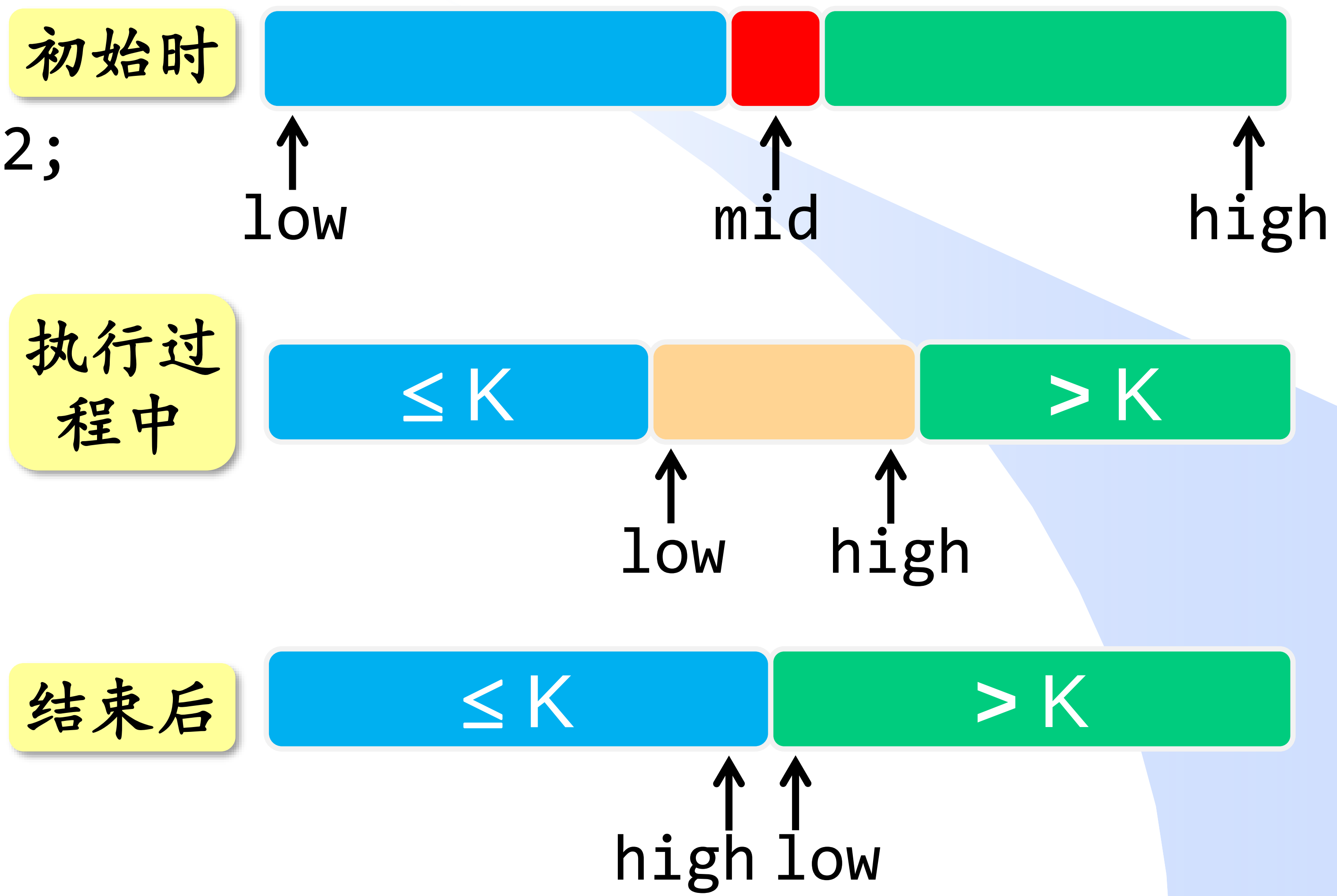



课下思考：若不存在≥K的位置，函数返回何值？

# 小于等于K的最后一个位置

C

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




课下思考：若不存在≤*K*的位置，函数返回何值？

# 例题：第一个/最后一个等于*K*的位置

C

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LeetCode34】

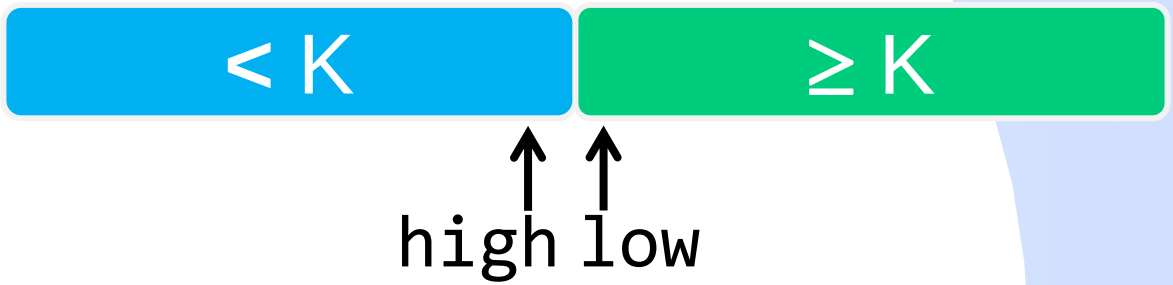
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 6 | 6 | 6 | 8 | 10 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# 第一个等于*K*的位置

**策略：先找≥*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≥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≥K的第一个位置
    if(low<n && R[low]==K) return low;
    return -1;
}
```

时间复杂度
$O(\log n)$



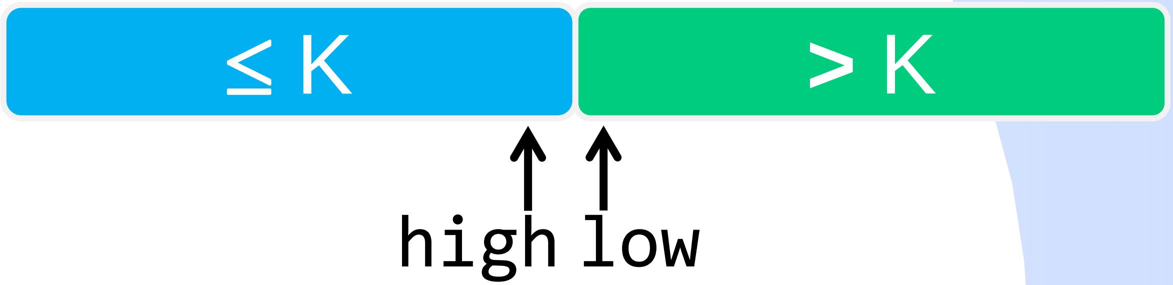
# 最后一个等于*K*的位置

C

策略：先找≤*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≤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≤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*)



# 二分搜索答案

C

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。编写程序计算她最小以什么速度吃香蕉，才能在$H$小时内吃掉所有香蕉。【华为、字节跳动、招商银行、中国移动、谷歌、苹果面试题LeetCode875】

| $p$ | 5 | 3 | 2 | 10 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | … | $n$-1 |

$s$的最小值1，$s$的最大值$m$=max{$p[i]$}

# 常规（暴力）解法

C

➢ $s$的最小值1，$s$的最大值$m$=max{$p$[$i$]}

```
for(s=1; s<=m; 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 | … | $s$ | … | $m$ |

# 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 | … | $s$ | … | $m$ |

# 二分搜索答案

C

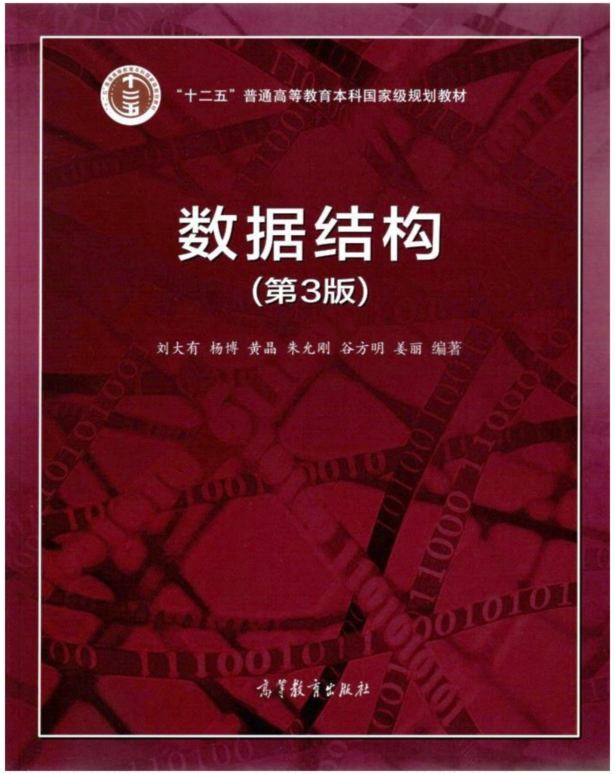
```
int MinEatingSpeed(int p[], int n, int H) {
    int m=-1;  //m存max{p[i]}
    for(int i=0;i<n;i++) if(p[i]>m) m=p[i];
    int low=1, high=m;
    while(low<=high){
        int mid = (low+high)/2;
        if(CanEat(p,n,mid,H)) high = mid-1;
        else low = mid+1;
    }
    return low;
}
```

时间复杂度
$O(n\log m)$
$m=\max\{p[i]\}$

| | | | | |
|---|---|---|---|---|
| **0** | **0** | **1** | **1** | **1** |
| **1** | **…** | ***mid*** | **…** | ***m*** |

# 线性结构查找

顺序查找
对半查找
斐波那契查找
插值查找
分块查找
再谈对半查找
**跳跃表**




数据之法
结构之美
算法之道

Last updated on 2023.12

zhuyungang@jlu.edu.cn

# 跳跃表（*Skip List*）——动机

- 二分查找无法应用于链表。
- 能否借鉴二分查找思想，提高有序链表中元素查找的效率?

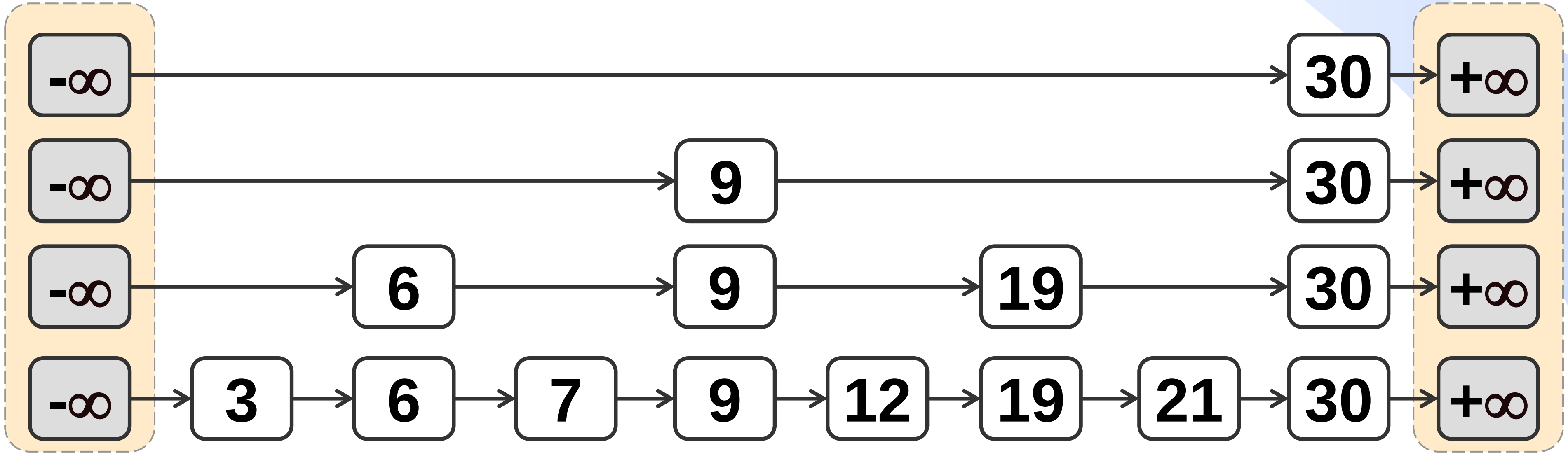
**William Pugh**
**康奈尔大学博士**
**马里兰大学教授**

William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. Proceedings of Workshop on Algorithms and Data Structures, 437-449, 1989.

# 跳跃表（*Skip List*）——动机

C

- 二分查找无法应用于链表。
- 能否借鉴二分查找思想，提高有序链表中元素查找的效率?
- 分层次、相互耦合的多个链表。

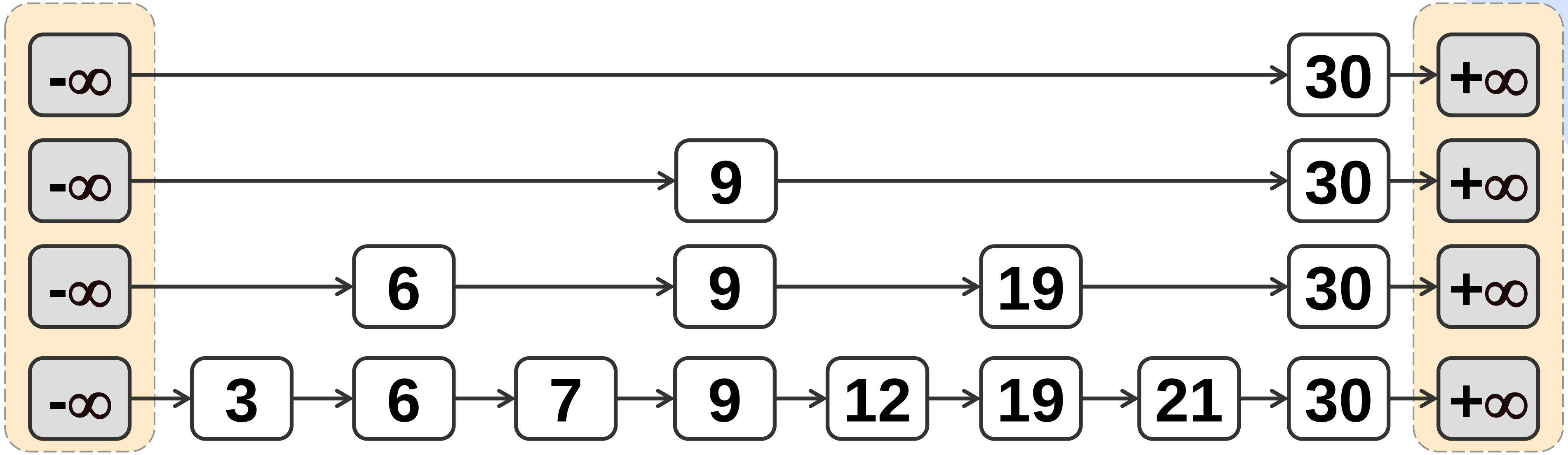



William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. Proceedings of Workshop on Algorithms and Data Structures, 437-449, 1989.

# 跳跃表—空间复杂度

C

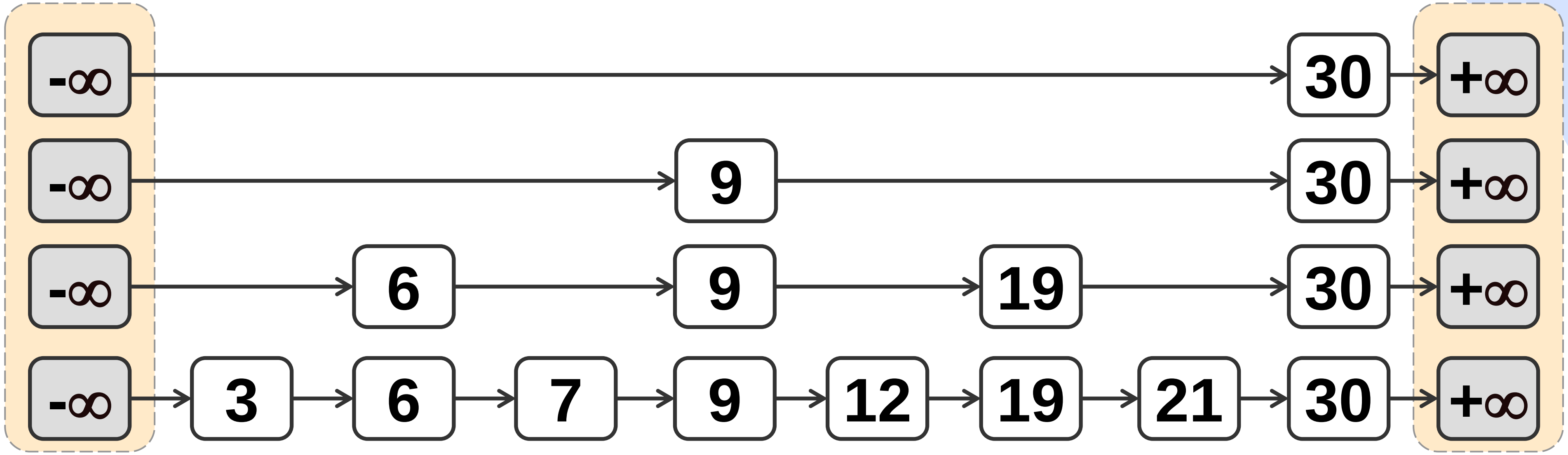
- 第0层有$n$个结点
- 第1层期望结点个数：$\lfloor n/2 \rfloor$
- 第2层期望结点个数：$\lfloor n/4 \rfloor$
- 第3层期望结点个数：$\lfloor n/8 \rfloor$
- 第$i$层期望结点个数：$\lfloor n/2^i \rfloor$



# 跳跃表 —空间复杂度

- 第$i$层期望结点个数：$\lfloor n/2^i \rfloor$
- 假设有$h$层，则

$$\sum_{i=0}^{h}\left\lfloor\frac{n}{2^i}\right\rfloor < \sum_{i=0}^{h}\frac{n}{2^i} = n\sum_{i=0}^{h}\frac{1}{2^i} = 2n\left(1-\frac{1}{2^{h+1}}\right) < 2n = O(n)$$



# 跳跃表 —查找

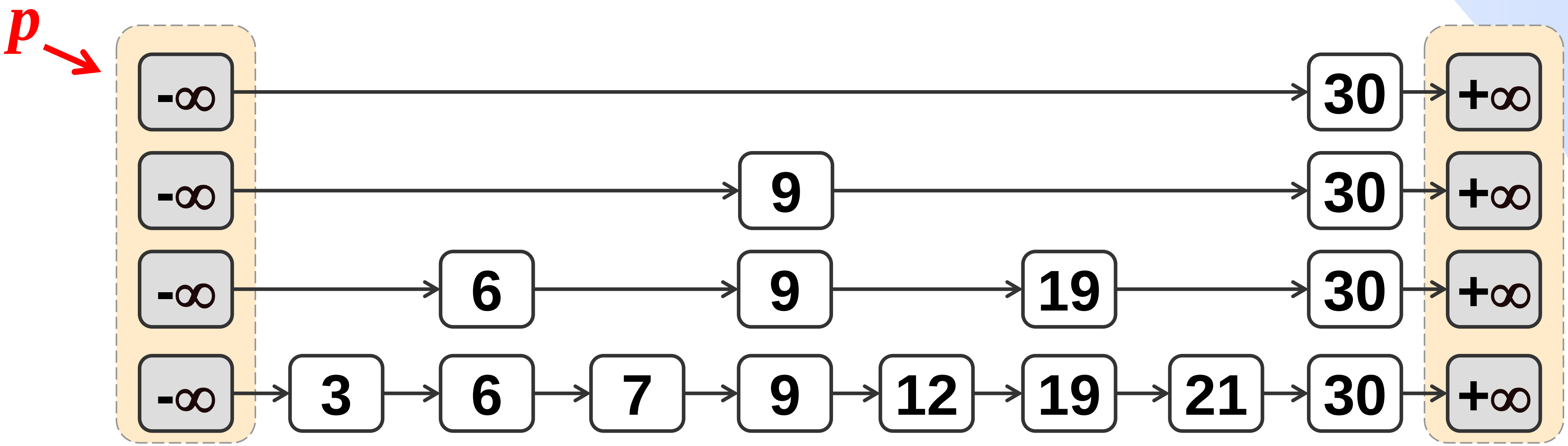
➢ 在跳表中查找元素*K*（例查找12）

令***p***指向最上层第一个结点，将*K*与p->next->data比较

① K < p->next->data: ***p***向下移动

② K > p->next->data: ***p***向右移动

③ K = p->next->data: 查找成功



# 跳跃表 —查找

C

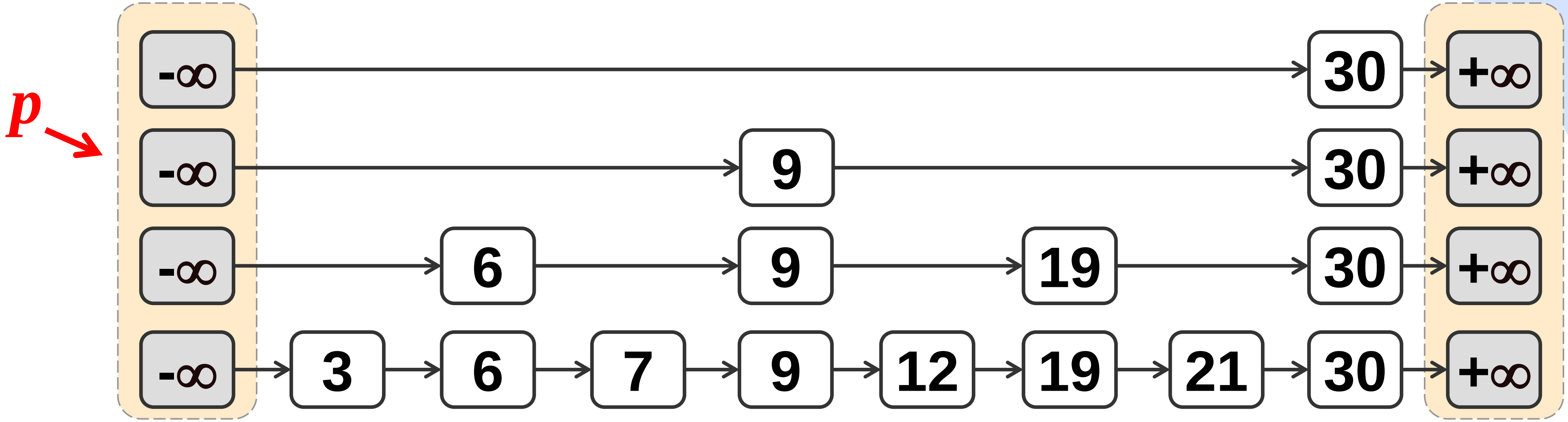
- 在跳表中查找元素*K*（例查找12）

令***p***指向最上层第一个结点，将*K*与p->next->data比较

① K < p->next->data: ***p***向下移动

② K > p->next->data: ***p***向右移动

③ K = p->next->data: 查找成功



# 跳跃表 —查找

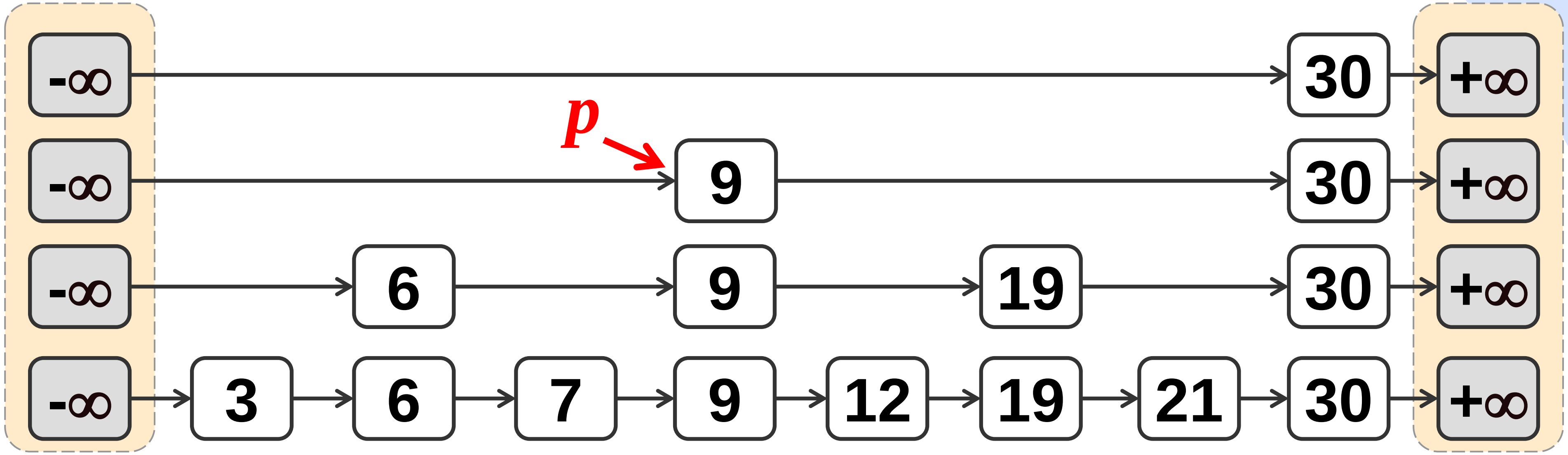
➢ 在跳表中查找元素*K*（例查找12）

令***p***指向最上层第一个结点，将*K*与p->next->data比较

① K < p->next->data: ***p***向下移动

② K > p->next->data: ***p***向右移动

③ K = p->next->data: 查找成功



# 跳跃表 —查找

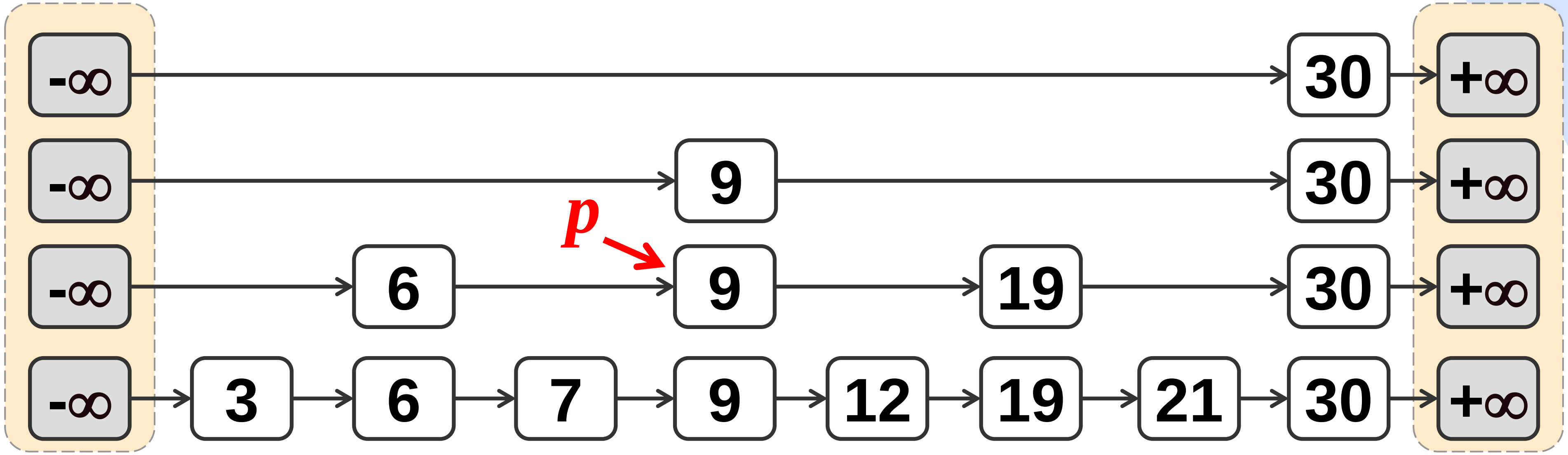
➢ 在跳表中查找元素*K*（例查找12）

令***p***指向最上层第一个结点，将*K*与p->next->data比较

① K < p->next->data: ***p***向下移动

② K > p->next->data: ***p***向右移动

③ K = p->next->data: 查找成功



# 跳跃表—查找

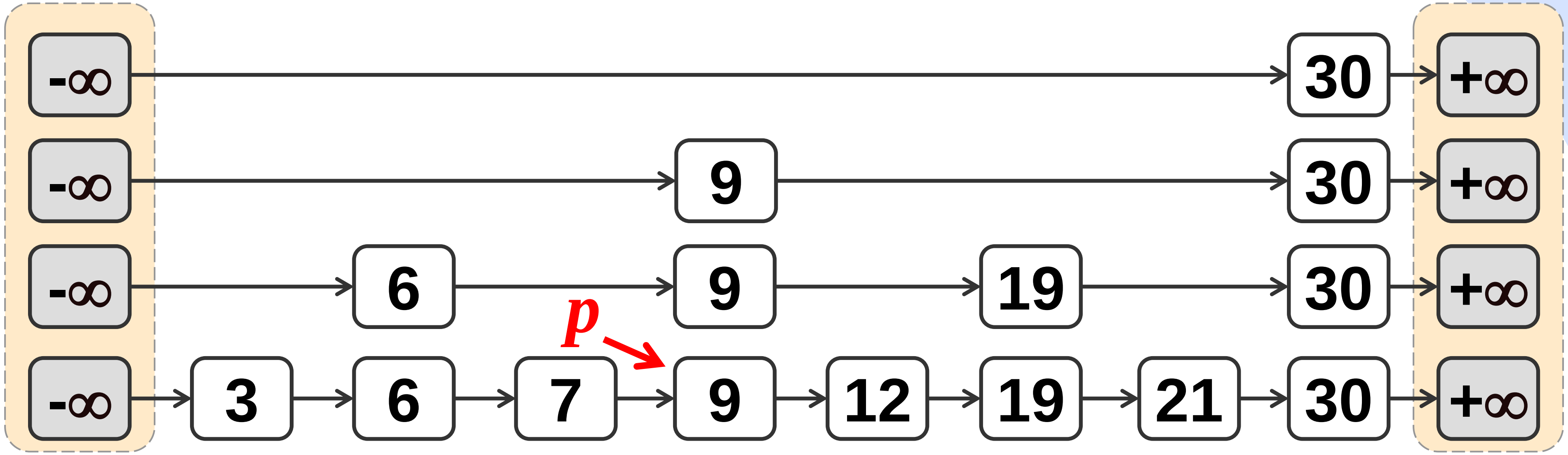
- 在跳表中查找元素*K*（例查找12）

令*p*指向最上层第一个结点，将*K*与p->next->data比较

① K < p->next->data: *p*向下移动

② K > p->next->data: *p*向右移动

③ K = p->next->data: 查找成功



# 跳跃表 —查找

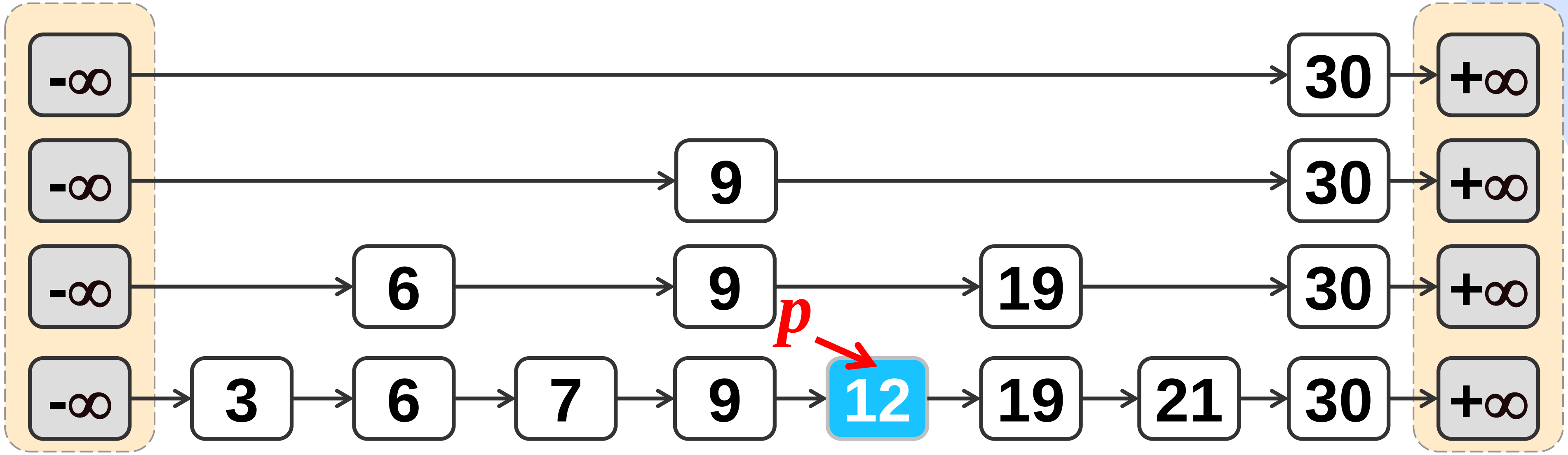
➢ 在跳表中查找元素*K*（例查找12）

令***p***指向最上层第一个结点，将*K*与p->next->data比较

① K < p->next->data：***p***向下移动

② K > p->next->data：***p***向右移动

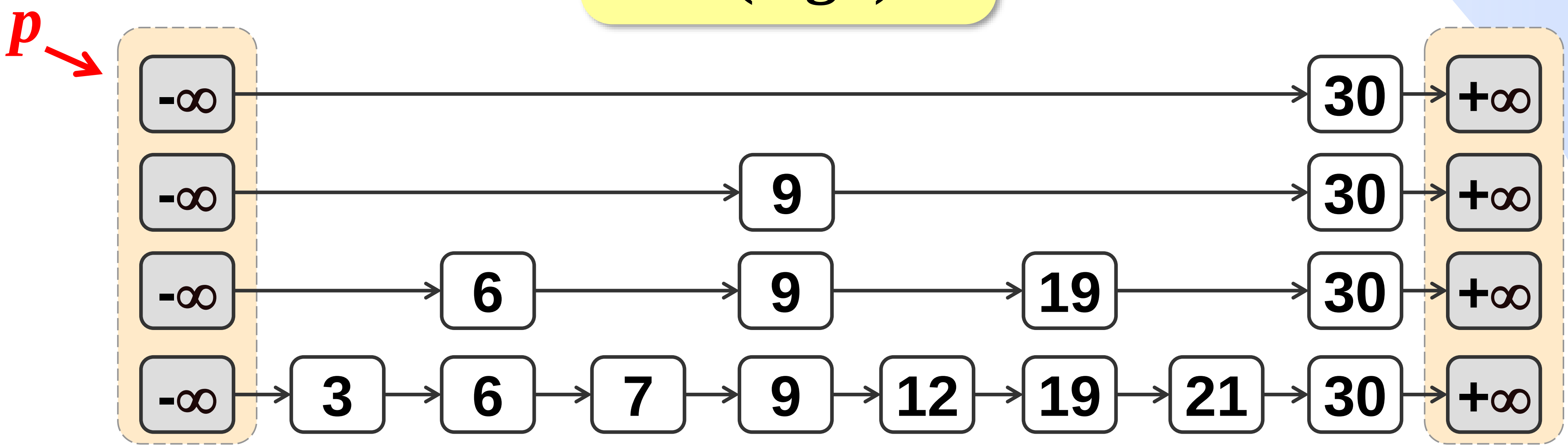
③ K = p->next->data：查找成功



# 跳跃表 — 时间复杂度

- 跳跃表的层数$h = O(\log n)$；
- 查找过程：由顶层到底层，每层元素比较次数为常数；
- 时间复杂度取决于跳跃表的层数。

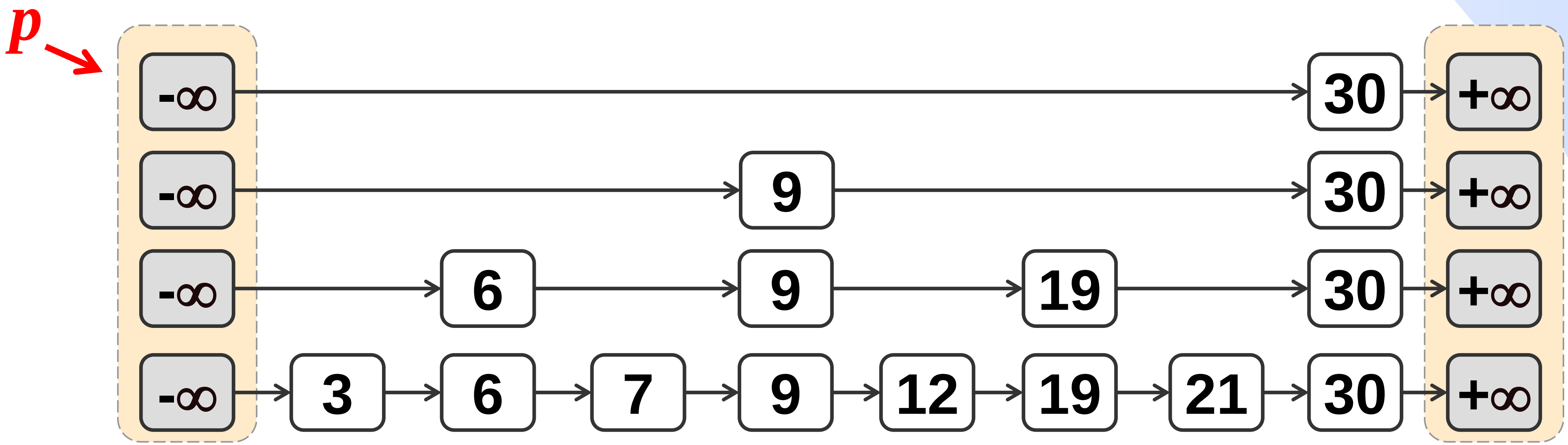
时间复杂度
O(log*n*)



# 跳跃表 — 插入

➢ 在跳表中插入元素*K*（例插入17）

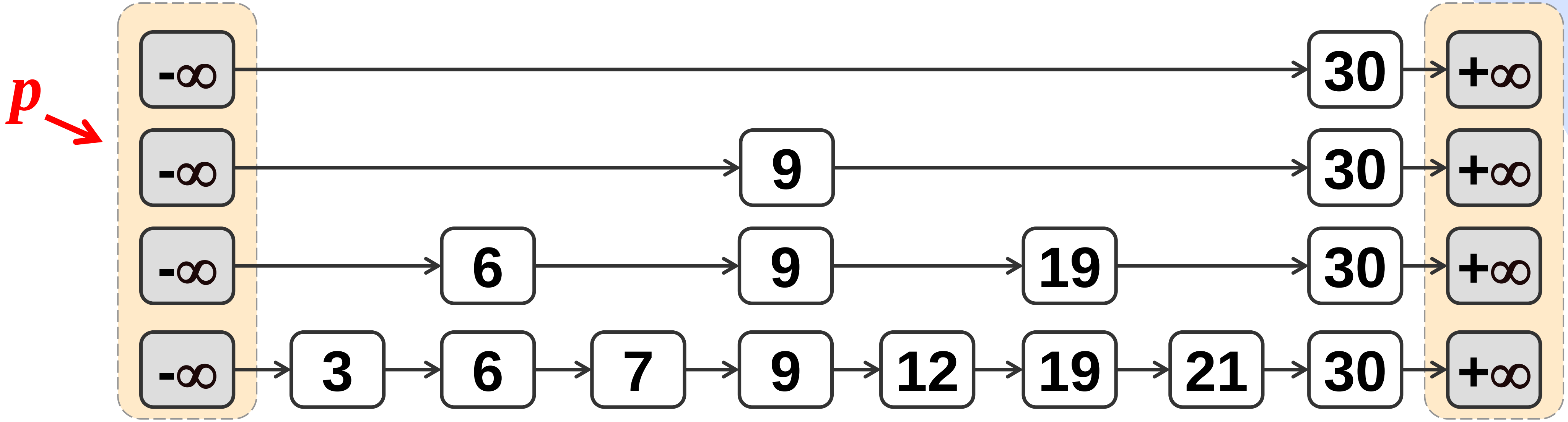
① 查找*K*，在查找失败的位置*p*插入*K*;



# 跳跃表 —— 插入

➢ 在跳表中插入元素$K$（例插入17）

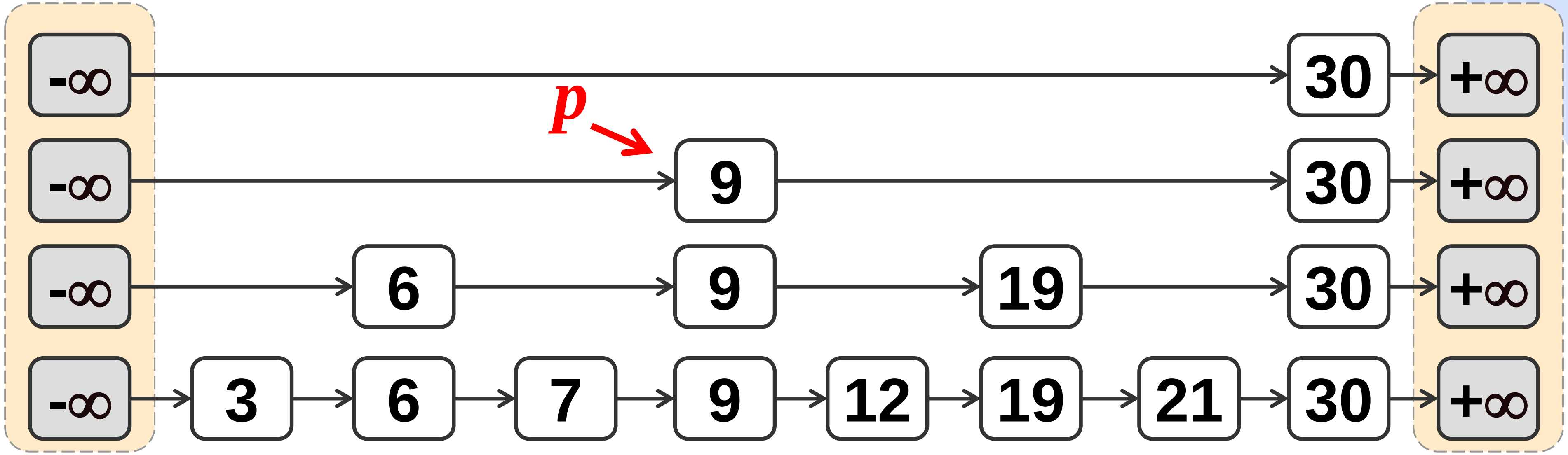
① 查找$K$，在查找失败的位置$p$插入$K$;



# 跳跃表 — 插入

- 在跳表中插入元素*K*（例插入17）

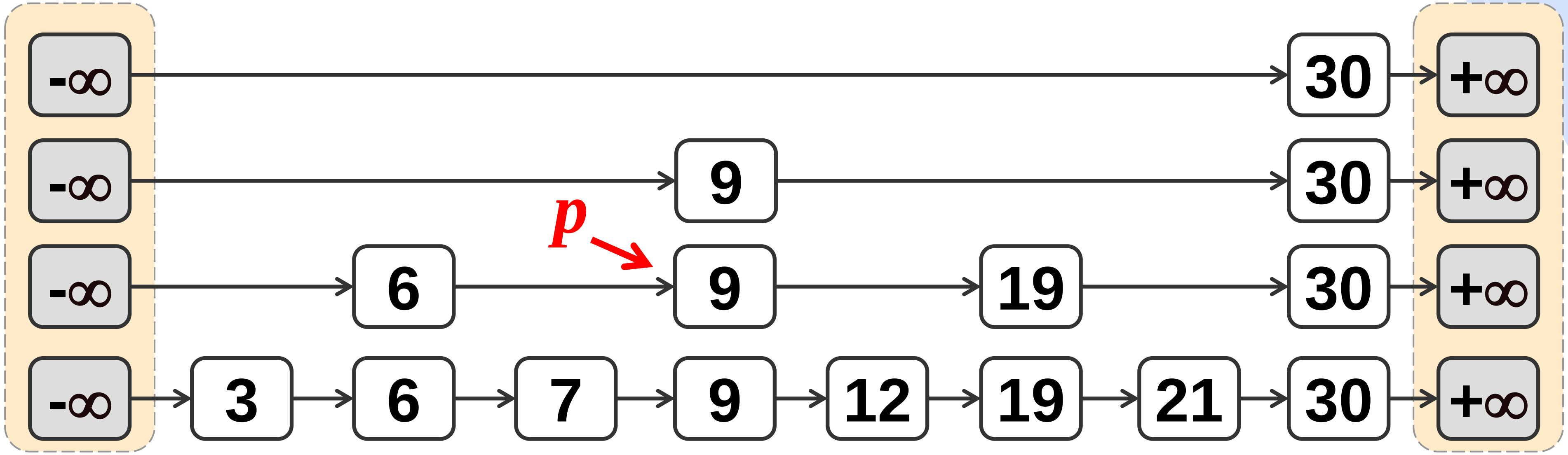
① 查找*K*，在查找失败的位置*p*插入*K*;



# 跳跃表 — 插入

➢ 在跳表中插入元素*K*（例插入17）

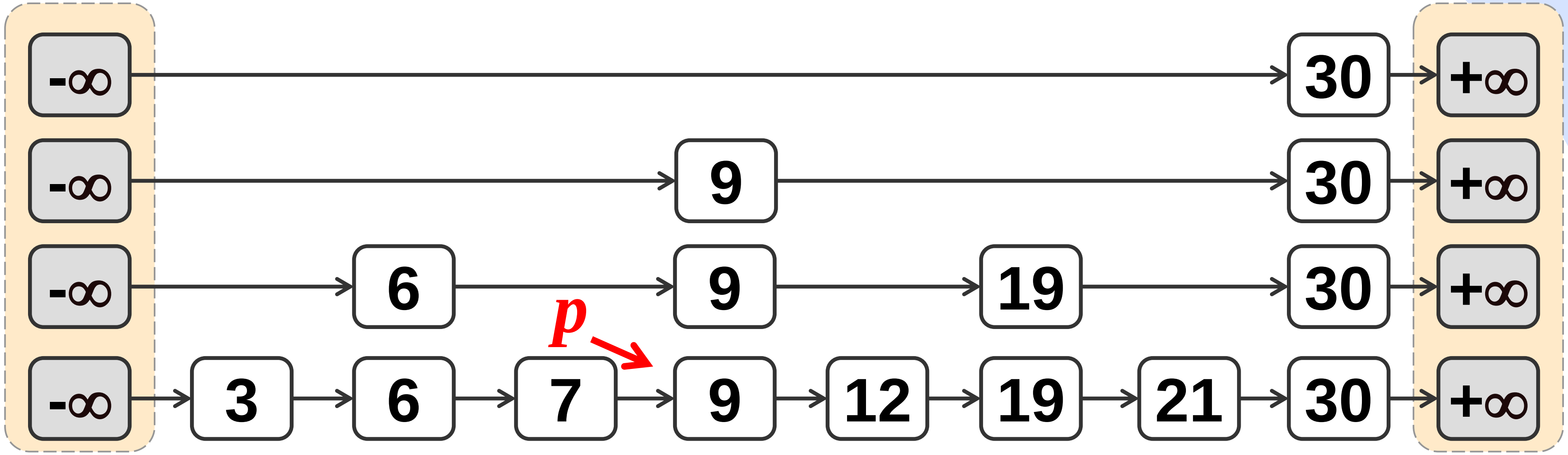
① 查找*K*，在查找失败的位置*p*插入*K*;



# 跳跃表 —— 插入

➢ 在跳表中插入元素*K*（例插入17）

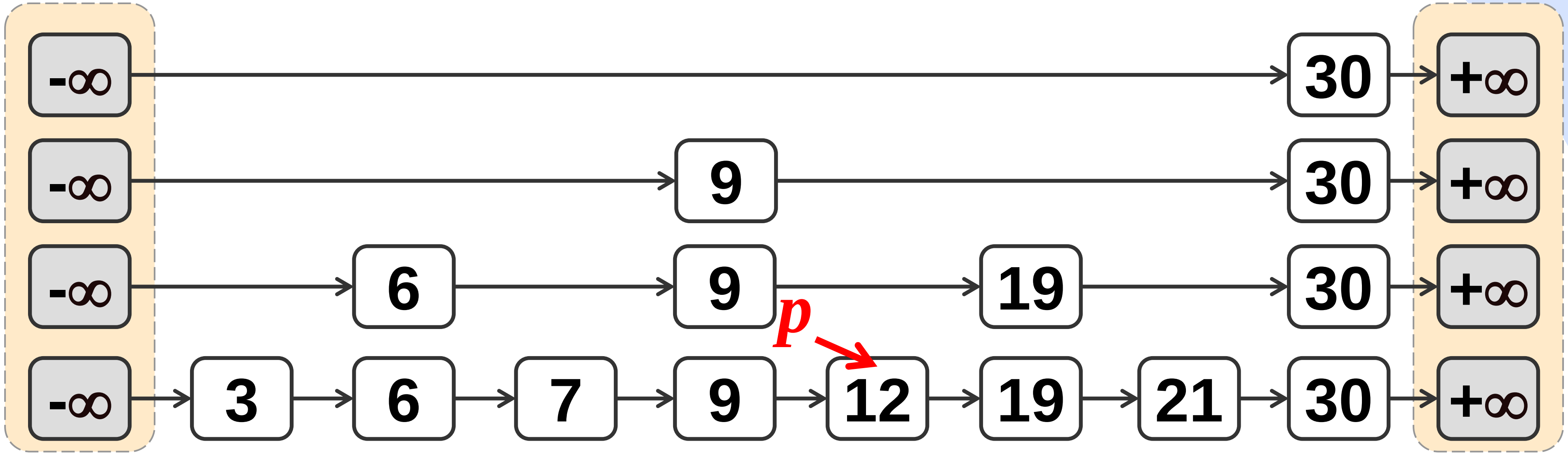
① 查找*K*，在查找失败的位置*p*插入*K*;



# 跳跃表 — 插入

➢ 在跳表中插入元素*K*（例插入17）

① 查找*K*，在查找失败的位置*p*插入*K*;



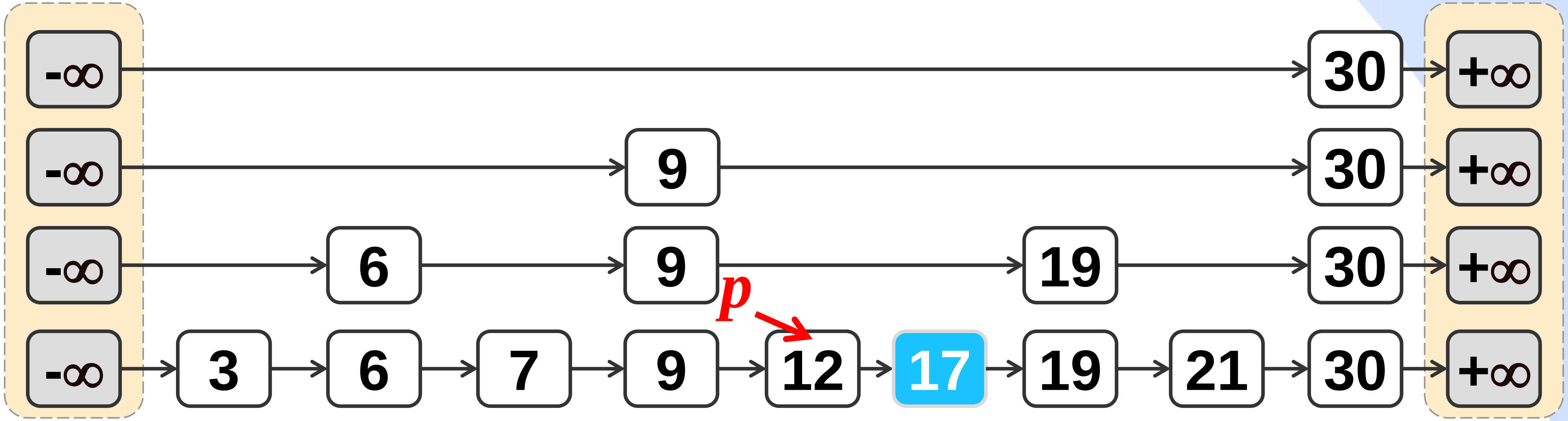
# 跳跃表 —— 插入

- 在跳表中插入元素*K*（例插入17）

① 查找*K*，在查找失败的位置*p*插入*K*；

② 以1/2的概率（可通过抛硬币实现）向上生长一层，生长概率逐层减半。

```
while(rand()%2==0)…
```



# 跳跃表 —— 插入

➢ 在跳表中插入元素*K*（例插入17）

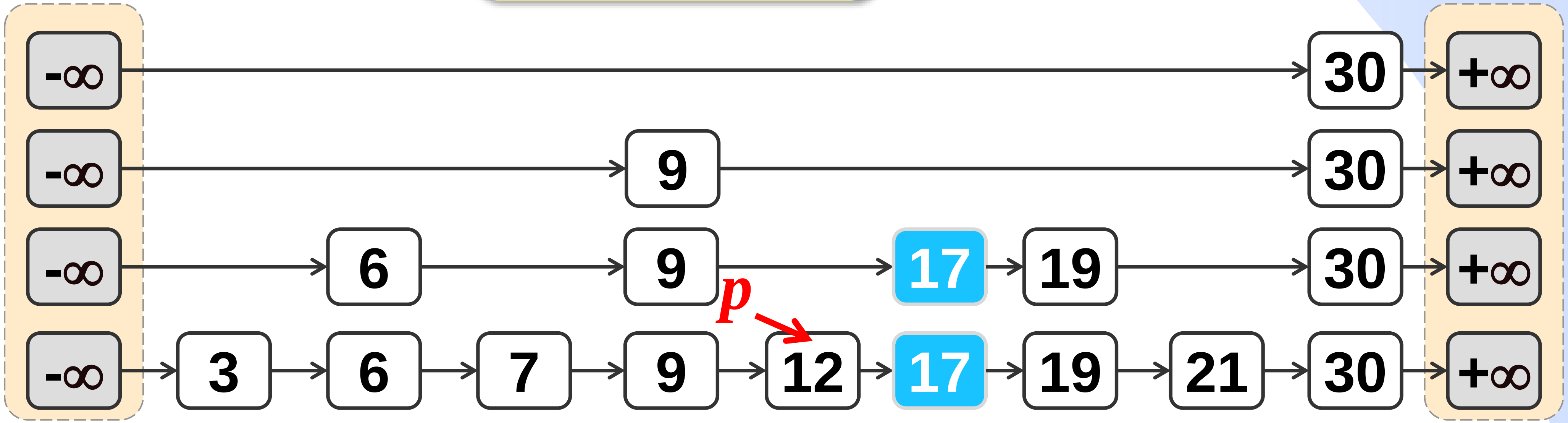
① 查找*K*，在查找失败的位置*p*插入*K*；

② 以1/2的概率（可通过抛硬币实现）向上生长一层，**生长概率逐层减半。**

```
while(rand()%2==0)…
```

时间复杂度

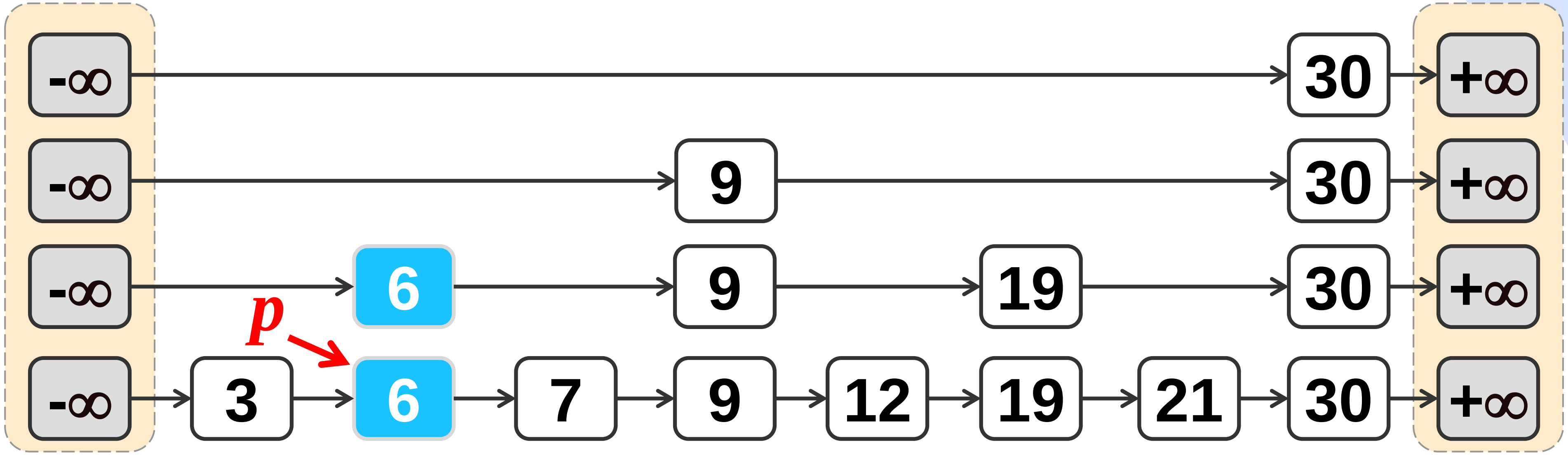
O(log*n*)



# 跳跃表 —— 删除

➢ 在跳表中删除元素*K*（例删除6）

查找*K*，删除每一层的*K*。



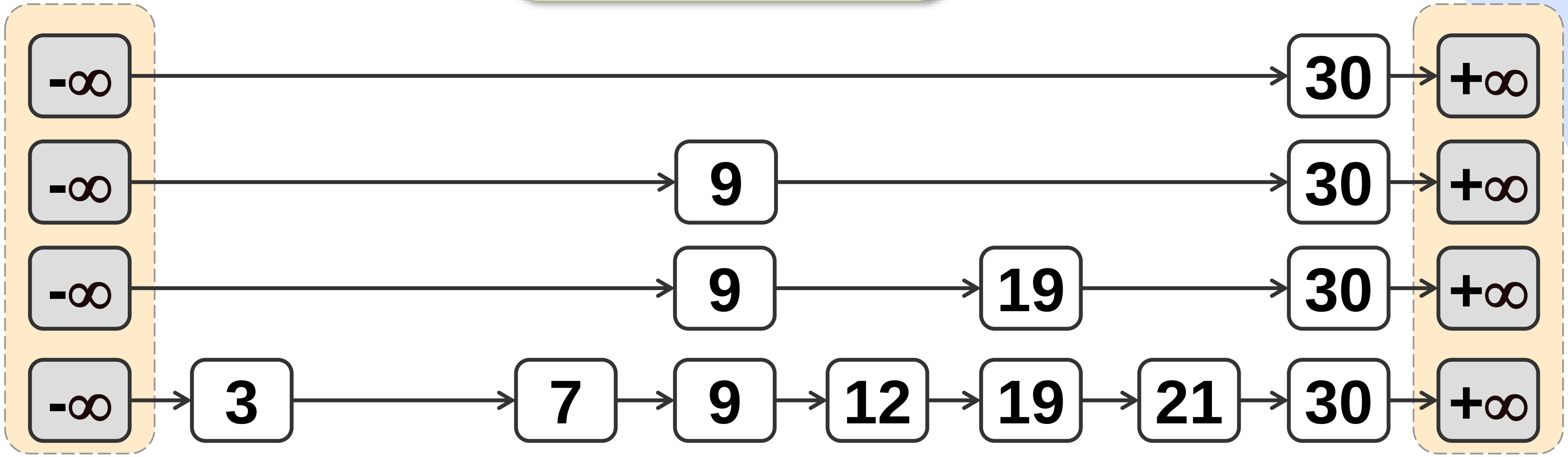
# 跳跃表 —— 删除

C

➢ 在跳表中删除元素*K*（例删除6）

查找*K*，删除每一层的*K*。

时间复杂度
O(log*n*)



# 跳跃表 — 总结

(1) 跳跃表的每一层都是一个有序的链表；

(2) 最底层的链表包含所有元素；

(3) 跳跃表是一种随机化的数据结构;

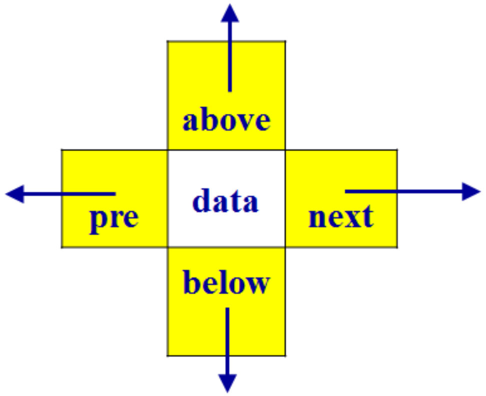
(4) 跳跃表的查找、插入、删除时间复杂度为O(log*n*);

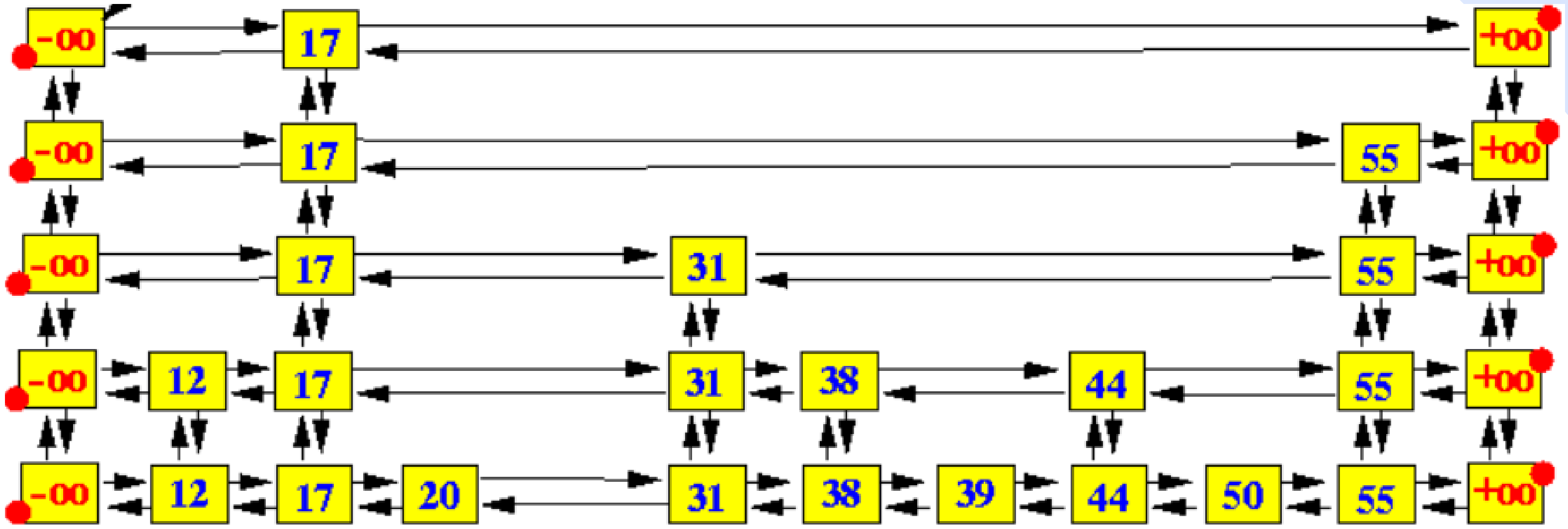
(5) 跳跃表的空间复杂度为 O(*n*);

(6) 适合动态查找场景。

# 跳跃表 —— 实现

➢ 实现方式一：四联表



# 跳跃表 —— 实现

C

➢ 实现方式二：指针数组

```
struct SkipNode{
      int data;
      SkipNode *next[];
};
```