查　　找

学习目标

理解查找的基本概念

掌握常用查找算法及其性能分析

掌握线性表查找（顺序、折半查找及其判定树）

掌握树表查找（二叉排序树、平衡二叉树）

掌握散列表的构造和查找

难点

二叉排序树的删除操作

平衡二叉树的调整

B树的插入和删除

查找技术

讲什么？

- 查找的基本概念
- 查找算法的时间性能

关键码

数据元素、结点、顶点

- 关键码：可以标识一个记录的某个数据项
- 键值：关键码的值

职工号	姓名	性别	年龄	工作时间
0001	王刚	男	48	1990.4
0002	张亮	男	35	2003.7
0003	刘楠	女	57	1979.9
0004	齐梅	女	35	2003.7
0005	李爽	女	56	1982.9

什么是查找

- 查找：在相同类型的记录构成的**集合**中找出**满足给定条件**的记录
  - 给定的查找条件可能是多种多样的
  - 把查找条件限制为“匹配”，即查找**关键码等于给定值**的记录
- 查找的结果：若在查找集合中找到了与给定值相匹配的记录，则称**查找成功**；否则，称**查找失败**

职工号	姓名	性别	年龄	工作时间
0001	王刚	男	48	1990.4
0002	张亮	男	35	2003.7
0003	刘楠	女	57	1979.9
0004	齐梅	女	35	2003.7
0005	李爽	女	56	1982.9

什么是查找

静态查找：不涉及插入和删除操作的查找

静态查找只注重查找效率，适用于：

（1）查找集合一经生成，便只对其进行查找，而不进行插入和删除操作

（2）经过一段时间的查找之后，集中地进行插入和删除等修改操作

动态查找：涉及插入和删除操作的查找

动态查找要求插入、删除、查找均有较好的效率，适用于：查找与插入和删除操作在同一个阶段进行

例如：当查找成功时，要删除查找到的记录

当查找不成功时，要插入被查找的记录

查找结构

查找结构：面向查找操作的数据结构，即查找基于的数据结构

已经有了数据结构的概念，为什么要强调查找结构?

（1）几乎所有的数据结构都提供了查找作为基本操作，但对于数据结构整体来说，查找并不是最重要的操作

（2）对于查找结构来说，查找是最重要的基本操作，重要的是查找效率

（3）数据结构 + 算法 = 程序：不同的查找结构，会获得不同的查找效率

查找结构 ⇨ 查找方法

查找结构

查找基于的数据模型是什么？ ⇨ 集合

集合
- 线性表：适用于静态查找，顺序查找、折半查找等技术
- 树　表：适用于动态查找，二叉排序树的查找技术
- 散列表：静态查找和动态查找均适用，散列技术

集合最常用的表示法是列举法，习惯上，也称为表

平均查找长度

- 如何评价查找算法的效率呢？ ⇨ 和关键码的比较次数
- 关键码的比较次数与哪些因素有关呢？
- 平均查找长度(Average Search Length)：查找算法进行的关键码比较次数的数学期望值

$$ASL = \sum_{i=1}^{n} p_i c_i$$

其中：n：问题规模，查找集合中的记录个数

p_i：查找第 i 个记录的概率

c_i：查找第 i 个记录所需的关键码的比较次数

p_i 与算法无关，取决于具体应用

c_i 取决于算法

如果 p_i 是已知的(一般取1/n)，则平均查找长度只是问题规模的函数

单选题 5分

设置

1. 查找结构是面向查找的数据结构，一般不涉及插入、删除等其他操作。

A 正确

B 错误

提交

单选题 5分

设置

2. 平均查找长度是查找算法进行的关键码比较次数。

A 正确

B 错误

提交

单选题 5分

设置

3. 评价查找算法的效率时，有时可以不考虑查找失败情况下的关键码比较次数。

A 正确

B 错误

提交

查找

- **基于线性表的查找：适用于静态查找**
  - **顺序查找**
  - **折半查找**
- 基于树 表的查找：适用于动态查找
- 基于散列表的查找：静态查找和动态查找均适用

线性表查找结构的类定义

```
const int MaxSize = 100;
class LineSearch
{
public:
    LineSearch(int a[ ], int n);
    ~LineSearch( ) { }
    int SeqSearch1(int k);
    int SeqSearch2(int k);
    int BinSearch1(int k);
    int BinSearch2(int low, int high, int k);
private:
    int data[MaxSize];
    int length;
};
```

不失一般性，做如下约定：

（1）待查关键码为 int 型

（2）记录只有一个数据项

（3）采用静态数组

（4）查找成功返回序号，失败返回 0

```
LineSearch :: LineSearch(int a[ ], int n)
{       //查找集合从下标1开始存放
    for (int i = 0; i < n; i++)
        data[i+1] = a[i];
    length = n;
}
```

注意，本章节线性表是以顺序表结构实现的

顺序查找

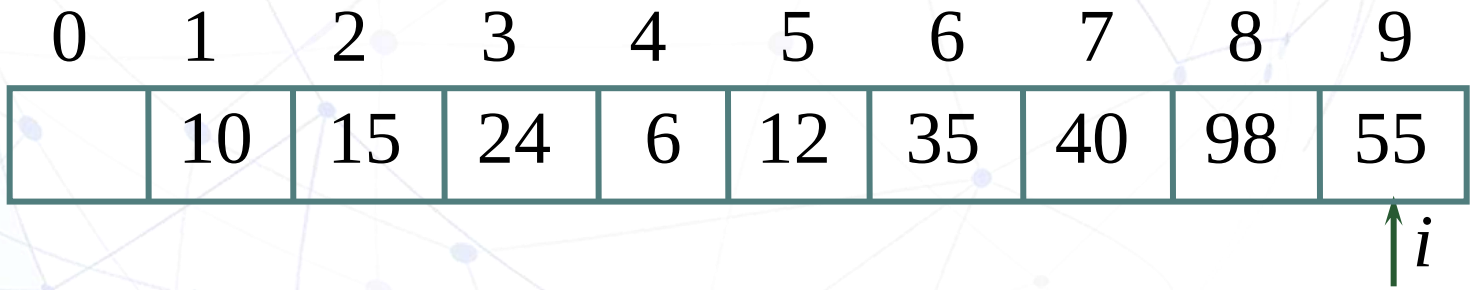
- 顺序查找的基本思想
- 改进的顺序查找算法
- 顺序查找的时间性能
- 顺序查找的优缺点

基本思想

顺序查找（线性查找）：从线性表的一端向另一端逐个将元素与给定值进行比较，若相等，则查找成功，给出该元素在表中的位置；若整个表检测完仍未找到与给定值相等的元素，则查找失败，给出失败信息

```
int LineSearch :: SeqSearch1(int k)
{
    int i = length;
    while (i > 0 && data[i] != k)
        i--;
    return i;
}
```

例：查找 35，查找 25



改进的顺序查找

- 顺序查找的改进：设置“哨兵”，就是待查值，放在查找方向的尽头处，免去了每一次比较后都要判断查找位置是否越界

```
int LineSearch :: SeqSearch2(int k)
{
    int i = length;  data[0] = k;
    while (data[i] != k)
      i--;
    return i;
}
```

```
int LineSearch :: SeqSearch1(int k)
{
    int i = length;
    while (i > 0 && data[i] != k)
      i--;
    return i;
}
```

例：查找 35，查找 25

性能分析

比较次数与key值所在位置有关

查找第i个记录需要比较n-i+1次，假设每个记录等概率查找，$P_i=1/n$

查找成功：

$$\sum_{i=1}^{n} p_i c_i = \sum_{i=1}^{n} p_i (n-i+1) = \frac{n+1}{2} = O(n)$$

查找不成功：

$$n+1 = O(n)$$

顺序查找优缺点

缺点：查找效率较低

特别是当待查找集合中元素较多时，不推荐使用顺序查找

优点：算法简单而且使用面广

（1）对表中记录的存储没有任何要求，顺序存储和链接存储均可

（2）对表中记录的有序性也没有要求，无论记录是否按关键码有序均可

讨论

1.记录的查找概率不相等时如何提高查找效率?

按照查找概率高低存储记录

(1) 查找概率越高，让其比较次数越少；
(2) 查找概率越低，让其比较次数越多；

讨论

2.记录的查找概率无法测定时如何提高查找效率?

按照查找概率动态调整记录顺序

(1) 在每个记录中设一个访问频率域;
(2) 始终保持记录按访问频率有序排列;

单选题 5分

设置

1. 设置哨兵可以在数量级上提高顺序查找的时间性能。

A 正确

B 错误

提交

单选题 5分

设置

2. 顺序查找的效率较低，一般不采用顺序查找技术。

A 正确

B 错误

提交

单选题 5分

设置

3. 在数组中进行顺序查找，平均要比较大约一半元素，因此平均时间性能是O(n/2)。

A 正确

B 错误

提交

单选题 5分

设置

4. 在数组中进行顺序查找，查找不成功的时间复杂度是O(n)。

A 正确

B 错误

提交

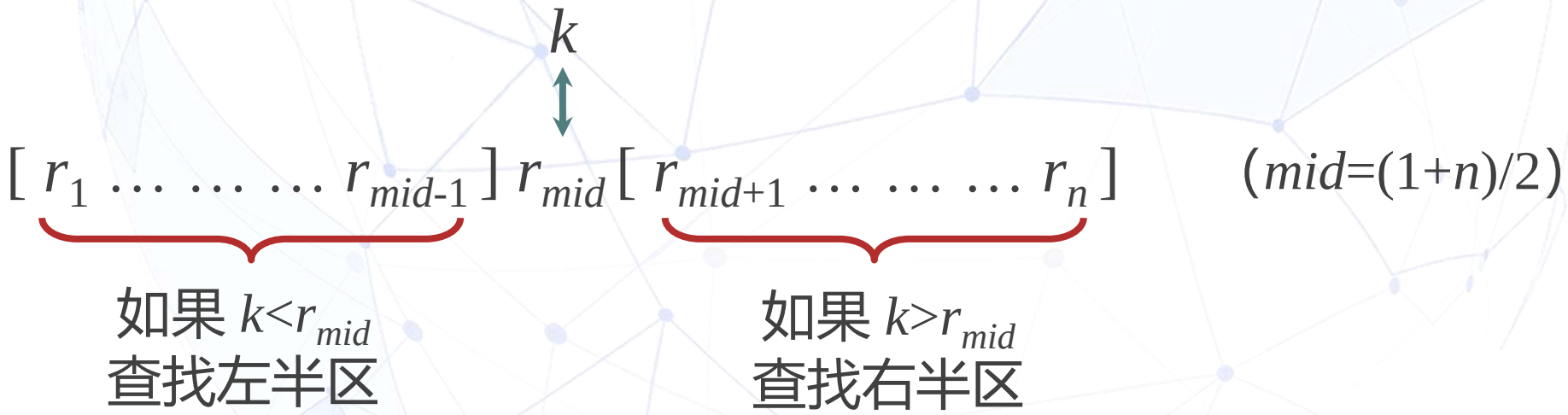
折半查找（二分查找）

讲什么？

- 折半查找的基本思想
- 折半查找的运行实例
- 折半查找的非递归算法
- 折半查找的递归算法
- 判定树
- 折半查找的性能分析

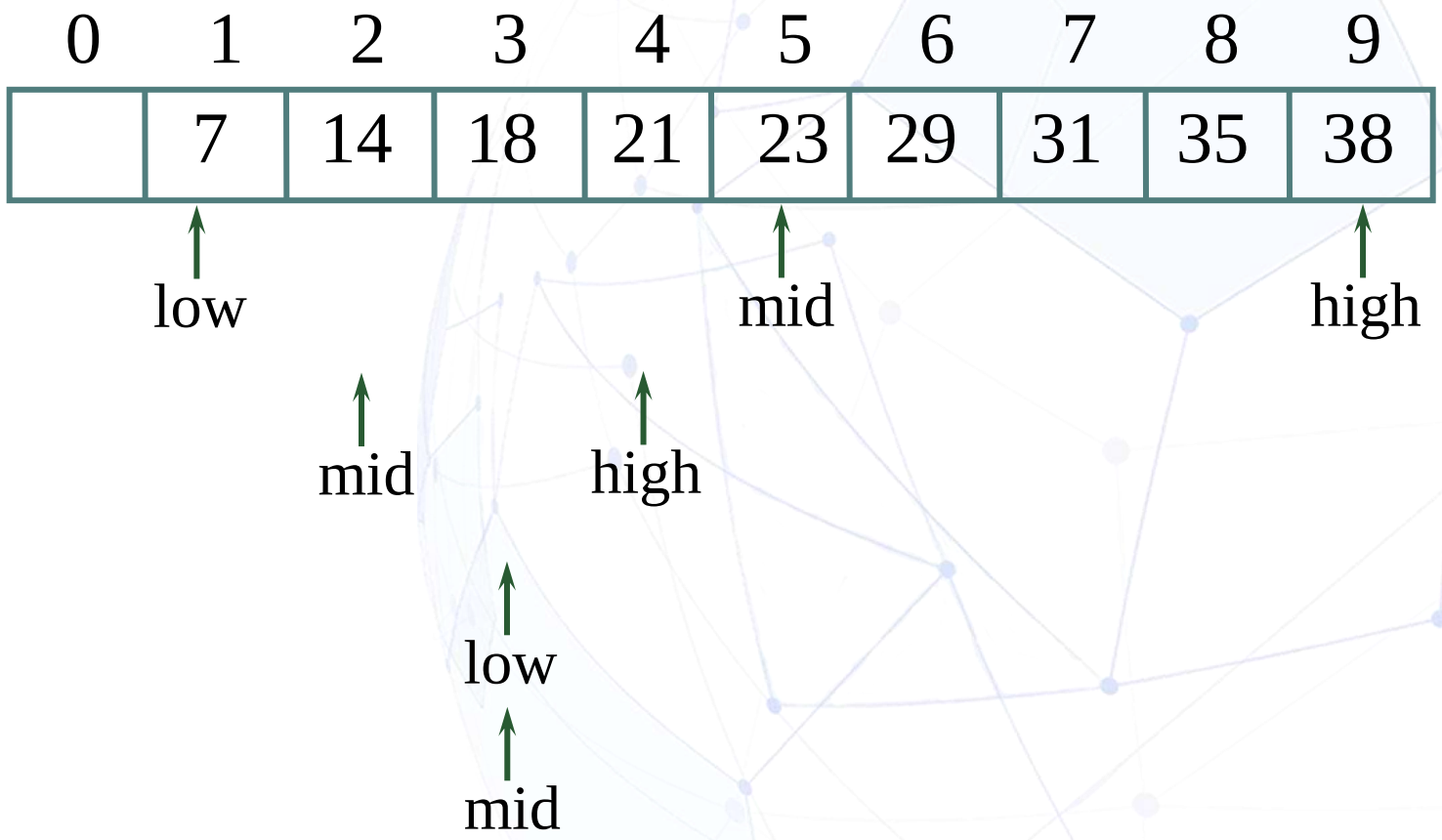
基本思想

折半查找（对半查找、二分查找）：在有序表（假设为递增）中，取中间元素作为比较对象，若给定值与中间元素相等，则查找成功；若给定值小于中间元素，则在有序表的左半区继续查找；若给定值大于中间元素，则在有序表的右半区继续查找。不断重复上述过程，直到查找成功，或查找区域无元素，查找失败



运行实例

例：查找 18



查找区间 [1, 9]

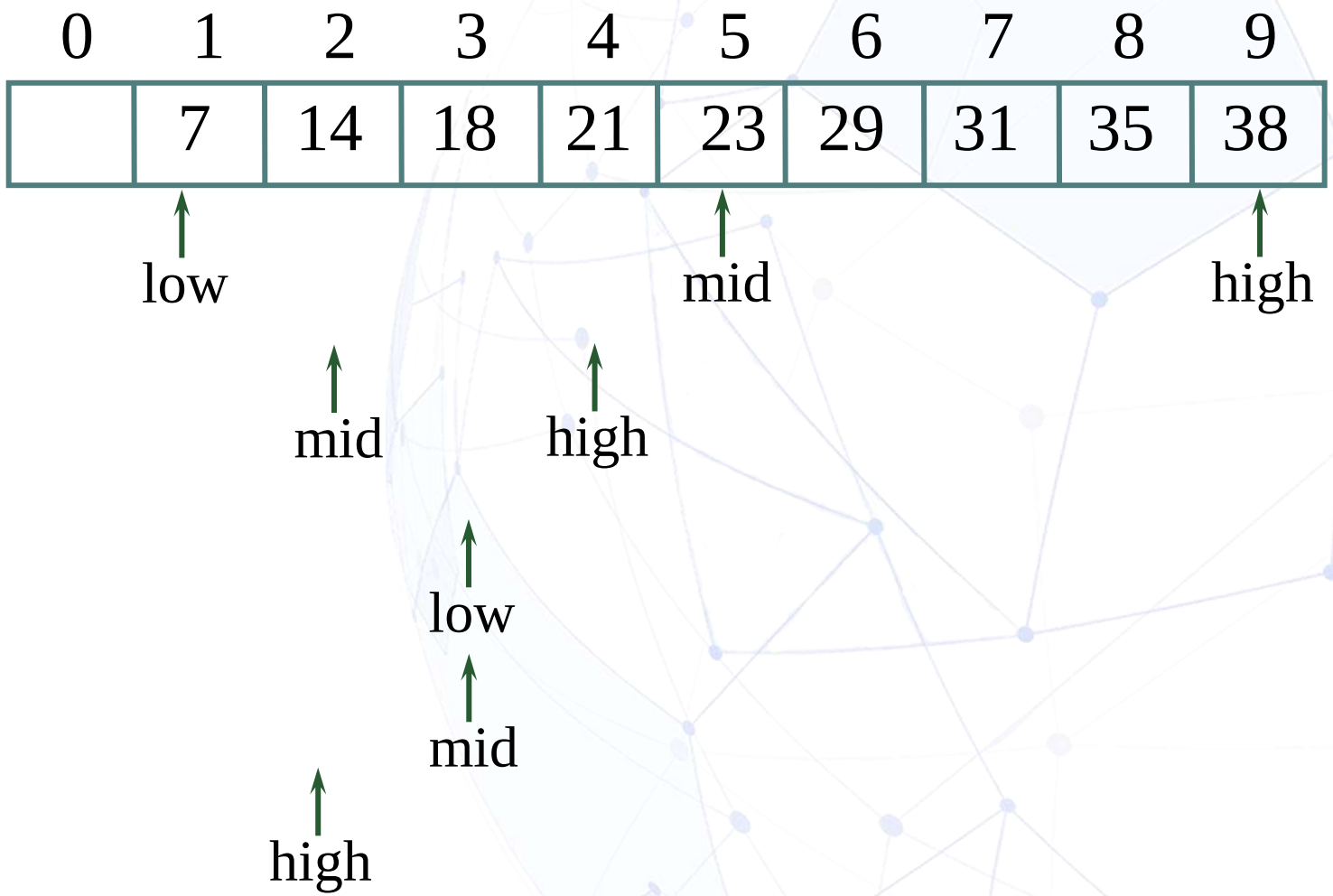
查找区间 [1, 4]

查找区间 [3, 4]

查找成功

运行实例

例：查找 15



查找区间 [1, 9]

查找区间 [1, 4]

查找区间 [3, 4]

查找区间 [3, 2]

low > high，查找失败

算法描述

1、初始化：low = 1; high = length;

2、循环地执行：

mid = (low+high)/2;

获得mid位置的记录，假设为data，将目标target与data做是否相等比较

- 如果相等，则查找成功，mid即为查找到的位置；
- 如果target>data，则继续到右半区间搜索，即low=mid+1
- 否则继续到左半区间搜索，即high=mid-1，从而将搜索范围每次缩小约一半。

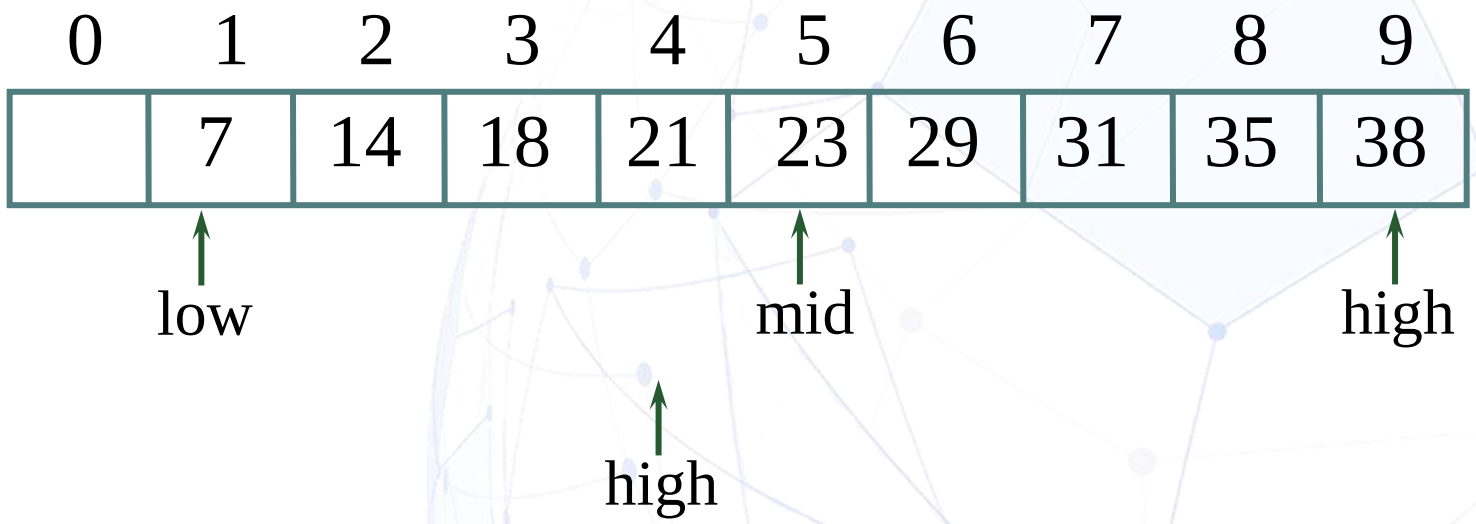
直至被搜索区间不存在，即low>high,表示查找失败。

非递归算法

```
int LineSearch :: BinSearch1(int k)                /*查找集合存储在r[1]~r[n]*/
{
   int mid, low = 1, high = n;                     /*初始查找区间是[1, n]*/
   while (low <= high)                             /*当区间存在时*/
   {
      mid = (low + high) / 2;
      if (k < data[mid])  high = mid - 1;
      else  if (k > data[mid])  low = mid + 1;
      else return mid;                             /*查找成功，返回元素序号*/
   }
   return 0;                                       /*查找失败，返回0*/
}
```

递归算法

例：查找 18



查找区间 [1, 9]

查找区间 [1, 4]

```
int LineSearch :: BinSearch2(int low, int high, int k)
{
    if (k < data[mid]) return BinSearch2(low, mid−1, k);
    else if (k > data[mid]) return BinSearch2(mid+1, high, k);
    else return mid;                                  /*查找成功，返回序号*/
}
```

递归算法

```
int LineSearch :: BinSearch2(int low, int high, int k)
{
    int mid;
    if (low > high) return 0;                          /*递归的边界条件*/
    else {
        mid = (low + high) / 2;
        if (k < data[mid]) return BinSearch2(low, mid-1, k);
        else if (k > data[mid]) return BinSearch2(mid+1, high, k);
        else return mid;                                /*查找成功，返回序号*/
    }
}
```

思考

上述二分查找算法能否在有序的链表下进行？

根据算法，需要不断地获得中间位置的元素，在链表下这个操作效率是线性阶O(n)的，也即一次获取中间位置元素的操作就跟顺序查找本身的效率同数量级了。因此链表下做二分查找效率是非常低下的，不值得尝试。而在顺序表下是根据位序直接读写的。

因此高效率的二分查找算法，必然是在有序顺序表下进行的。这也是二分查找的前提。

算法分析

每次比较将下一步的搜索区间范围缩小约一半

每次比较后剩余记录大约个数：

比较次数	剩余记录大约个数
1	n/2
2	n/4
3	n/8
…	
i	$n/2^i$

当比较足够多次后，搜索区间范围内仅剩余1个元素；

不管这个数据项是否匹配查找项，比较最终都会结束，解下列方程：$\frac{n}{2^i} = 1$

得到：$i=\log_2(n)$

算法时间复杂度：$O(\log_2 n)$

折半查找性能分析-判定树

判定树（折半查找判定树）：描述折半查找判定过程的二叉树

设查找区间是[low, high]，判定树的构造方法：

（1）当low > high时，判定树为空；

（2）当low ≤ high时，判定树的根结点是有序表中序号为mid=(low+high)/2的记录，根结点的左子树是与有序表r[low]~r[mid-1]相对应的判定树，根结点的右子树是与有序表r[mid+1] ~ r[high]相对应的判定树。

判定树

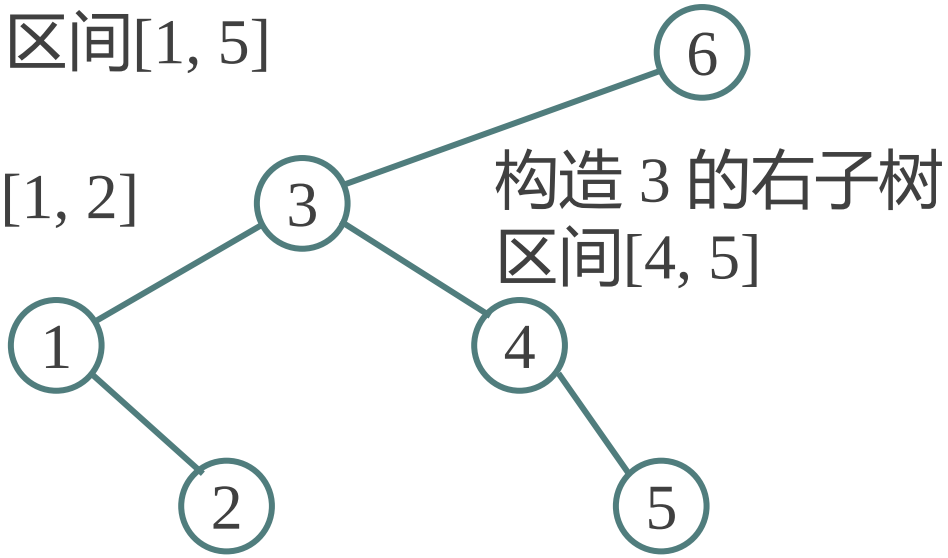
例如，结点个数（查找集合的记录个数）为11的判定树

i	1	2	3	4	5	6	7	8	9	10	11
	5	13	19	21	37	56	64	75	80	88	92
C_i	3	4	2	3	4	1	3	4	2	3	4

查找区间[1, 11]，中间记录的序号是6

构造 6 的左子树，区间[1, 5]

构造 3 的左子树，区间[1, 2]

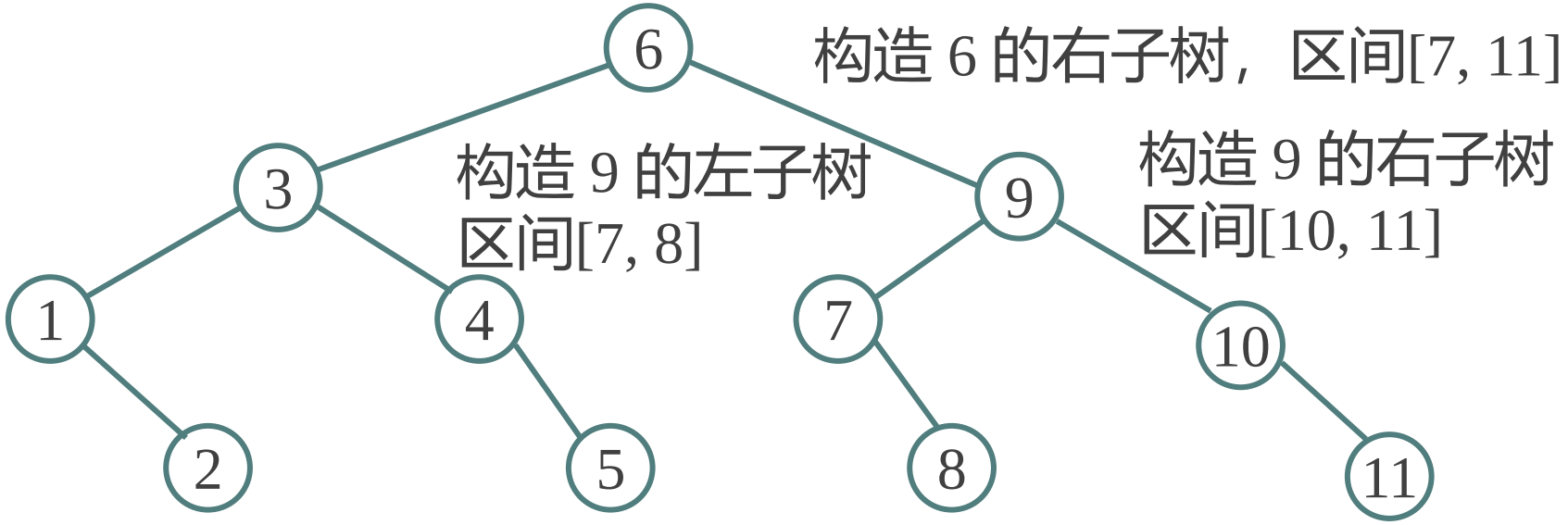


构造 3 的右子树
区间[4, 5]

判定树

例如，结点个数（查找集合的记录个数）为11的判定树

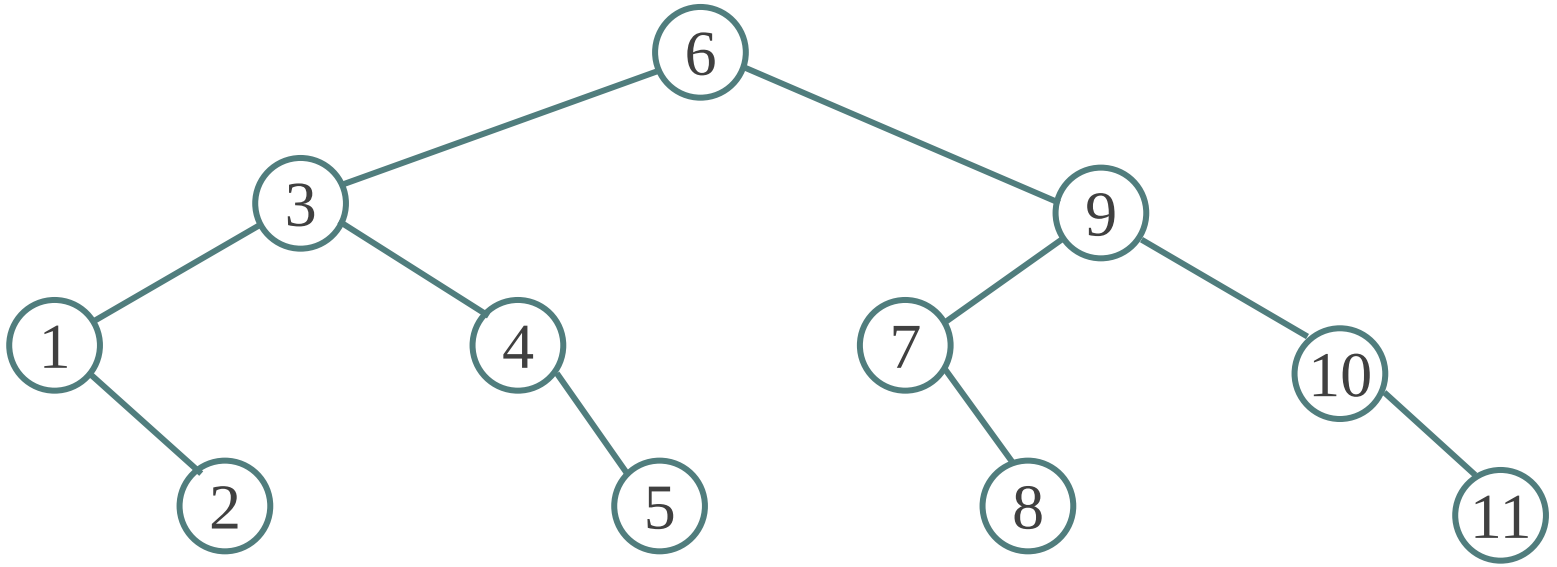
i	1	2	3	4	5	6	7	8	9	10	11
	5	13	19	21	37	56	64	75	80	88	92
C_i	3	4	2	3	4	1	3	4	2	3	4



折半查找性能分析

例如，结点个数（查找集合的记录个数）为11的判定树

查找第 4 个元素比较多少次?



折半查找任一记录的过程，即是判定树中从根结点到该记录结点的路径

比较次数 = 路径上的结点数

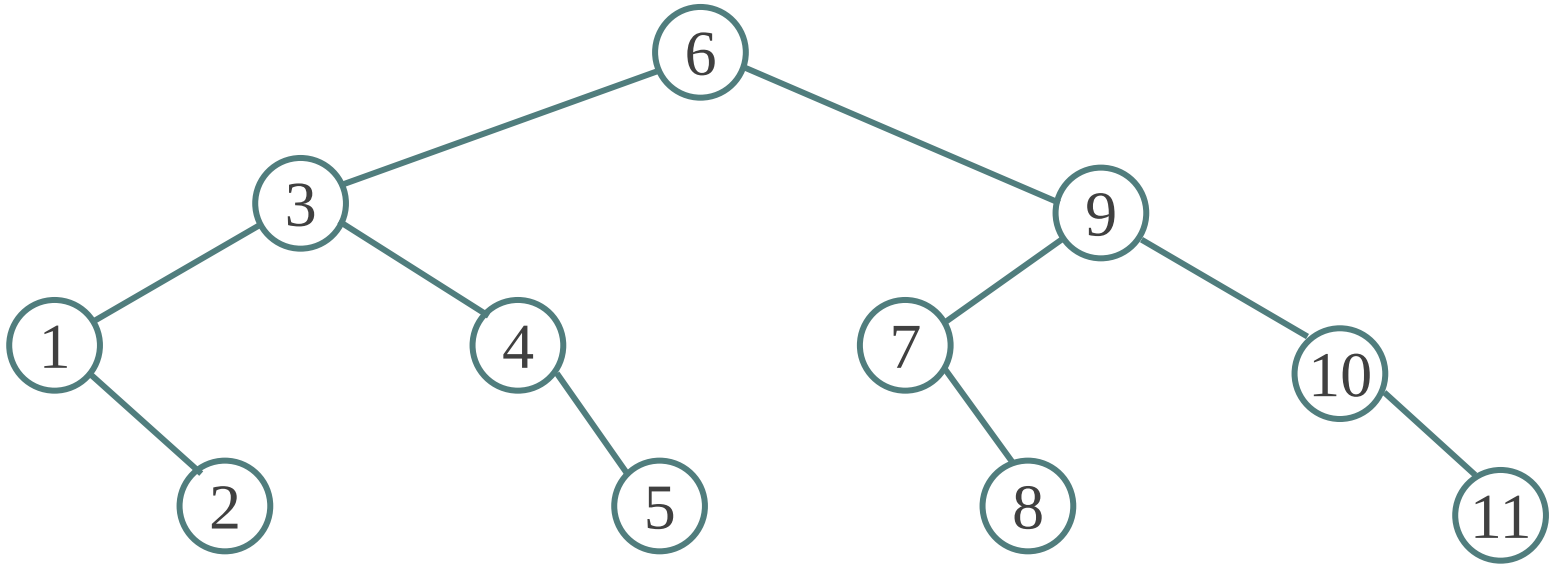
比较次数 = 结点 的层数

比较次数
IA
树的深度

折半查找性能分析

例如，结点个数（查找集合的记录个数）为11的判定树

查找成功情况下，与判定树的深度有关，判定树的深度是多少呢?



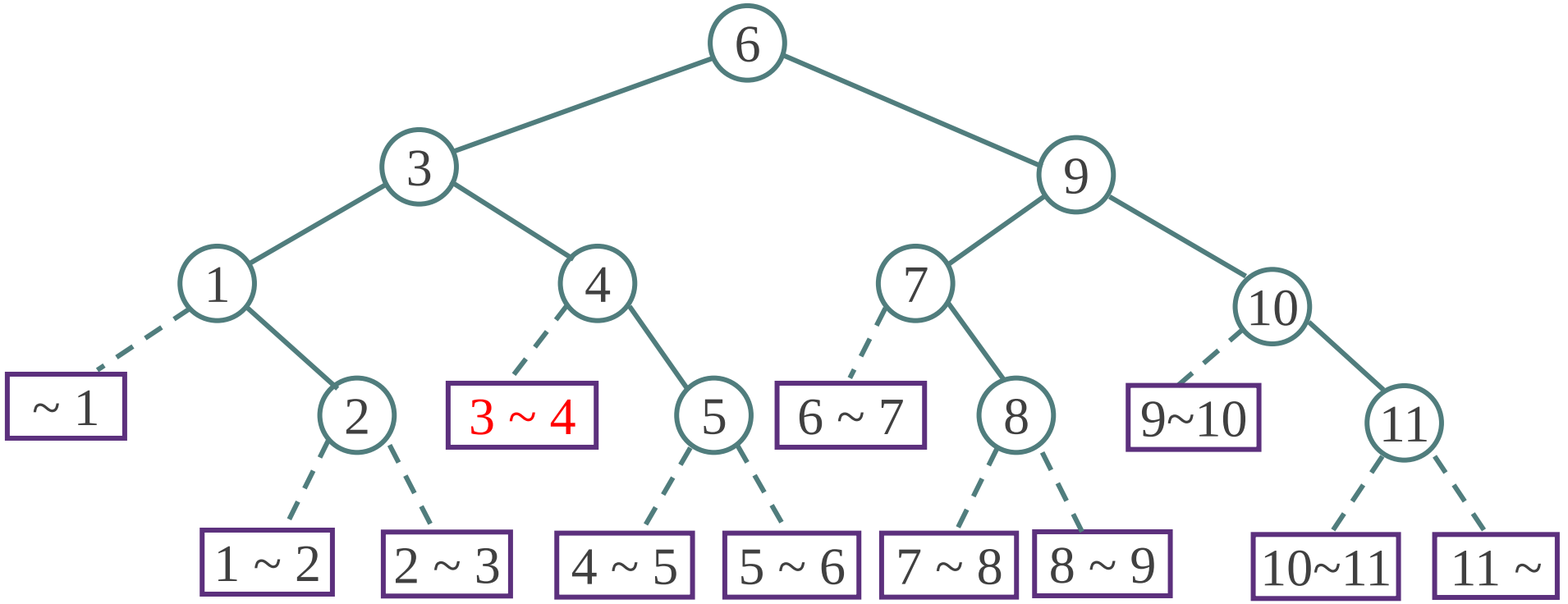
判定树深度为 $\lfloor \log_2 n \rfloor + 1$ ⇨ 比较次数至多为 $\lfloor \log_2 n \rfloor + 1$ ⇨ 时间复杂度为$O(\log_2 n)$

折半查找性能分析

例如，结点个数（查找集合的记录个数）为11的判定树

如何确定查找失败呢？ 例如查找的元素比第3个元素大比第4个元素小？

查找不成功的过程是从根结点到外部结点的路径，和给定值进行的比较次数等于该路径上内部结点的个数。



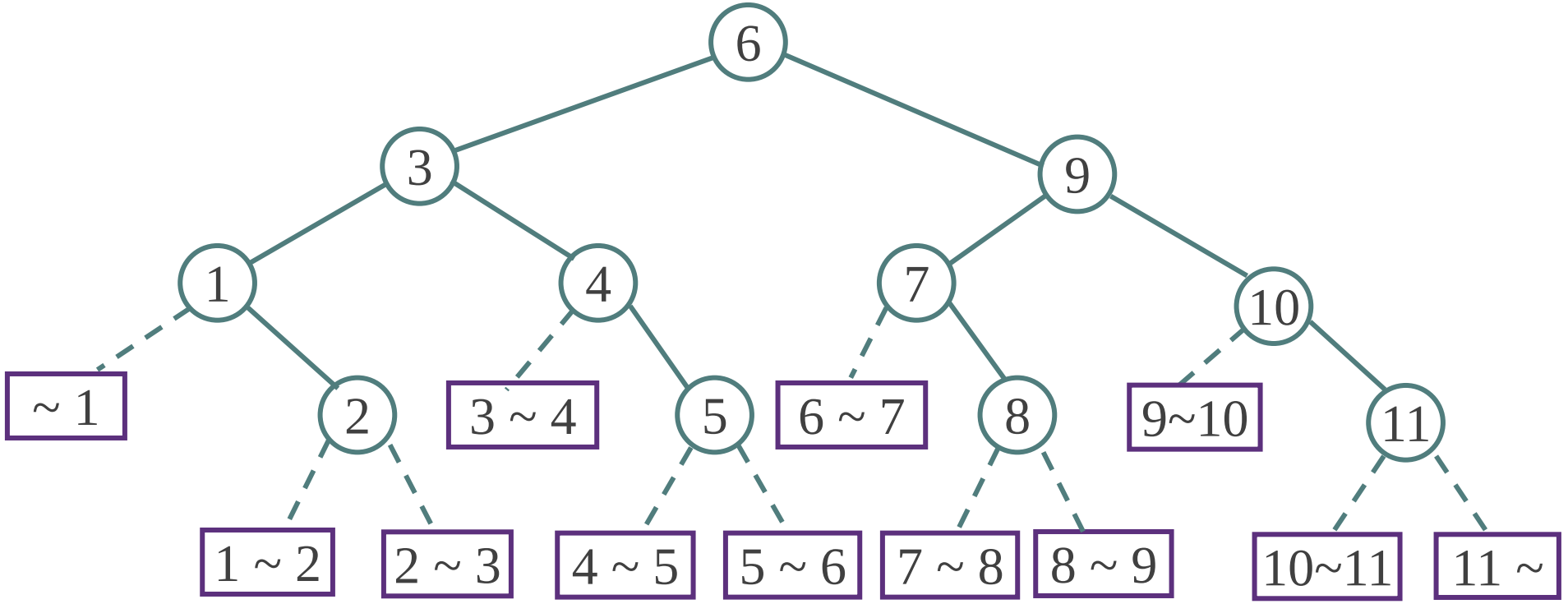
圆形——内结点
代表查找成功

矩形——外结点
代表查找不成功

外结点数量为n+1

折半查找性能分析

- 查找成功的平均比较次数 = (1×1+2×2+3×4+4×4)/11 = 3
- 查找不成功的平均比较次数 = (3×4+4×8)/12 = 11/3



补充

不识别相等非递归折半查找

算法描述

1、初始化 ： low =0; high =length-1。

2、当满足条件low<high时，循环执行：

mid = (low+high) / 2;

获得mid位置记录的关键字，假设为current；

将目标k与current相比较；

如果k>target，low=mid+1，否则high=mid，

3、如果low>high，说明整个待查表初始即为空表，查找失败，返回0；

4、如果low==high，则检查该区间唯一的元素是否是目标k，如果是则查找成功，否则查找失败。

不识别相等非递归折半查找

```
int LineSearch :: BinSearch1(int k) {
    int mid, low = 1, high = n;  //0处可放哨兵
    while (low <= high)     //左右闭合
    {
     mid = (low + high) / 2;
     if (k < data[mid])  high = mid - 1;
     else  if (k > data[mid])  low = mid + 1;
     else return mid; //查找成功
    }
    return 0; /*查找失败*/
}
```

识别相等

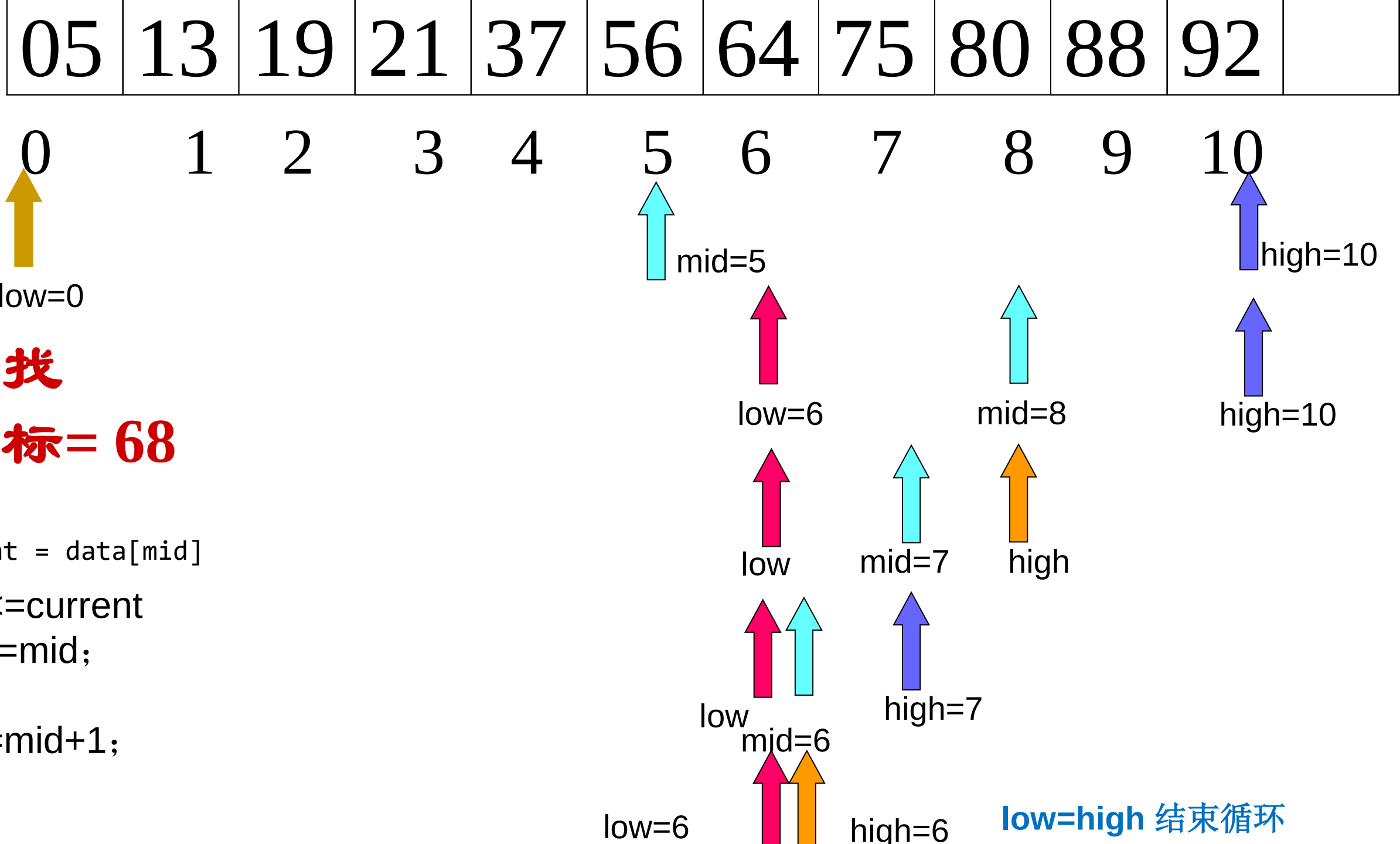
```
int LineSearch :: BinSearch2(int k) {
    int mid, low = 1, high = n; //0处可放哨兵

    while (low < high) {
        mid = (low + high) / 2;
        if (k <= this->data[mid])  //左闭右开
            high = mid;
        else
            low = mid + 1;
    }
    if (low > high)
        return 0; /*查找失败*/
    else  //即low==high
        if (this->data[low]==k)
            return low; //查找成功
        else
            return 0; /*查找失败*/
}
```

不识别相等递归折半查找

```
template <class DataType>
int OrderedList<DataType>::BinSearch11(int k) {
        return recBinSearch1(k, 1, this->length - 1);
}

template <class DataType>
int OrderedList<DataType>::recBinSearch1(int k, int low, int high) {
        if (low >high)
                return 0;
        if (low == high)
                if (this->data[low] == k)
                        return low;
                else
                        return 0;
        int mid = (low + high) / 2;
        if (k <= this->data[mid])
                return recBinSearch1(k,low,mid);
        else
                return recBinSearch1(k, mid+1, high);
}
```

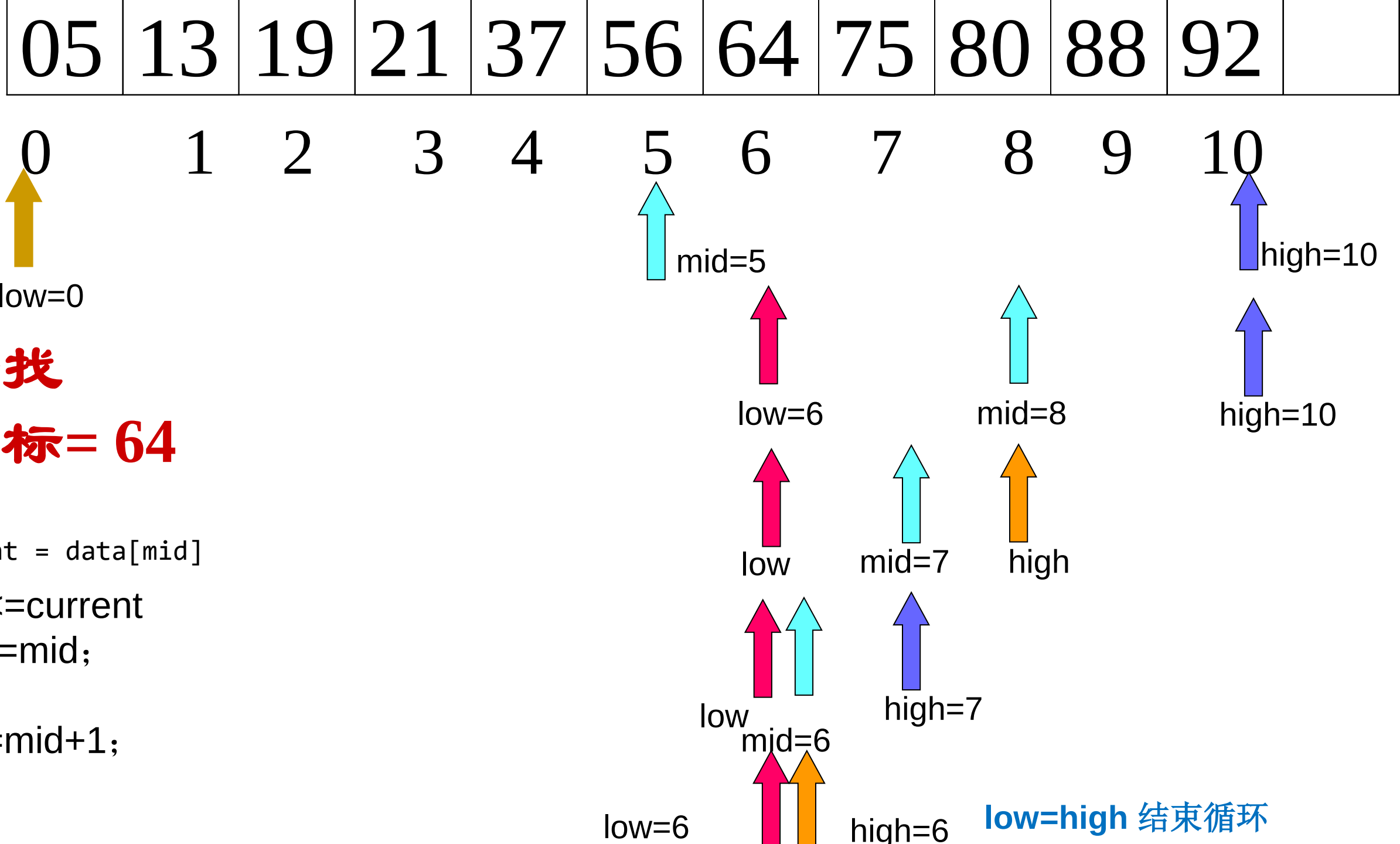


查找

目标= 68

```
current = data[mid]
```

当k<=current
high=mid；
否则
low=mid+1；



查找

目标= 64

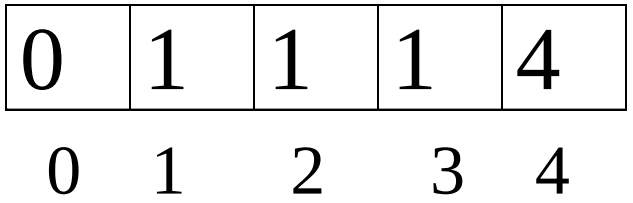
```
current = data[mid]
```

当k<=current
high=mid；
否则
low=mid+1；

特点

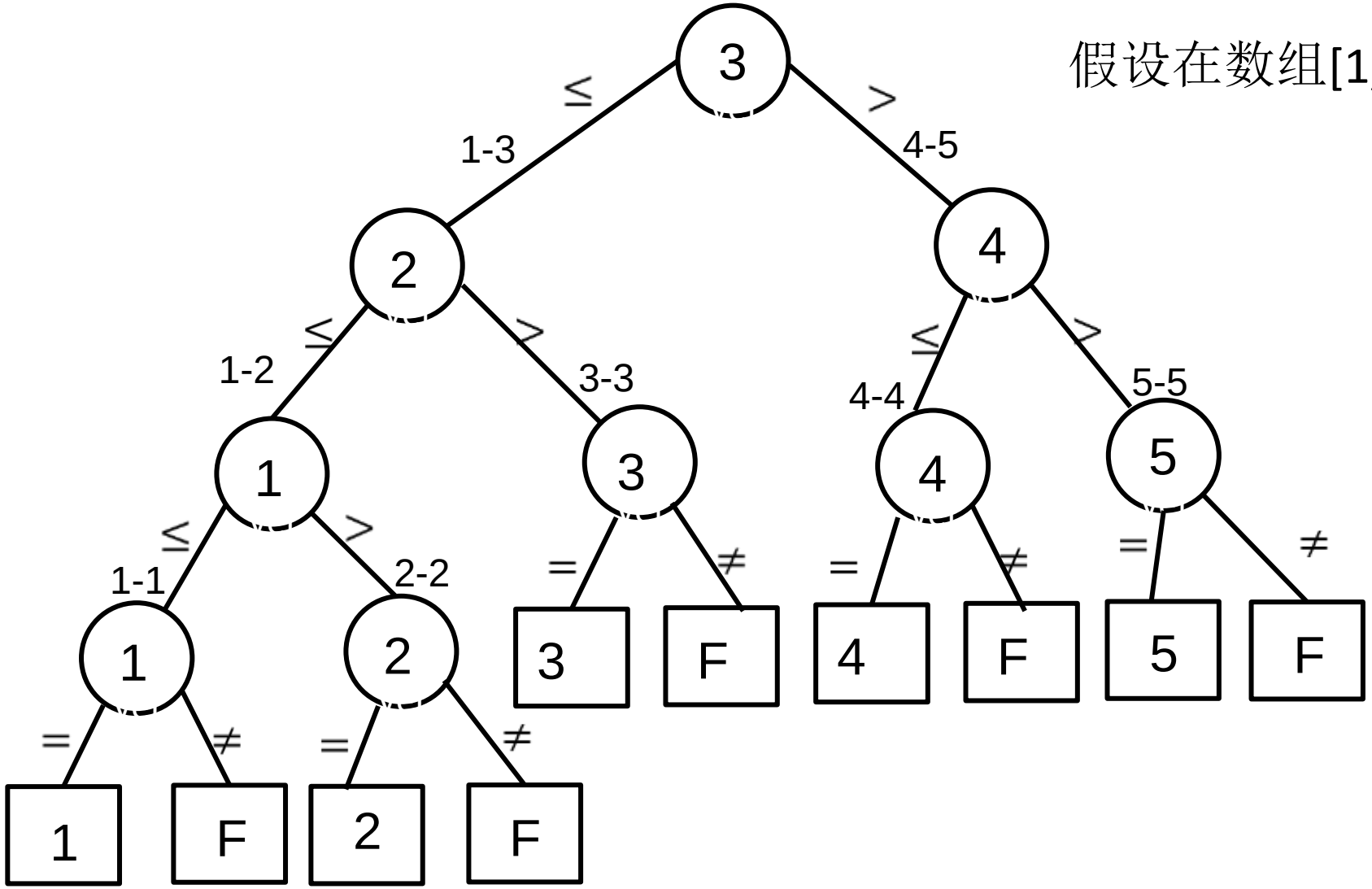
- 保证能找到符合条件的最左边的关键字

查找k=1



与识别相等的折半查找的另一个区别：判定树

假设在数组[1,2,3,4,5]中进行二分查找



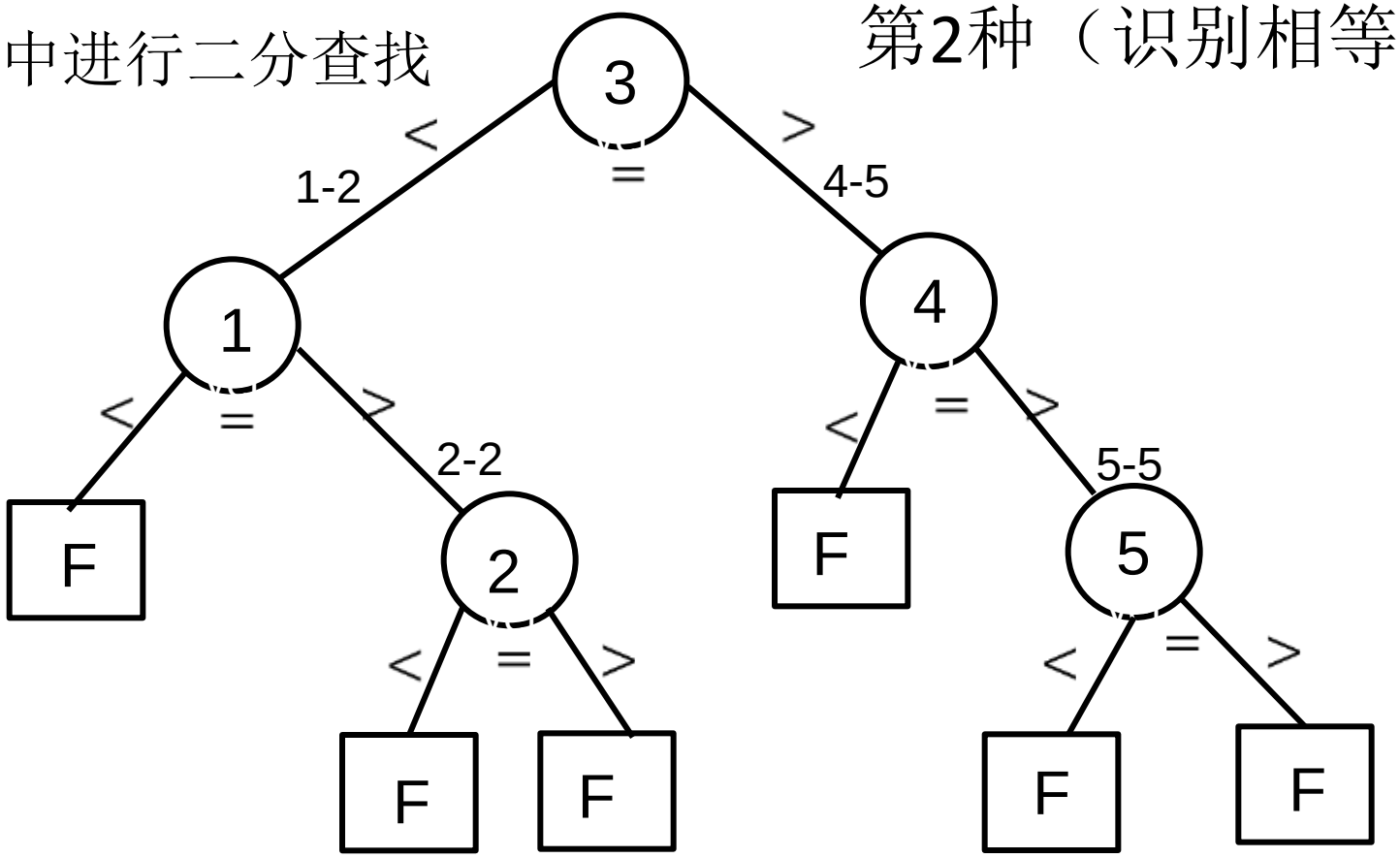
不识别相等
二分查找判定树画法

(成功时)平均比较次数：
(失败时)平均比较次数：

$$ASL=\sum_{i=1}^{n} p_i C_i = \frac{1}{5}(3*3+2*4)= \frac{17}{5}$$

假设在数组[1,2,3,4,5]中进行二分查找

第2种（识别相等）二分判定树



成功时：

$$ASL=\sum_{i=1}^{n} p_i C_i = \frac{1}{5}(1+2*3+2*5)= \frac{17}{5}$$

BinSearch1，成功比较次数 c=2*d-1，d是深度

失败时：

$$ASL=\sum_{i=1}^{n} p_i C_i = \frac{1}{6}(2*4+4*6)= \frac{16}{3}$$

因此，总体来讲，识别相等的二分查找效率没有不识别相等的高

单选题 5分

设置

1. 折半查找只能在数组中进行，且要求元素升序存放。

A 正确

B 错误

提交

单选题 5分

设置

2. 具有n个结点的识别相等的折半查找判定树，外结点有（ ）个。

A n

B n-1

C n+1

D 不确定

提交

单选题 5分

设置

3. 折半查找的时间复杂度是（ ）。

A O(1)

B O(n)

C $O(\log_2 n)$

D $O(n\log_2 n)$

提交

折半查找的平均查找长度

假设判定树的**深度为k**的满二叉树（n=2^k-1）,并且每个记录的查找概率相等，即p=1/n，因为判定树上第i层有2^{i-1}个结点

$$ASL = \sum_{i=1}^{n} p_i c_i = \frac{1}{n} \sum_{j=1}^{k} j * 2^{j-1}$$

$$= \frac{1}{n} (1*2^0+2* 2^1 +\ldots\ldots+k* 2^{k-1})$$

$$\approx \log_2(n+1) - 1$$

⇨ 时间复杂度为$O(\log_2 n)$

两种查找算法的比较

查找方法	存储结构要求	时间效率	优缺点	查找原理
顺序查找	顺序结构或链式结构的线性表	O(n)	当n较大时，查找比较耗时	从线性表的一端开始，将目标target依次与每个记录进行比较，直至找到一个与目标关键字相同的记录，或者直到表的另一端都没有找到为止。
二分查找	顺序存储结构、元素已有序	O(lgn)	需要事先保持记录有序	在一个有序顺序表下，将目标关键字与表的中间记录去比较，根据大小关系不断缩小被查区间。