



数据结构

教师：姜丽红 jianglh@sjtu.edu.cn

IST 实验室 <http://ist.sjtu.edu.cn>

助教：芮召普 ruishaopu@qq.com 江嘉晋 IST实验室 软件大楼5号楼5308

《软件基础实践》教师：杜东 IPADS 实验室 软件大楼3号楼4层



上海交通大学

SHANGHAI JIAO TONG UNIVERSITY

第8章 集合与静态查找表



- **集合的基本概念**
- **查找的基本概念**
- **无序表的查找**
- **有序表的查找**

集合：元素互相之间没有关系

学习目标：比较集合的不同实现方式，设计并实现满足时空复杂度要求的查找算法。

集合的基本概念



```
template <class KEY, class OTHER>
struct SET {
    KEY key;
    OTHER other;
};//可以包含除了关键字域等的多个数据域
```

- 集合的物理实现
- 任何容器都能存储集合
- 常用的表示形式是借助于线性表或树
- 唯一一个仅适合于存储和处理集合的数据结构是散列表

- 主要运算
- 查找某一元素是否存在
- 将集合中的元素按照它的唯一标识排序

静态查找技术



静态搜索表：集合中的结点总数是固定的或者很少发生变化。

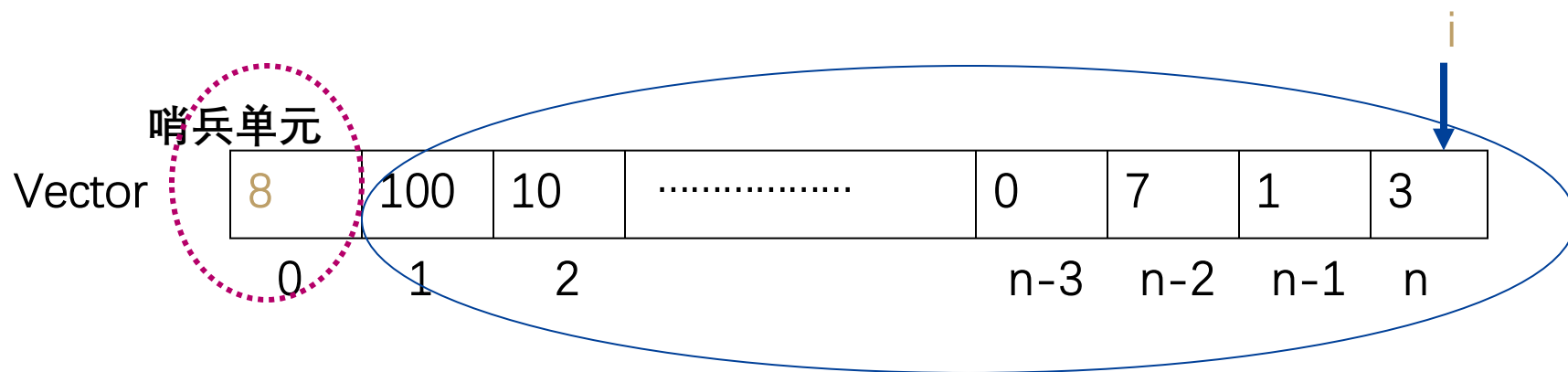
动态搜索表：集合中的结点总数是经常在发生变化。

在内存中进行的搜索：重点减少比较、或查找的次数。评价标准：平均搜索长度。

在外存中进行的搜索：重点在于减少访问外存的次数。评价标准：读盘次数。

静态查找

采用顺序存储，0号单元用作哨兵单元，1号单元到n号单元保存结点。



顺序查找：无序表的查找

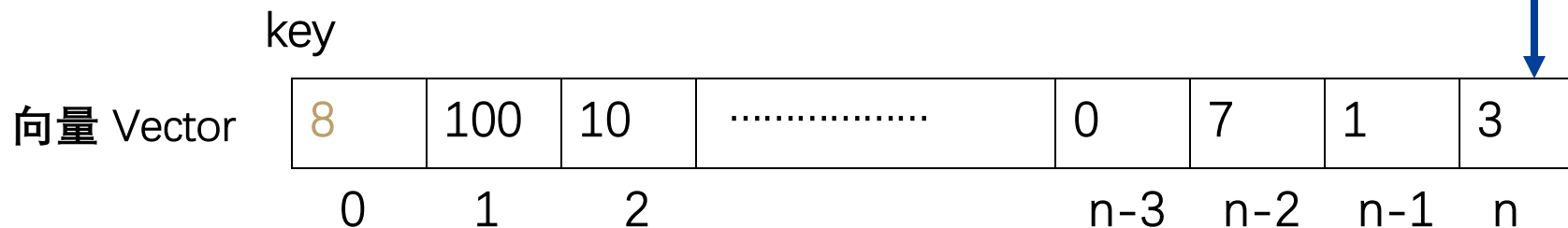


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 8$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

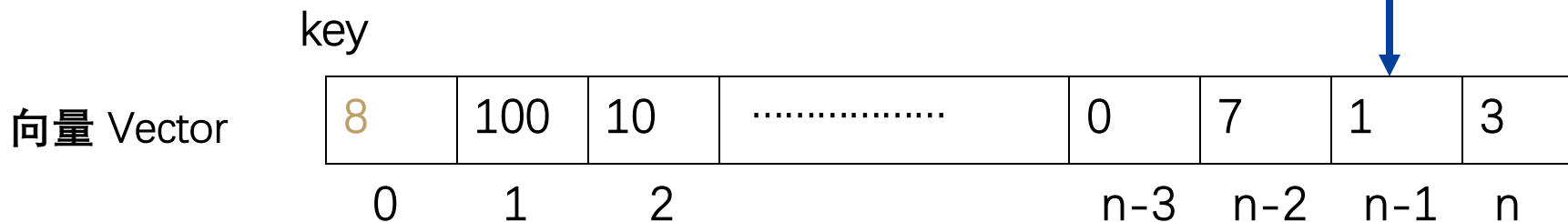


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 8$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

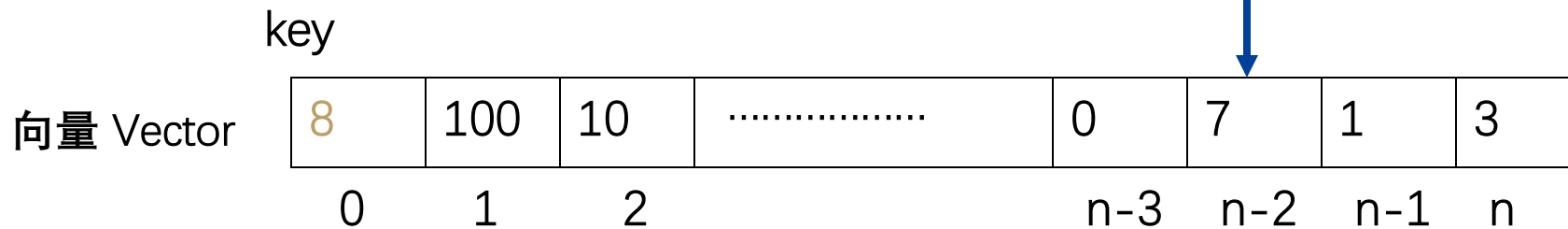


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 8$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

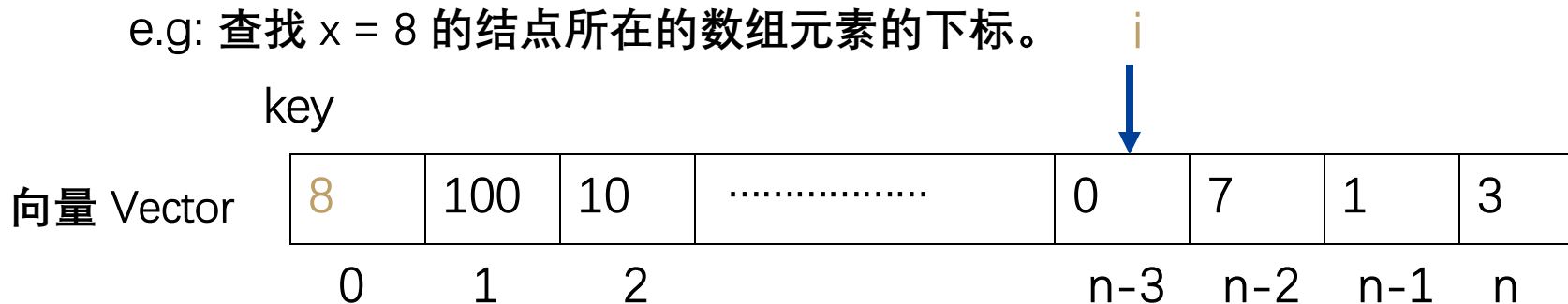


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 8$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

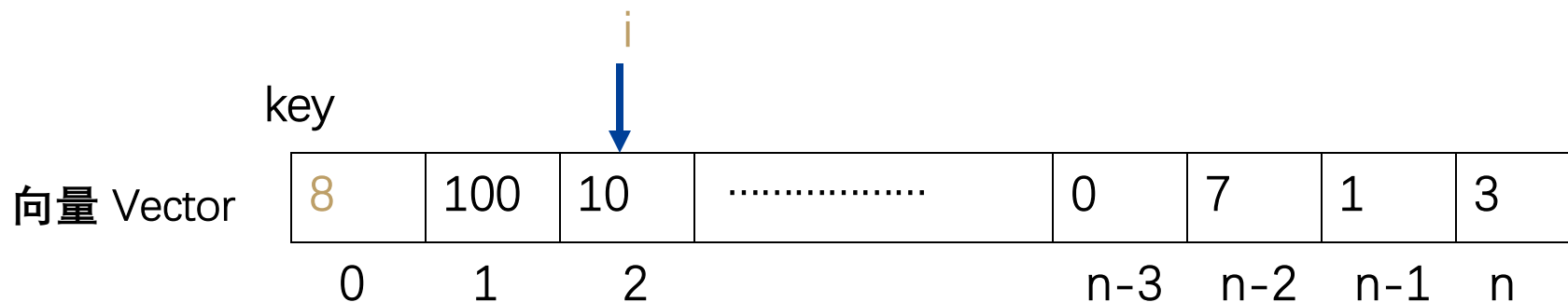


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 8$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

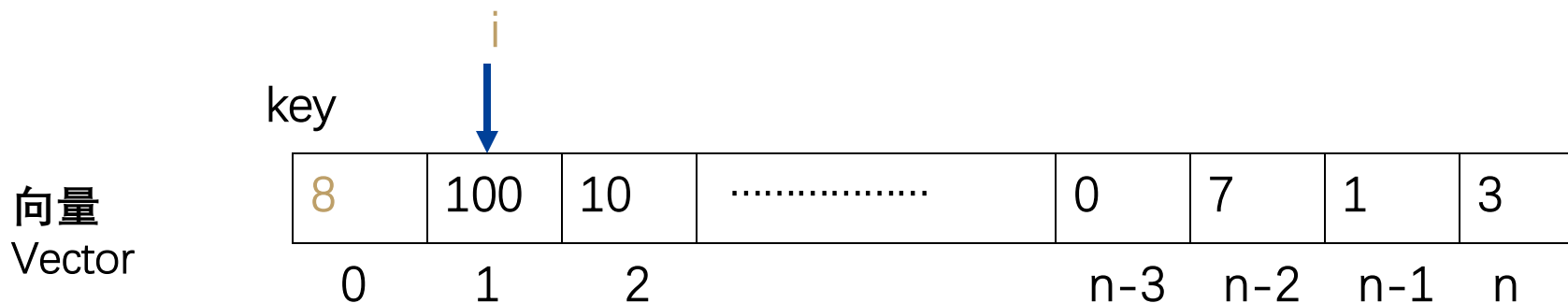


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 8$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

ununordered listordered list

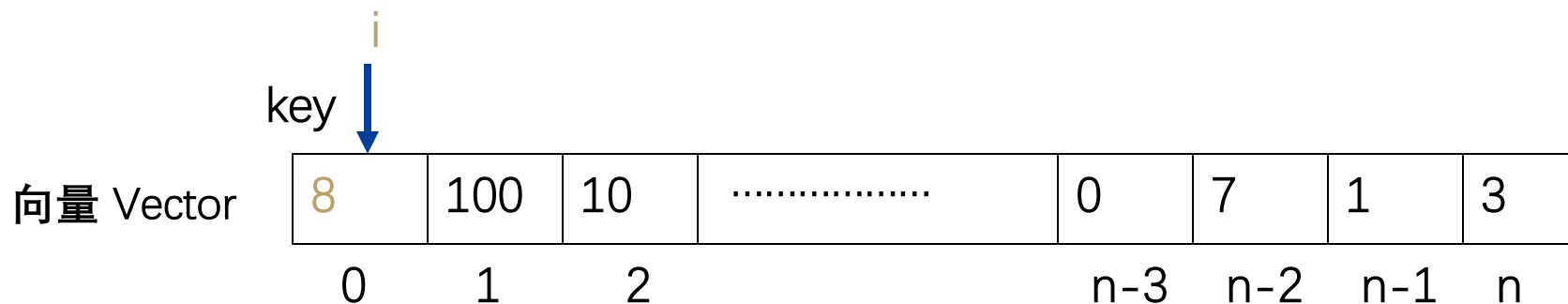


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 8$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

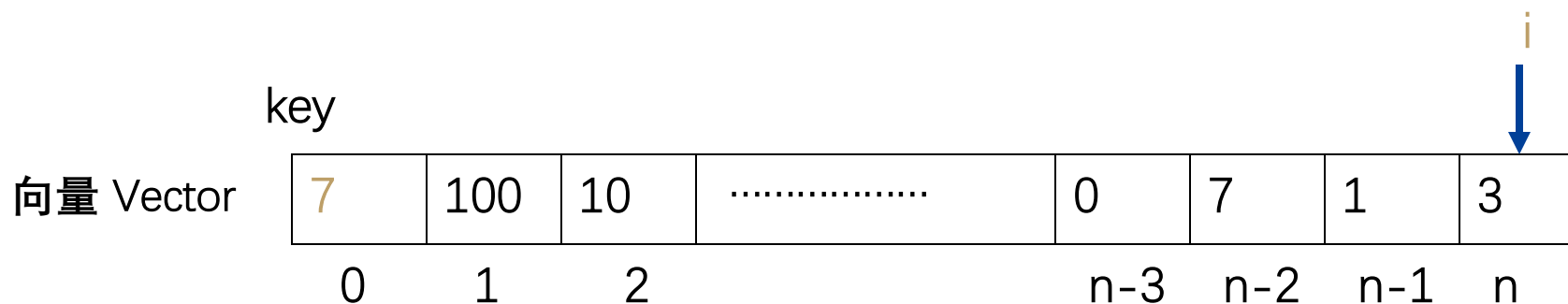


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 7$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

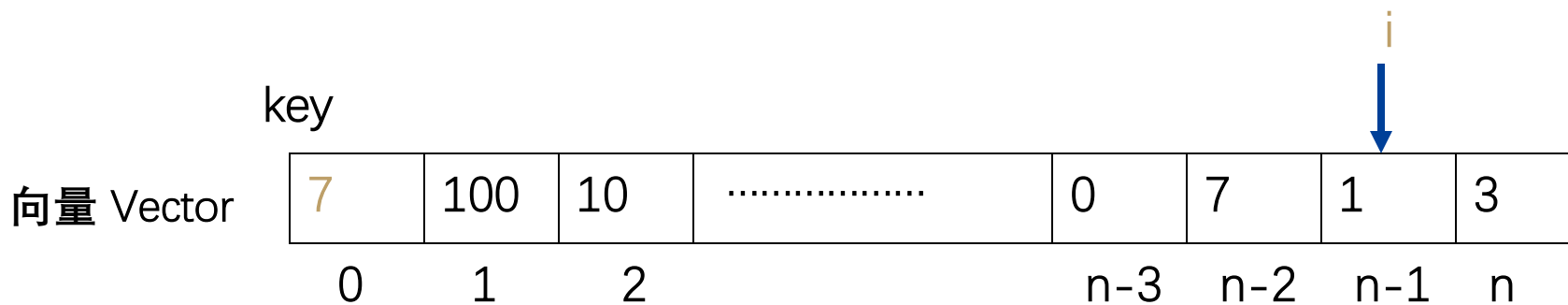


应用范围：顺序表或链表，表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 7$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找

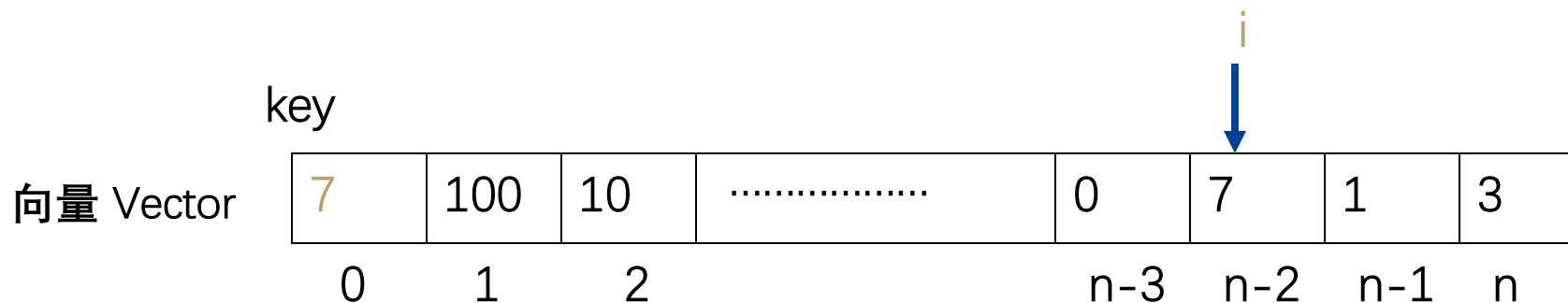


应用范围：顺序表或线性链表表示的静态搜索表。表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

e.g: 查找 $x = 7$ 的结点所在的数组元素的下标。



顺序查找：无序表的查找



应用范围：顺序表或线性链表表示的静态搜索表。表内元素之间无序或有序。

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

- 性能分析： n 为结点的总数。
- 平均查找长度AVL（Average Search Length）（比较次数）
只考虑成功查找情况下：设每个结点的查找概率相等

$$\begin{aligned} AVL &= \sum ((n-i+1) / n) \\ &= (n+1) / 2 \end{aligned}$$

$$1 \leq i \leq n$$

顺序查找：无序表的查找

应用范围：顺序表或链表，表内元素之间无序或有序。



```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[],
              int size, const KEY &x)
{
    data[0].key = x;
    for (int i = size; x != data[i].key; --i);
    return i;
}
```

// CurrentSize: 结点个数
设置哨兵的好处：在顺序表中总可以找到待查结点。否则，必须将判断条件 $i \geq 0$ 加进 for 语句。

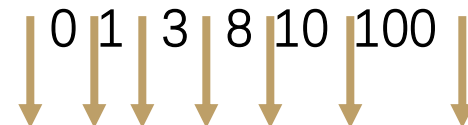
一般查找情况下（包括成功、不成功两种情况）：设成功与不成功两种情况概率相等，同时，假设每个结点的成功查找的概率也相等。

$$AVL = \sum ((n-i+1) / 2n) + (n+1)/2$$

$$= 3(n+1)/4$$

	100	10	0	8	1	3
--	-----	----	---	---	---	---

0 1 2 3 4 5 6



共有 $n+1=7$ 种不成功的查找情况

有序表的顺序查找



```
template <class KEY, class OTHER>  
int seqSearch(SET<KEY, OTHER> data[],  
              int size, const KEY &x)  
{  
    data[0].key = x;  
    for (int i = size ; x < data[i].key; --i);  
    if (x == data[i].key) return i;  
    else return 0;  
}
```

折半查找（或二分查找法）



```
template <class KEY, class OTHER>
int binarySearch(SET<KEY, OTHER> data[],
                int size, const KEY &x)
{
    int low = 1, high = size, mid;
    while (low <= high ) {           //查找区间存在
        mid = (low + high) / 2;      //计算中间位置
        if ( x == data[mid].key ) return mid;
        if ( x < data[mid].key) high = mid - 1;
        else low = mid + 1;
    }
    return 0;
}
```

适用范围？有序 + 顺序存储

折半查找（或二分查找法）

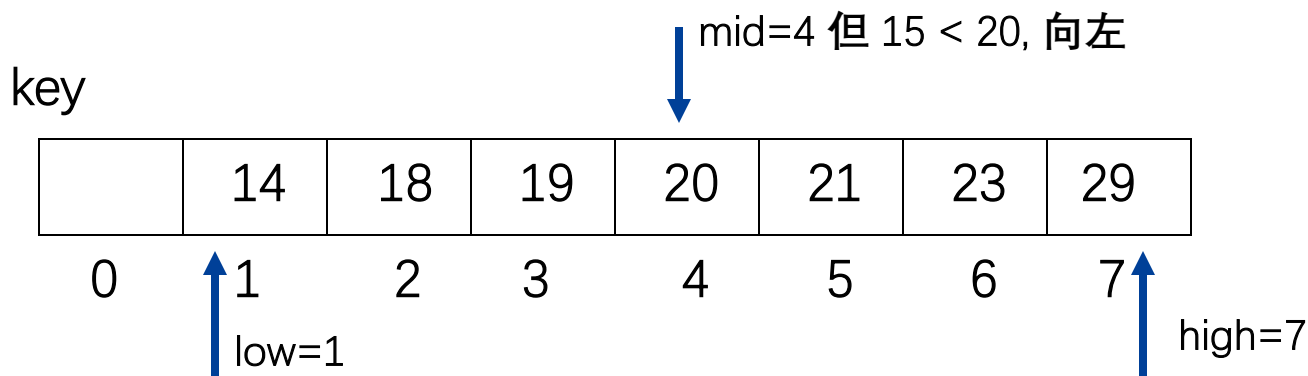


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]$. $Key \leq data[i+1]$. Key ; $i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 7（初始时为最大下标 n ）；
- 比较对象：中点元素，其下标地址为 $mid = (low + high) / 2$



折半查找（或二分查找法）

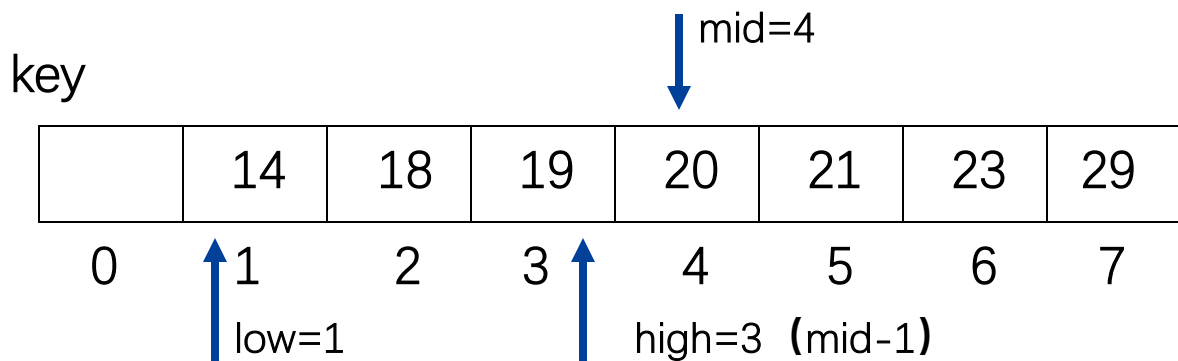


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]. Key \leq data[i+1]. Key; i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 3;



折半查找（或二分查找法）

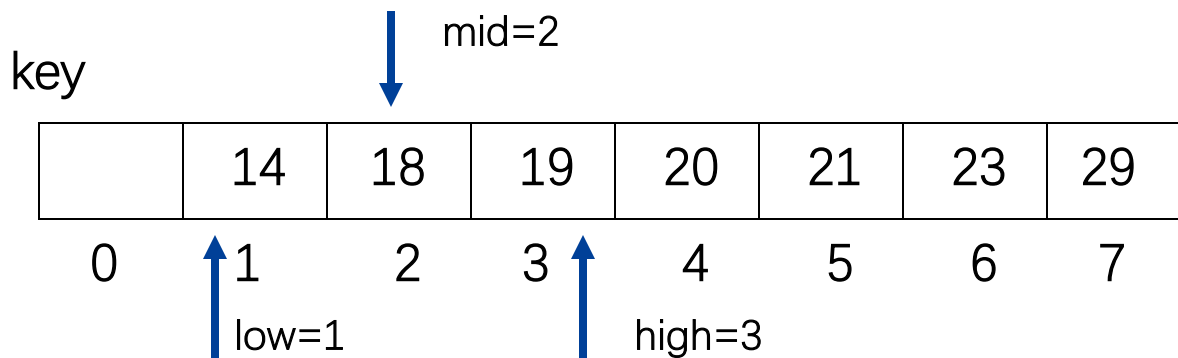


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]$. $Key \leq data[i+1]$. Key ; $i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 3;
- 比较对象：中点元素，其下标地址为 $mid = (low + high) / 2 = 2$



折半查找（或二分查找法）

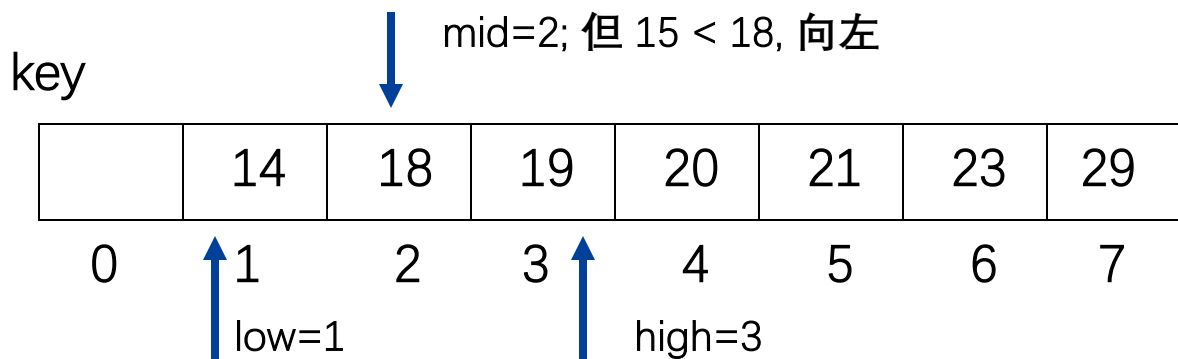


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]$. $Key \leq data[i+1]$. Key ; $i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 3;
- 比较对象：中点元素，其下标地址为 $mid = (low + high) / 2 = 2$



折半查找（或二分查找法）

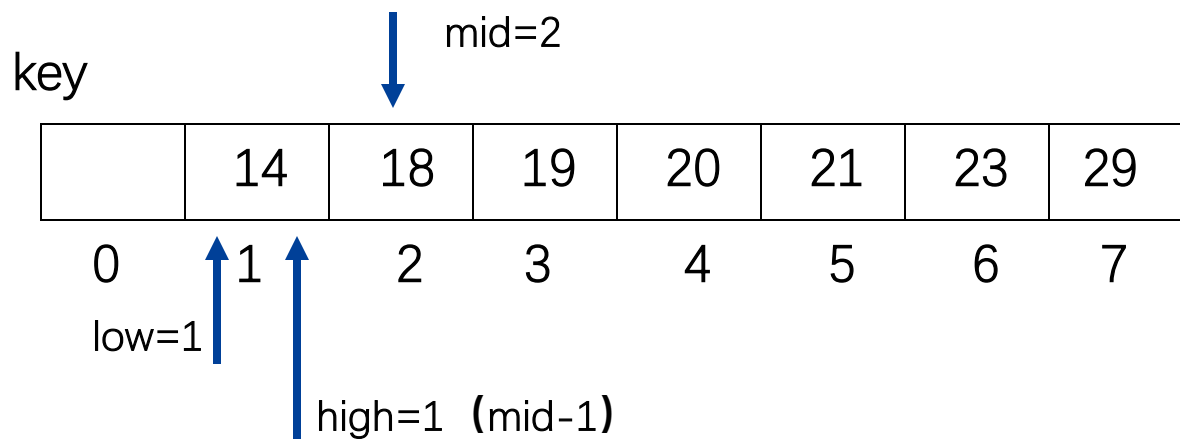


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]. Key \leq data[i+1]$. $Key; i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 1;



折半查找（或二分查找法）

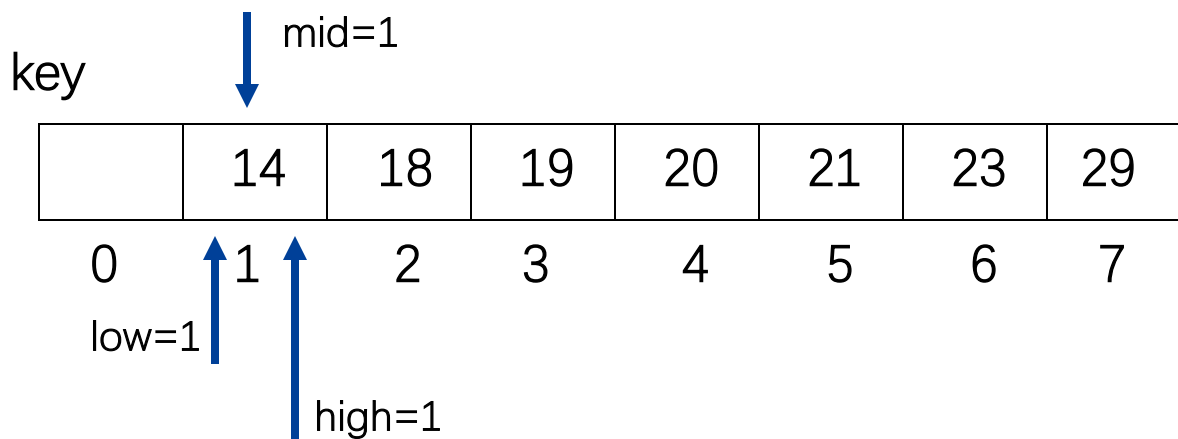


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]$. $Key \leq data[i+1]$. Key ; $i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 1;
- 比较对象：中点元素，其下标地址为 $mid = (low + high) / 2 = 1$



折半查找（或二分查找法）

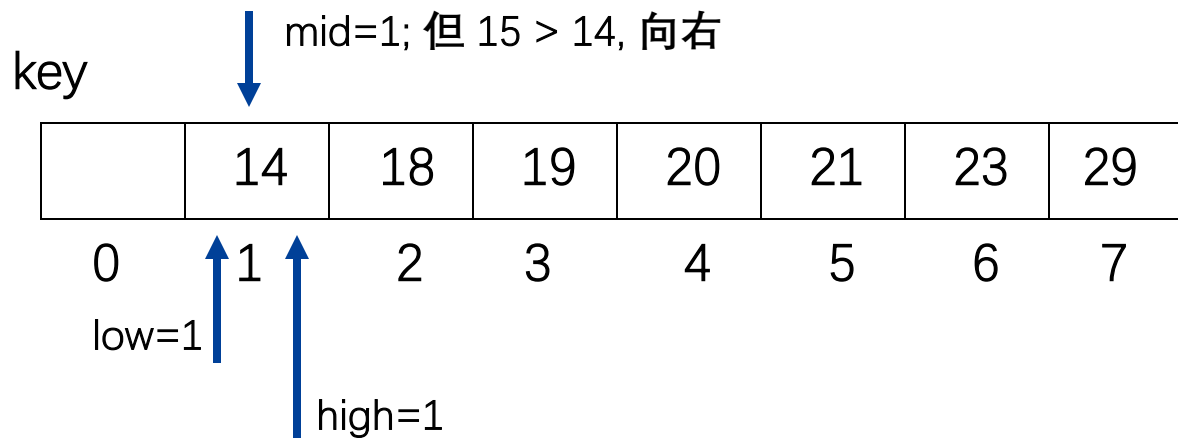


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]$. $Key \leq data[i+1]$. Key ; $i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 1;
- 比较对象：中点元素，其下标地址为 $mid = (low + high) / 2 = 1$



折半查找（或二分查找法）

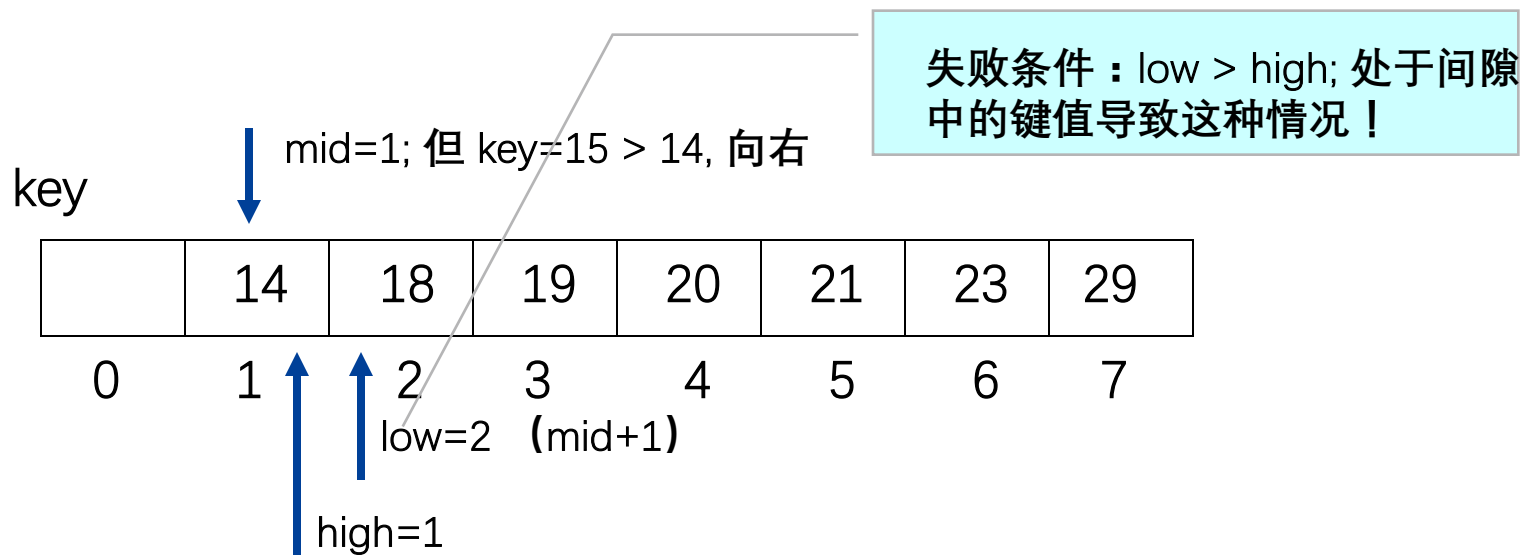


应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

e.g: 查找 $key = 15$ 的结点所在的数组元素的下标地址。

查找不成功的情况：数组 data 如下图所示有序

- 数组 data：递增序 $data[i]$. $Key \leq data[i+1]$. Key ; $i = 1, 2, \dots, n$
- 查找范围：low（低下标）= 1; high（高下标）= 1;
- 比较对象：中点元素，其下标地址为 $mid = (low + high) / 2 = 1$



折半查找（或二分查找法）

应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

注意： $(n-1)/2 = \lfloor n/2 \rfloor$

1、最坏情况分析：设 key 和中点的二次比较的时间代价 1

注意： $n/2 = \lfloor n/2 \rfloor$

如果 $n = 1$, 则 $low = high = mid$, 则代价为 1, 记为 $S(1) = 1$

如果 n 是奇数, 那么中点元素的左、右段各有 $(n-1)/2$ 个元素

如果 n 是偶数, 中点元素的左段有 $n/2-1$ 个元素; 右段有 $n/2$ 个元素

因此, 算法工作的那一段, 最多有 $\lfloor n/2 \rfloor$ 项

$$\begin{aligned}
 \therefore S(n) &= 1 + S(\lfloor n/2 \rfloor) \\
 &= 1 + 1 + S(\lfloor n/2^2 \rfloor) \\
 &= 1 + 1 + 1 + S(\lfloor n/2^3 \rfloor) \\
 &= \underbrace{1 + 1 + 1 + \dots + 1}_{\text{总共 } k \text{ 个 } 1} + S(\lfloor n/2^k \rfloor)
 \end{aligned}$$

当 $1 \leq n/2^k < 2$ 时; 则 $S(\lfloor n/2^k \rfloor) = 1$

此时: $2^k \leq n < 2^{k+1}$ 即 $k \leq \log_2 n < k+1$

注意: k 不可为小数, 它是正整数。 $\therefore k = \lfloor \log_2 n \rfloor$

故: $S(n) = 1 + \lfloor \log_2 n \rfloor$

折半查找（或二分查找法）

应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。

1、最坏情况分析：

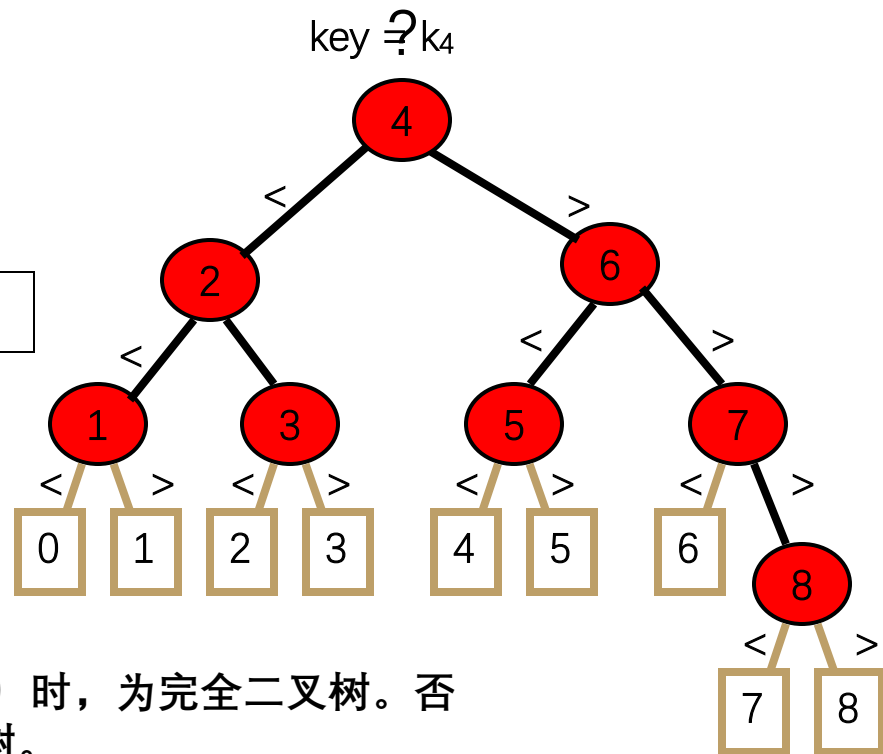
定理：在最坏情况下，二分查找法的查找有序表的最大的比较次数为 $1 + \lfloor \log_2 n \rfloor$ ，大体上和 $\log_2 n$ 成正比。

也可用判定树的方法进行推导。

如：

1	2	3	4	5	6	7	8
4	8	9	10	11	13	19	29

当寻找 $key = 9$ 及小于、大于 9 的键值的相应结点时，查找次数最大。达到了判定树的深度或高度。



注意：当判定树为 $n = 2^t - 1$ ($t=1,2,3 \dots$) 时，为完全二叉树。否则，除最下一层外，余为完全二叉树。

折半查找（或二分查找法）

应用范围：顺序表，表内元素之间有序。不可直接用于线性链表。



2、平均情况分析（在成功查找的情况下）：设每个结点的查找概率相同都为 $1/n$ 。为了简单起见，设结点个数为 $n = 2^t - 1$ ($t = 1, 2, 3, \dots$)。

\therefore 经过 1 次比较确定的结点个数为 $1 = 2^0$ 个。

经过 2 次比较确定的结点个数为 $2 = 2^1$ 个。

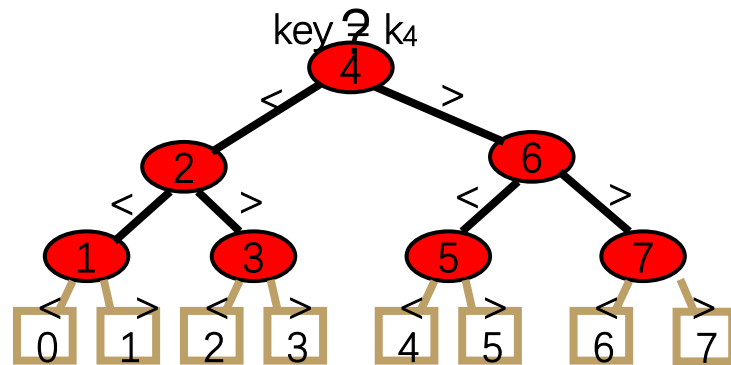
经过 3 次比较确定的结点个数为 $4 = 2^2$ 个。

经过 t 次比较确定的结点个数为 2^{t-1} 个。

注意： $\because 2^0 + 2^1 + 2^2 + \dots + 2^{t-1} = 2^t - 1$

\therefore 最多经过 t 次比较可以找到有序表中的任何一个结点

e.g: 当 $t = 4$ 时的例子：最多经过 $t=4$ 次比较找到任何一个结点



折半查找（或二分查找法）



2、平均情况分析（只考虑查找成功的情况下）：

$$\begin{aligned}\therefore ASL &= (2^0 \times 1 + 2^1 \times 2 + 2^2 \times 3 + \cdots + 2^{t-1} \times t) / n \\ &= \sum_{i=1}^t (i \times 2^{i-1}) / n \\ &= [(n+1) \times (\log_2(n+1) - 1) + 1] / n \\ &= (n+1) \times (\log_2(n+1) / n - 1)\end{aligned}$$

结论：在成功查找的情况下，平均查找的代价约为 $ASL = \log_2(n+1) - 1$

或者简单地记为： $ASL = \log_2 n - 1$

折半查找（或二分查找法）

3、平均情况分析（考虑成功、非成功查找两种的情况下）：

为了简单起见，设结点个数为 $n = 2^t - 1$ ($t = 1, 2, 3 \dots$)。成功查找的情况共有 n 种情况，非成功查找的情况共有 $n + 1$ 情况。

设每种情况出现的概率相同，即都为 $1/(2n+1)$ 。

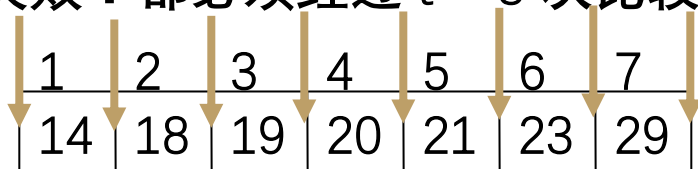
$$AVG = \left(\sum_{i=1}^t (i \times 2^{i-1}) + t \times (n + 1) \right) / (2n+1)$$

$$= \log_2 n + 1/2$$

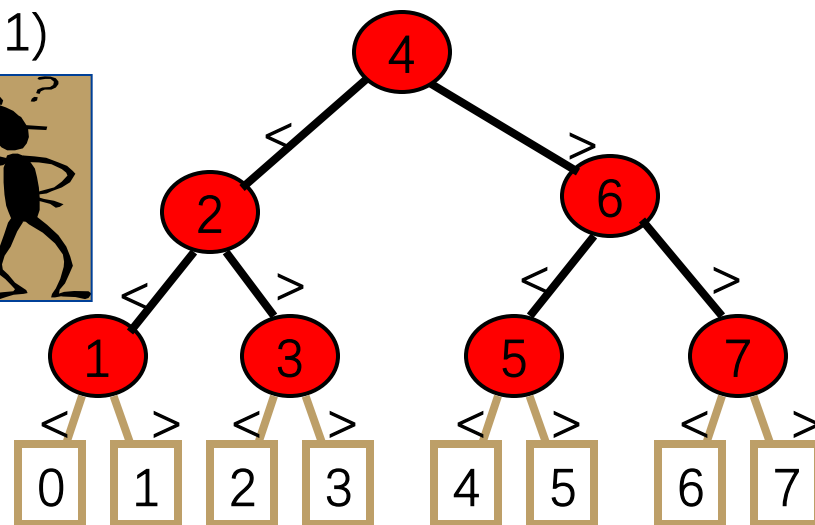
e.g: 当 $t = 3$ 时的例子：

成功：最多经过 $t=3$ 次比较

失败：都必须经过 $t = 3$ 次比较



共有 $7+1=8$ 种不成功的查找情况





插值查找

- 适用于数据的分布比较均匀的情况，可以快速定位
- 查找位置的估计

$$next = low + \left[\frac{x - a[low]}{a[high] - a[low]} \times (high - low + 1) \right]$$

- 缺点：计算查找位置比较复杂

分块查找：索引顺序块的查找

- 它把整个有序表分成若干块，块内的数据元素可以是有序存储，也可以是无序的，但块之间必须是有序的。

块内最大关键字	17	44	60
块起始地址	0	6	14	

3	10	9	6	14	17	19	34	23	44	39	20	42	18	60	48	51	58	47	...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	

查找由两个阶段组成：查找索引(有序)和查找块

总结



- **本章介绍了集合关系的基本概念，以及集合类型的数据结构中的基本操作。**
- **针对静态的集合，介绍了查找操作的实现。包括顺序查找、二分查找、插值查找和分块查找。**

学习目标：综合应用数据结构与算法设计方法，实现满足时空复杂度要求的查找任务。

练习题



- 一、对于长度为18的有序顺序表，若采用折半查找，给出查找到表中第15个元素所需要的查找次数是多少？
- 二、设有序顺序表中的元素依次为：017，094，154，170，275，503，509，512，553，612，677，765，897，908。试画出对其进行折半查找时的二叉判定树，并计算平均查找长度（假定查找成功与查找失败等概率分布，并且元素查找等概率分布）
- 三、线性表中各结点的查找概率不等时，可以采用如下策略提高顺序查找的效率：若找到所查找的元素，则将该元素与其直接前驱元素（若存在前驱元素）交换，使得经常被查找的元素尽量位于表的前端。试写出该算法（用单链表方式存储）。

Thanks! & QA

