数据结构

第十四讲　字典与集合

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年11月23日

课程内容

- 字典及其抽象数据类型
- 字典的线性表实现
- 二分法与插值法（interpolation）检索
- 集合的抽象数据类型及实现

数据存储、检索和字典

- 数据的存储和访问是计算中最重要最基本的工作，也是各种计算机应用和信息处理的基础。
- 在许多情况下，计算过程并不知道数据的存储位置，但需要使用数据，此时首先需要找到数据存储的位置，这一操作称为检索。
- 一些具体应用中数据的存储和检索（查询）的例子：
  - 电子字典，基本功能就是基于算法的数据检索
  - 图书馆编目目录和检索系统，支持读者检索书籍资料的有关信息
  - 规模巨大的有机物库，需要基于结构或光谱等参数进行检索
- 我们要讨论的是基于关键码的数据存储和检索
  - 关键码指数据项的某种（具有唯一性的）特征，可以是数据内容的组成部分，也可以是专门为数据检索建立的标签
  - 支持这种操作的数据结构，通常称为字典、查找表或映射

字典

- 字典就是实现存储和检索的结构。需要存储和检索的信息和环境有许多具体情况，因此要考虑各种不同的字典实现技术
- 字典的实现可以用到前面讨论过的许多想法和结构。包括
  - 各种线性结构、树性结构及其各种组合
  - 涉及到在这些结构上操作的许多算法
  - 组织方法很多，下面讨论顺序、散列、二叉树和其他树形结构等
  - 这里的基本问题是空间利用率和操作效率
- 字典的最主要也是使用最频繁的操作是检索（searching，也称查找）
  - 检索效率是字典实现中最重要的考虑因素
  - 由于规模不同，检索效率的重要性也可能不同

字典

- 字典可以分为两类：
  - 静态字典：建立后保持不变，只做检索，实现只需考虑检索效率
  - 动态字典：内容经常动态变动的字典。除检索外，基本操作还包括插入和删除等，实现时就必须考虑插入删除操作的效率
- 动态字典的插入删除操作可能导致字典结构的变化。要支持长期使用，还需要考虑字典在动态变化中能否保持良好的结构，能否保证良好的检索效率？（字典的性能不应该随着反复操作而逐渐恶化）

字典的基本概念

- 字典：一种特殊的集合；
  - 每个元素都有两部分组成，即关键码和属性(也称为值)：
    - 可以依据关键码对字典元素进行排序。
    - 字典元素的插入、删除和检索等操作一般以关键码为依据进行。
- 包含关键码和属性的二元组称作关联。
  - 关联<*k*,*v*>：关键码值*k*到属性值*v*的一个对应。
  - 字典: 从关键码值集合到值集合的二元关系:

$$D=\{<k,v> \mid k\in K \wedge v\in V\}。$$

- 静态字典：一经建立就基本固定不变。
- 动态字典：需要经常更新。

字典的基本概念

- 关于字典最主要的操作：
  - 检索(也称查找)给定一个值key，在字典中找出关键码等于key的元素。
  - 静态字典选择存储需要考虑：
    - 检索效率；
    - 空间的利用效率。
- 动态字典选择存储需要考虑：
  - 存储效率
  - 检索效率
  - 元素的插入、删除运算的效率

字典检索算法的效率

- 衡量一个字典检索算法, 需要考虑
  - 平均检索长度*ASL* (Average Search Length) ;

    检索过程中对关键码的平均比较次数, 即

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

    其中 n 是字典中元素的个数，p_i 是查找第 i 个元素的概率，c_i 是找第 i 个元素的比较次数。
  - 检索失败的概率及各种失败情况所需花费的比较次数;
  - 空间开销;
  - 算法是否易于理解等等。

字典的抽象数据类型

- 假设用Dictionary表示抽象数据类型字典，
- 用DicElement表示字典元素类型，
- 用KeyType 来表示元素中关键码的类型，
- 用Position表示字典中元素的位置。

字典的抽象数据类型

ADT Dictionary **is**

operations

Dictionary createEmptyDictionary(void)

创建一个空字典。

int search(Dictionary dic, KeyType key, Position p)

在字典dic中检索关键码为key的关联的位置p。

int insert(Dictionary dic, DicElement ele)

在字典dic中插入关联ele。

int delete(Dictionary dic, KeyType key)

在字典dic中删除关键码为key的关联。

end ADT Dictionary

字典的实现

- 从最基本的存储需求看，字典也就是关联的汇集。
  - 前面讨论过的各种数据汇集结构都可用作字典的实现基础。
  - 例如线性表，是元素的线性的顺序汇集。如果以关联作为元素，就可以看作是字典。下面首先考虑这种实现。
  - 作为字典实现，最重要的问题是字典操作的实现。由于字典可能有一定规模，需要频繁执行查询等操作，操作的效率非常重要。

顺序存储结构C语言描述

```
typedef  int KeyType;
typedef  int DataType;
typedef struct {
        KeyType key;                    /* 字典元素的关键码字段  */
        DataType value;                 /* 字典元素的属性字段 */
} DicElement;
typedef struct {
        int MAXNUM ;                    /*字典中元素的个数上界*/
        int n;                          /*为字典中实际元素的个数 */
        DicElement *elem;               /*存放字典中的元素*/
}  SeqDictionary;
```

顺序检索算法的实现

```
int seqSearch(SeqDictionary * pdic, KeyType key, int * position) {
                                        /*在字典中顺序检索关键码为key的元素*/
  int i;
  for(i=0; i<pdic->n; i++)                    /* 从头开始向后扫描 */
     if(pdic->elem[i].key==key) {
         *position=i;
          return 1;
     }                                                /* 检索成功 */
   *position=pdic->n;
   return 0;                                          /* 检索失败 */
}
```

简单顺序表字典的性质

- 插入的元素放在最后，$O(1)$ 时间复杂性。
- 删除元素时需要先检索，确定了元素的位置后删除（表中删除元素）。
- 主要操作是检索（删除依赖于检索），分析其复杂性，考虑比较次数：

$$
\begin{aligned}
ASL &= 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n \\
&= (1 + 2 + \ldots + n)/n && \text{when } p_i = 1/n \\
&= (n+1)/2 = O(n)
\end{aligned}
$$

- 基于线性表的字典实现，优点和缺点都很明显。
- 在字典的动态变化中，各种操作的效率不变（因为都已经是效率很低的操作了）。

讨论

- 顺序检索的优点是算法简单且适应面广。无论字典中元素是否有序均可使用。缺点是平均检索长度较大，特别是当 n 很大时，检索效率较低。
- 一般情况下，字典中各元素的检索概率不相等。
- 则当$P_1 \geq P_2 \geq \ldots \geq P_n$时，平均检索长度最小。
- 因此，如果已知各元素的检索概率，则应将字典中元素按检索概率由大到小排序，以便提高检索效率。

有序顺序表与二分法检索

- 另一种提高检索效率的方法是将字典元素按关键码递增(或者递减)的顺序排序，这种按照关键码大小排序的顺序表称为有序顺序表。
- 对于有序顺序表，可以还采用顺序检索的方法进行查找。
- 下面介绍的另外一种检索方法，称为二分法检索，是专门为有序顺序表设计的。

二分法检索

- 二分法检索又称折半检索，是一种效率较高的检索方法，通过按比例缩小检索范围的方式，快速逼近要检索的数据。使用这种方法检索时，要求字典的元素在顺序表中按关键码排序（假设按升序排列）。
- 思想：首先将给定值key与顺序表中间位置上元素的关键码比较，如果相等，则检索成功；否则，若key小，则在前半部分中继续进行二分法检索，否则在后半部分中继续进行二分法检索；直到检索成功或范围中已经没有数据，则检索失败结束。
- 这样，经过一次比较就缩小一半的查找区间，如此进行下去，直到能够确定检索成功或检索失败为止。

用二分法查找关键码为key的元素

```
int binarySearch(SeqDictionary * pdic, KeyType key, int *position)
  {
 int low, mid, high;
 low=0;  high=pdic->n-1;
 while(low<=high) {
   mid=(low+high)/2;                              /* 当前检索的中间位置 */
   if(pdic->elem[mid].key==key)
        {*position=mid;  return(1);}                          /* 检索成功 */
   else if(pdic->elem[mid].key>key) high=mid-1;     /* 检索左半区 */
    else  low=mid+1;                                          /* 检索右半区 */
   }
  *position=low;  return 0 ;                                  /* 检索失败 */
}
```

二分法检索示例

- **二分法检索的具体示例：**

  以下面 11 个数的检索为例：

  关键码：　5　13　19　21　37　56　64　75　80　88　92

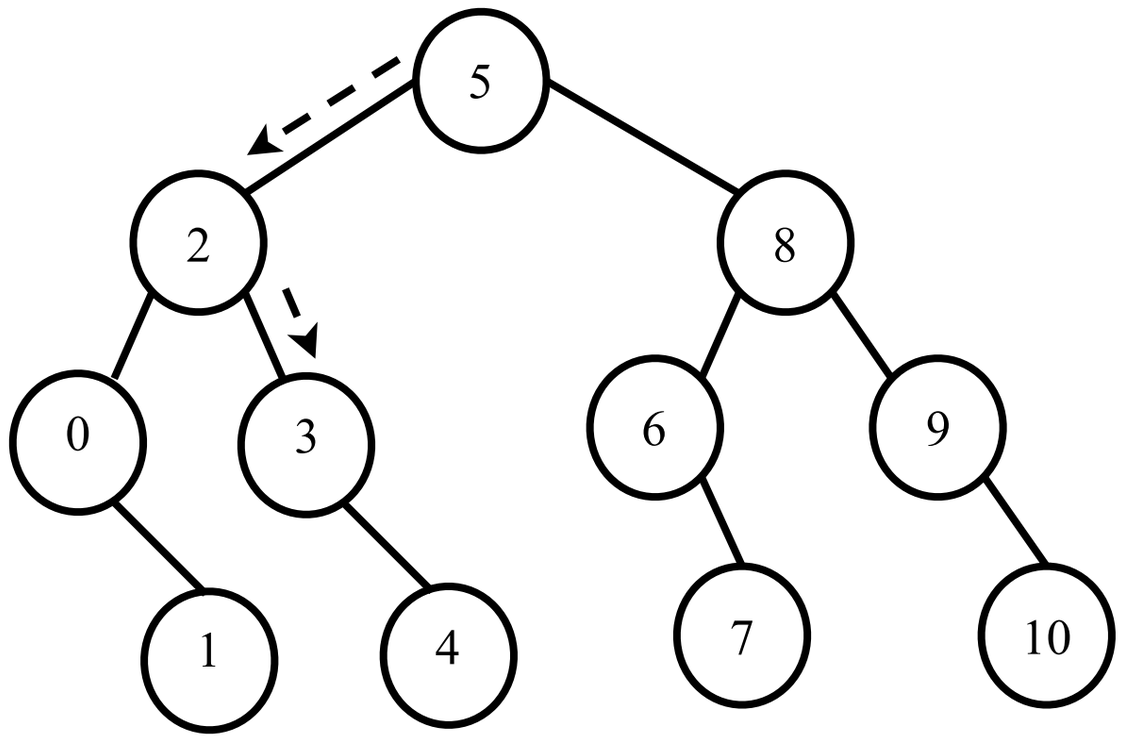
  位置：　0　1　2　3　4　5　6　7　8　9　10

  采用二分法检索：找到位置 5 的数需比较 1 次，找到位置 2、8 需比较 2 次，找到位置 0、3、6、9 需 3 次，找到另外 4 个位置需比较 4 次。

检索过程可用二叉树表示。树结点所标数字是数据的位置。

通过检索找到某结点的比较次数等于该结点的层数加 1。

检索结点的过程沿着从根结点到需要检索的结点的路径，在每个位置做了一次比较。



检索过程中对关键码的最大检索长度

二分法检索每经过一次比较就将检索范围缩小一半，

第i次比较可能比较的元素个数如下：

比较次数	可能比较的元素个数
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
⋮	⋮
j	2^{j-1}

若字典元素个数n刚好为$2^0+2^1+\ldots\ldots+2^{j-1}=2^j-1$则最大检索长度为 j；

若$2^j-1<n\leq 2^{j+1}-1$，则最大检索长度为$j+1$。

所以，二分法检索的最大检索长度为 :$\lceil log_2(n+1)\rceil$。

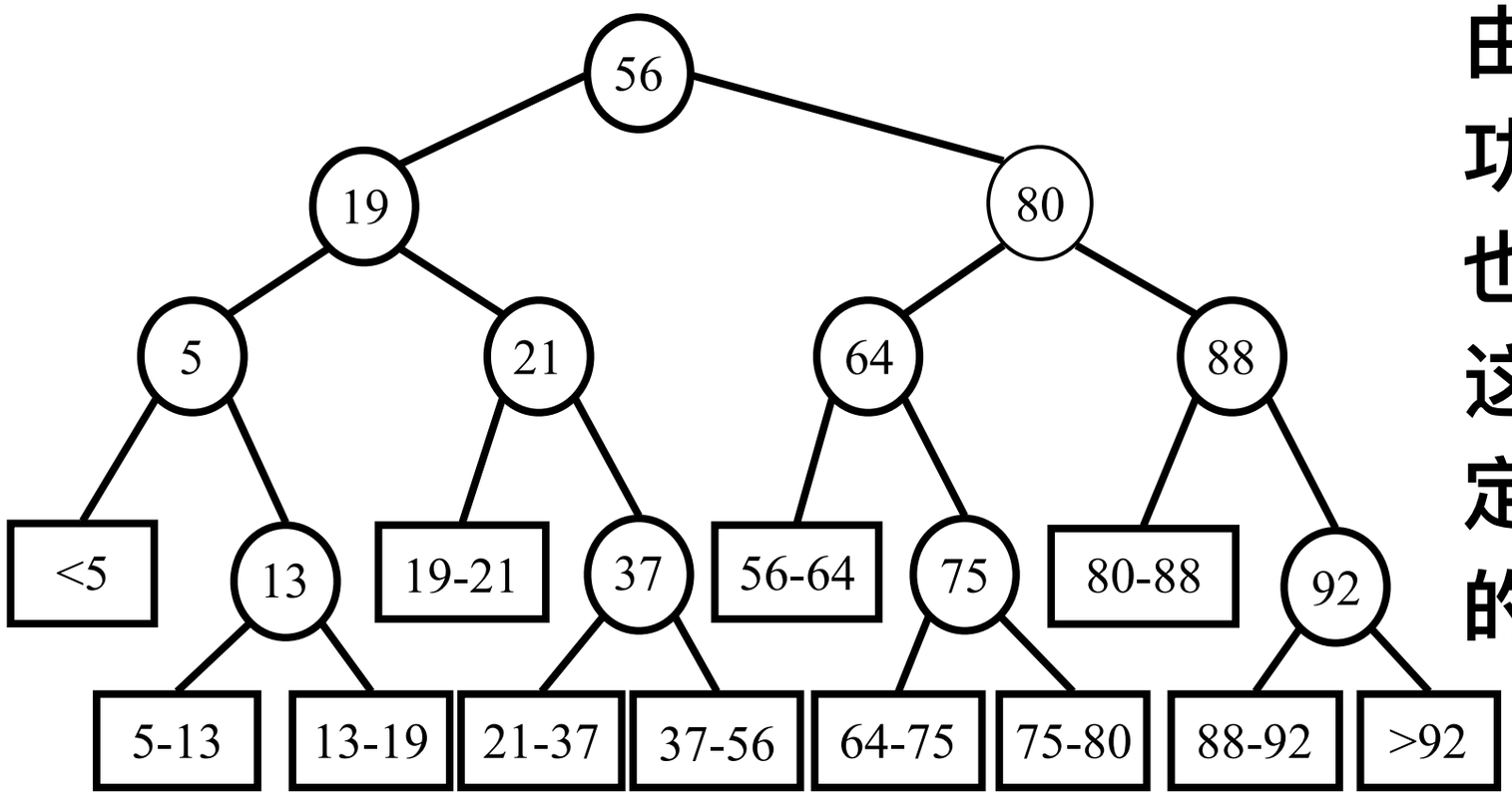
检索过程中对关键码的平均比较次数

- **设字典元素个数 $n=2^j-1$，检索每个元素的概率相同，则平均检索长度为：**

$$
\begin{aligned}
ASL &= \frac{1}{n} \times \sum_{i=1}^{j} \sum_{m=i}^{j} 2^{m-1} \\
&= \frac{1}{n} \times \sum_{i=1}^{j} (2^j - 2^{i-1}) \\
&= \frac{1}{n} \times (j \times 2^j - \sum_{i=1}^{j} 2^{i-1}) \\
&= \frac{1}{n} \times (j \times 2^j - 2^j + 1) \\
&= \frac{n+1}{n} \times log_2(n+1) - 1
\end{aligned}
$$

检索过程中对关键码的平均比较次数

- 包含检索成功和不成功情况的判定树如下。方框表示检索不成功（是扩充二叉树的外部结点），例如下图中标着 13-19 的方框表示被检索关键码值在 13 和 19 之间。检索到达方框就是失败。

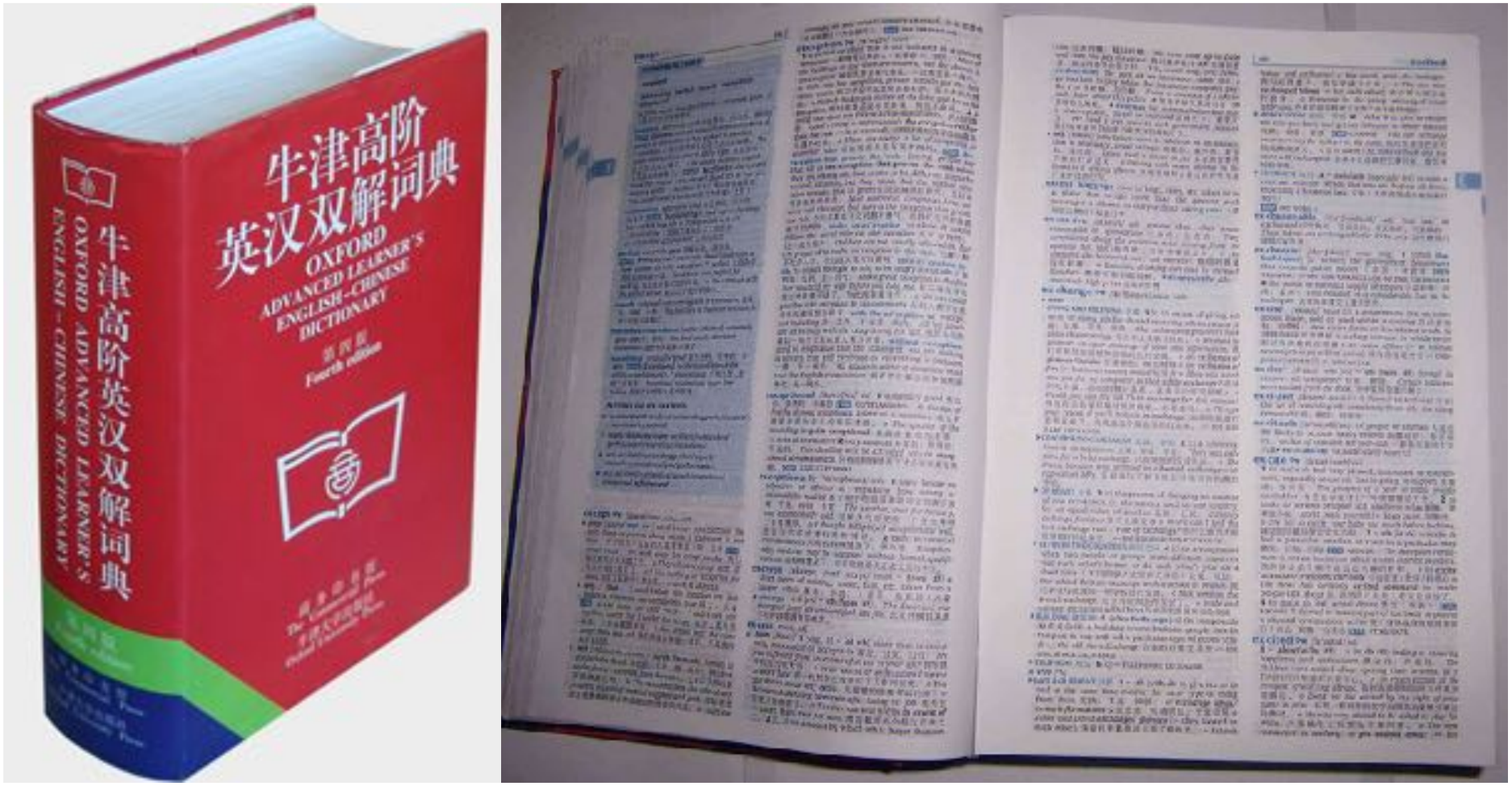


由此可见，检索不成功时，最大比较次数也是$\lceil log_2(n+1) \rceil$。这是基于检索中的判定操作形成的判定树的分析。

二分法检索的优缺点

- 二分法检索的优点是检索速度快($O(log_2(n))$)。
- 不足：
  - 只能用于顺序存储的元素按关键码排序的字典；
  - 为了保持顺序表的有序性： 插入和删除时需要移动元素($O(n)$时间代价)；
  - 不适合大的动态字典。

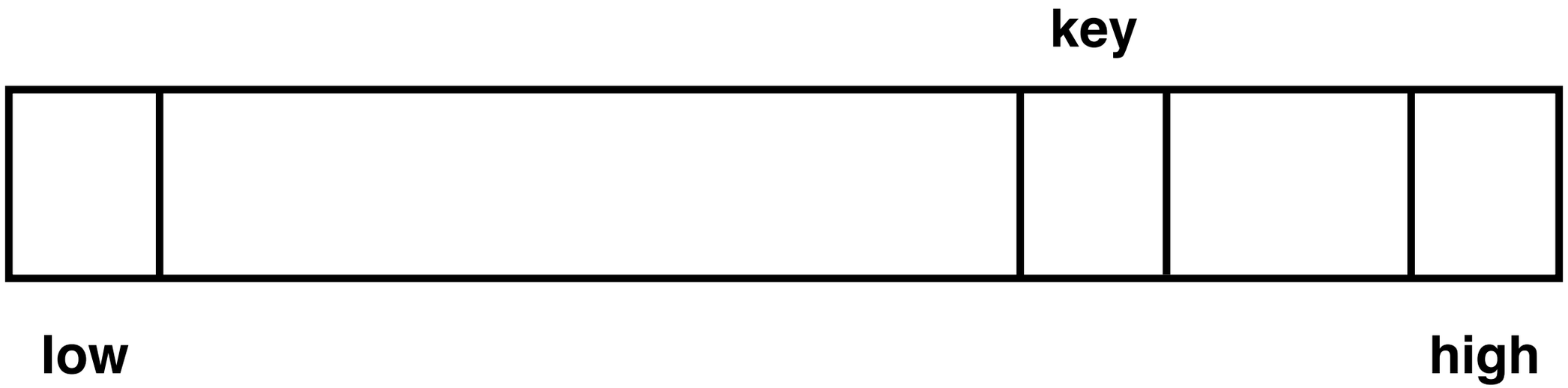
插值检索


牛津高阶英汉双解词典
OXFORD ADVANCED LEARNER'S ENGLISH-CHINESE DICTIONARY
第四版
Fourth edition

用插值法查找关键码为key的元素

```
int interpolationSearch(SeqDictionary * pdic, KeyType key, int *position){
  int low, mid, high;
  low=0;  high=pdic->n-1;
  while(low<=high) {
    mid = low + (key-pdic->elem[low] / pdic->elem[high]-pdic->elem[low])*
    (high-low);                          /* 插值检索与二分法检索的差别 */
    if(pdic->elem[mid].key==key)
         {*position=mid;  return(1);}                    /* 检索成功 */
    else if(pdic->elem[mid].key>key) high=mid-1;         /* 检索左半区 */
     else  low=mid+1;                                    /* 检索右半区 */
    }
  *position=low;  return 0 ;                             /* 检索失败 */
}
```

用插值法查找关键码为key的元素



字典的链表实现

- 也可以考虑用单链表或双链表实现字典，有关情况：
  - 如果字典里的数据项任意排列：插入时可以简单插入在表头，$O(1)$操作；检索和删除需要顺序扫描检查，$O(n)$操作；
  - 如果数据项按关键码升序或者降序排列，插入需要检索位置，$O(n)$操作；检索和删除需要顺序扫描检查，平均查半个表，$O(n)$操作；
  - 易见：用链接表实现字典没有任何优势，实际中很少采用。具体的实现技术很简单，不需要再进一步讨论。

字典的操作效率

- 采用线性表技术实现字典常常不能满足实际需要。
- 实际中需要存储和检索的数据集的数据量常常很大，内容动态变化。
  - 采用简单连续表或链接表，顺序检索的效率太低，不能满足存储和检索大规模的数据集合的需要。
  - 采用排序连续表和二分 / 插值检索，检索的速度大大提高，但仍有两大问题：
    1. 不能很好支持数据的变化（数据插入和删除）；
    2. 必须采用连续方式表示。如果数据集很大（例如几百 M 或者几个 G 或更大的数据集），连续实现方式就很难接受了。
  - 要支持在大且变动的数据集合里高效检索，必须考虑其他组织结构。

集合的抽象数据类型及实现

- 基本概念：
  - 集合是一些互不相同元素的无序汇集。
  - 集合中的元素也称为该集合的成员。
    - 集合中的成员可以是一个原子（不可再分解）；也可以是一个结构，甚至又是一个集合。
- 数据结构中讨论的集合，一般有以下限制：
  1. 限制为有穷集；
  2. 所有元素属同一类型；
  3. 元素之间存在一个小于关系 (有序集)。

集合的定义

- **列举法： 定义一个有穷集，可以将成员放在一对花括号中，成员之间用逗号隔开。**

$$\{1, 2, 3, 4, 5\}$$

- **谓词描述法:**

$$I = \{x \mid Z(x) \wedge x \geq 0\}$$

- **集合的大小: 集合中所包含的元素的个数。**
- **空集、子集、超集**
- $x \in A,\ x \notin A$

主要运算

- **集合间的高层运算：**

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$
$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$
$$A - B = \{x \mid x \in A \wedge x \notin B\}$$
$$A \subseteq B \Leftrightarrow \forall x.(x \in A \Rightarrow x \in B)$$
$$A \subseteq B \Leftrightarrow B \supseteq A$$
$$A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$$

- **元素的基本操作：**
  - **测试一个元素是否存在于集合中；**
  - **增加一个元素；**
  - **删除一个元素等。**

通过元素的基本操作实现集合间的运算

- 已知集合A和B，求它们的并集：
  - 只要以集合A（或B）为基础，把集合B（或A）的元素逐个插入。
- 如果要求两个集合的交集：
  - 只要从A（或B）出发，检查各元素是否在B（或A）中出现，选出那些也在另一个集合里出现的元素，插入（初态为空集的）结果集合。
- 求A与B的差集$A-B$时：
  - 只要以A为基础，对每个B中的元素做删除运算。

抽象数据类型

ADT Set **is**

operations

Set createEmptySet (void)

int member （Set A， DataType x）

int insert （Set A， DataType x）

int delete （Set A， DataType x）

int union （Set A， Set B， Set C）

int intersection （Set A， Set B， Set C）

int difference （Set A， Set B， Set C）

int subset （Set A， Set B）

end ADT Set

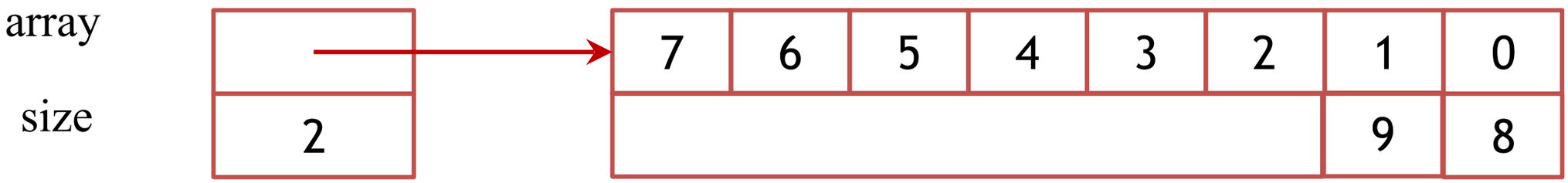
字典和集合

- 任何字典实现方法都可以用于实现集合：
  - 只需把集合的元素直接存储在保存字典项（关联）的位置。
  - 集合的最基本操作是元素与集合的关系，对应于字典查询；集合数据结构需要的创建、空集检查、加入、删除等都有字典操作与之对应。
- 集合实现需要考虑常用集合运算的实现：
  - 求并集和交集，求相对于某个集合的补集（集合差）都是从两个已有集合得到另一个集合。集合的实现需要考虑这些操作的效率。

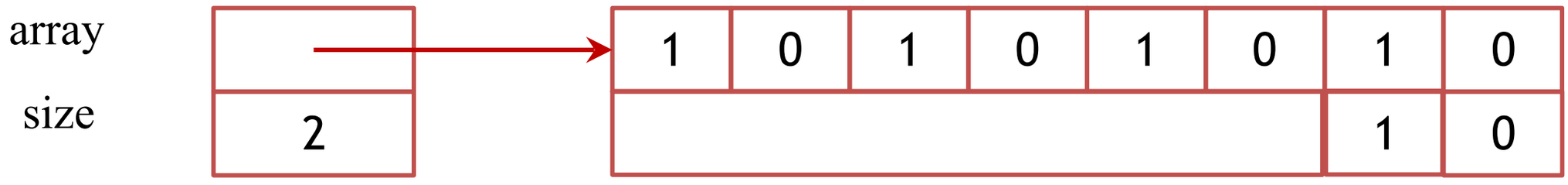
基于位向量表示的集合实现

- 一个元素是否属于一个集合，是一种二值判断。基于这一认识，人们提出了一种专门的集合实现技术：位向量表示。
- 位向量是一种每个元素都是二进制位（即0/1值）的数组。当表示的集合存在某个不太大的公共超集时，采用位向量的方式来表示这种集合往往十分有效。
- 如果所需要的集合对象有一个公共超集U，也就是说，需要实现的集合都是U的子集，就可以采用位向量技术实现这些集合，方法是：
  - 假定 U 包含 n 个元素，给每个元素一个编号作为该元素的下标。
  - 对任何要考虑的集合 S（注意 $S \subseteq U$），用一个 n 位的二进制序列（位向量）v_S 表示 S。对元素 $e \in U$，如果 $e \in S$，令 v_S 里对应于 e（的编号，下标）的那个位取值 1，否则令该位取值 0。

公共超集是{0,1,...,9}的数字集合的位向量表示



(a) 位向量表示存储结构示意图



(b) 集合{1,3,5,7,9}的实际存储状态

存储结构

- 假设需要表示的集合的公共超集中，共有*n*个不同的元素，为叙述的方便，不妨假设这些元素就是整数0，1，2，…，n-1。
- 每个集合可以采用一个有*n*位的位向量来表示。若整数 *i* 是这个集合的一个元素，则位向量中对应的位为1（真），否则为0（假）。
- 用位向量表示集合时，所占空间的大小与公共超集的大小 *n* 成正比，而与要表示的集合的大小无关。
- 在位向量表示里只有对应元素是否成员的标记，没有具体给成员的详细信息。

基于位向量表示的集合操作实现

- 位向量集合的元素操作
  - 加入删除元素，就是把位向量里相应的二进制位置 1 或置 0
  - 判断元素关系，对应于检查相应的二进制位是否为 1
- 位向量集合的集合运算都可以通过逐位操作实现：
  - S 和 T 的第 i 位都是 1 时 $S \cap T$ 第 i 位取值 1，否则取值 0
  - S 和 T 的第 i 位都是 0 时 $S \cup T$ 第 i 位取值 0，否则取值 1
  - S 第 i 位是 1 而 T 第 i 位是 0 时 $S - T$ 第 i 位取值 1，否则取值 0
- 例：假设 U 是 $\{a, b, c, d, e, f, g, h, i, j\}$，其子集
  - $S = \{a,b,d\}$: 1101000000
  - $T = \{a,e,i\}$: 1000100010
  - $S \cap T = \{a\}$: 1000000000
  - $S \cup T = \{a,b,d,e,i\}$: 1101100010
  - $S - T = \{b,d\}$: 0101000000

C语言定义

- C语言中无法直接定义位数组，只能定义字符数组。
- 一个字符占用8位二进制编码，它实际上包含了8个二进制位。
- 要定义长度为n的位向量，需要用长度为$\lceil n/8 \rceil$的字符数组。

位向量表示集合的存储结构

```
typedef struct {
    int size;           /*字符数组的长度*/
    char * array;       /*位向量空间。每一数组元素保存8位*/
} BitSet;
```

需要使用的位运算

假设x和y都是8位的字符，其值分别是：

X=　　01010111

Y=　　11011010。

各种字位运算，得到的结果如下：

- ~x　　10101000(求补)
- x & y　　01010010
- x ^ y　　10001101(按位加/异或)
- x | y　　11011111
- x << 3　　10111000
- y >> 5　　00000110

空集合的创建

```
BitSet * createEmptySet (int n) {          /*创建n位的位向量000...0*/
   int i;
   BitSet * s = (BitSet *)malloc(sizeof(BitSet));
   if (s!=NULL) {

   }
   printf( "Out of space! \n" ) ;          /* 存储分配失败 */
    return NULL;
}
```

空集合的创建

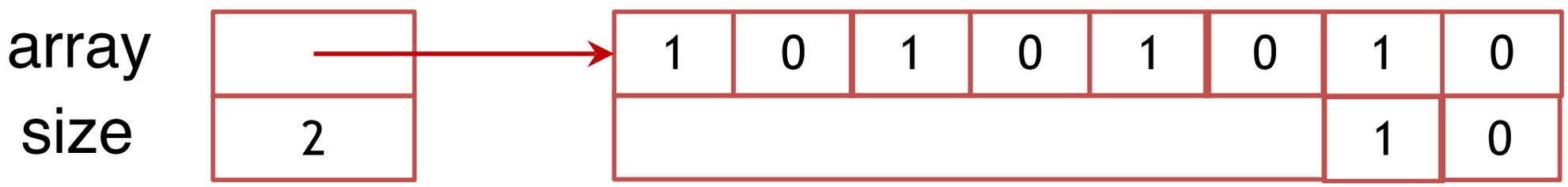
```
BitSet * createEmptySet (int n) {        /*创建n位的位向量000...0*/
   int i;
   BitSet * s = (BitSet *)malloc(sizeof(BitSet));
   if (s!=NULL) {
        s->size = (n + 7) / 8;
        s->array = (char *)malloc(s->size * sizeof(char));
        if (s->array != NULL){
              for (i = 0; i < s->size; i++)
                 s->array[i] = '\0';
              return s;
              }
   }
   printf( "Out of space! \n" ) ;          /* 存储分配失败 */
    return NULL;
}
```

判断index的元素是否属于集合

```
int member(BitSet * s, int index) {
    /*检查位向量中下标为index的位置是否为1*/
  if (index >= 0 && index >> 3 < s->size&&
            (s->array[index >> 3] & (1 << (index & 07))))
     return 1;
  return 0;
}
```

判断index的元素是否属于集合

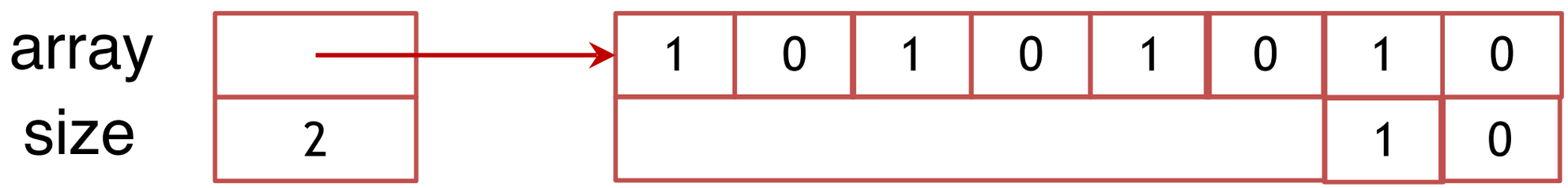
```
int member(BitSet * s, int index) {
    /*检查位向量中下标为index的位置是否为1*/
  if (index >= 0 && index >> 3 < s->size&&
            (s->array[index >> 3] & (1 << (index & 07))))
      return 1;
  return 0;
}
```



集合{1,3,5,7,9}的实际存储状态

将值为index的元素插入集合

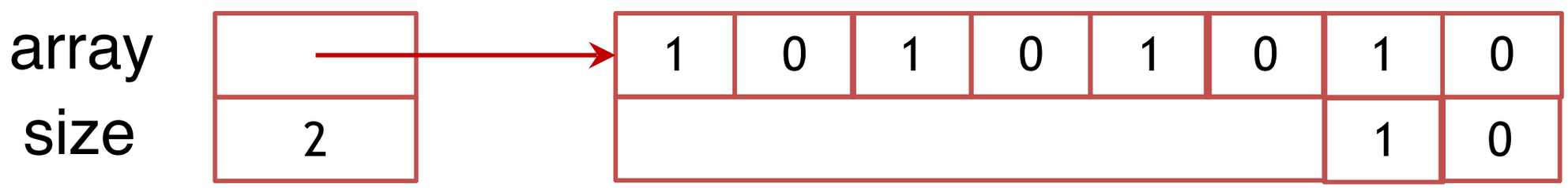
```
int insert (BitSet * s, int index){ /*将位向量中下标为index的位置为1*/
  if (index >= 0 && index>>3 < s->size) {
    s->array[index >> 3] |= (1 << (index & 07));
    return 1;
  }
  return 0;
}
```



集合{1,3,5,7,9}的实际存储状态

将值为index的元素插入集合

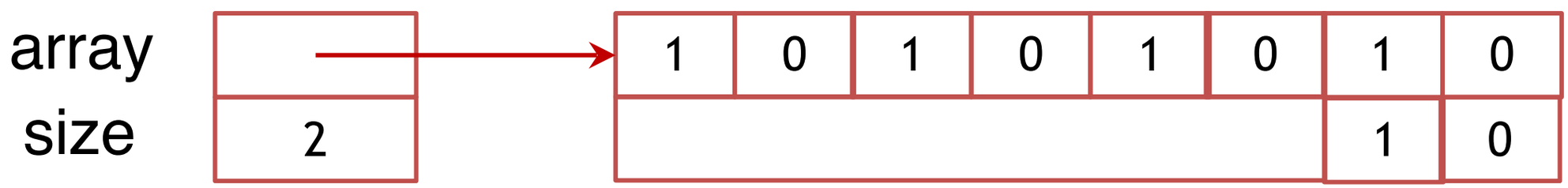
```
int insert (BitSet * s, int index){ /*将位向量中下标为index的位置为1*/
  if (index >= 0 && index>>3 < s->size) {
    s->array[index >> 3] |= (1 << (index & 07));
    return 1;
  }
  return 0;
}
```



集合{1,3,5,7,9}的实际存储状态

将值为index的元素从集合中删除

```
int delete(BitSet * s, int index) {/*将位向量中下标为index的位置为0*/
  if (index >= 0 && index >> 3 < s->size) {
    s->array[index >> 3] &= ~(1 << (index & 07));
    return 1;
  }
  return 0;
}
```



集合{1,3,5,7,9}的实际存储状态

集合的并

```
int union (BitSet * s0, BitSet * s1, BitSet * s2) {
/*当三个位向量长度相等时返回1，置s2为s0与s1的并；否则
   返回0。*/
   int i;
   if (s0->size != s1->size|| s2->size != s1->size) return 0;
   for (i = 0; i < s1->size; i++)
       s2->array[i]=s0->array[i] | s1->array[i];
   return 1;
}
```

集合的交

```
int intersection (BitSet * s0, BitSet * s1, BitSet * s2) {
/*当三个位向量长度相等时返回1，置s2为s0与s1的交；否则
   返回0。*/
   int i;
   if (s0->size != s1->size|| s2->size != s1->size) return 0;
   for (i = 0; i < s1->size; i++)
       s2->array[i]=s0->array[i] & s1->array[i];
   return 1;
}
```

集合的差

```
int difference (BitSet * s0, BitSet * s1, BitSet * s2) {
/*当三个位向量长度相等时返回1，置s2为s0与s1的差；否则
   返回0。*/
   int i;
   if (s0->size != s1->size|| s2->size != s1->size) return 0;
   for (i = 0; i < s1->size; i++)
       s2->array[i]=s0->array[i] & ~s1->array[i];
   return 1;
}
```

基于单链表表示的集合实现

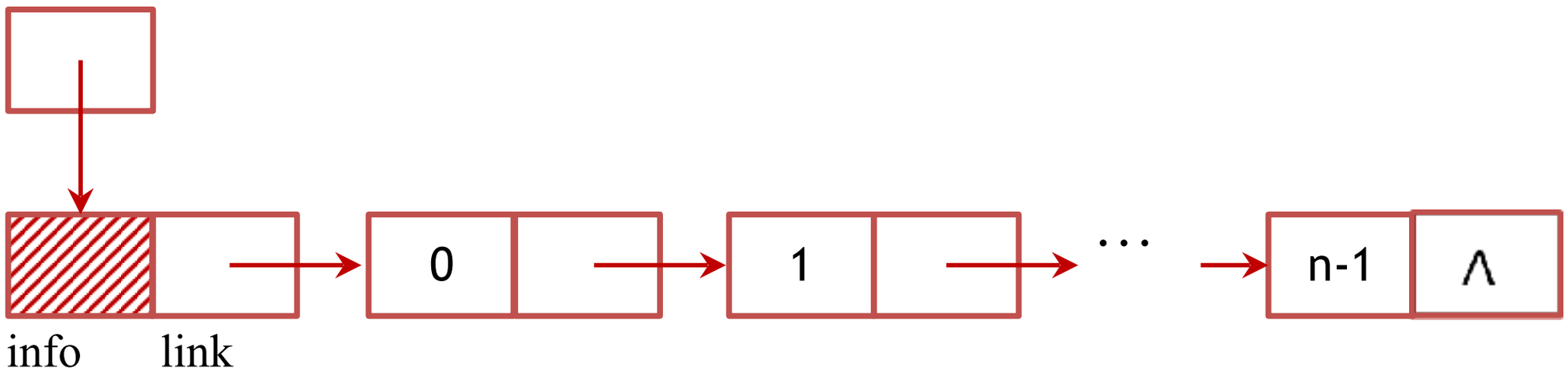
- 链接表示方式下，在链表中存放的是集合中元素的实际值，而不是元素是否属于集合的标记。
- 链表中的每个结点表示集合中的一个元素，具体方式与单链表的结点类似。
- 不同之处在于：线性表的单链表中，link字段表示线性表元素之间的逻辑后继关系，而在这里仅仅是把属于同一集合的所有元素链接成一个整体。
- 因为我们讨论的是有序集，如果将集合中的所有元素按“＜”关系排序构造有序链表，给集合的某些运算会带来方便。

C语言描述

```
struct  Node;
typedef  struct Node  *PNode;
struct  Node {
   DataType  info;
   PNode     link;
};
typedef  struct Node  *LinkSet;
```

例子

- 集合$S=\{0, 1, 2, \ldots, n-1\}$采用带表头结点的有序链表表示时，如图所示：



集合的交

```
int intersectionLink (LinkSet s0,LinkSet s1,LinkSet s2) {
  PNode x;
  if(s0= =NULL||s1= =NULL||s2= =NULL){
      printf("no head node error");return 0;}
  s2->link=NULL;                                          /*将s2置成空集合*/
  s0=s0->link;  s1=s1->link;
  while(s0!=NULL&&s1!=NULL)
  if(s0->info>s1->info) s1=s1->link;
  elseif(s0->info<s1->info) s0=s0->link;
  elseif(s0->info= =s1->info) {                           /*找到相同元素*/
    x=(PNode)malloc(sizeof(struct Node));                 /*分配结点空间*/
    if(x= =NULL)  {printf("out of space");return 0;}
    x->info=s0->info; x->link=NULL; s2->link=x;           /*在s2中插入*/
    s0=s0->link; s1=s1->link; s2=s2->link;                /*指针后推*/
  }
  return 1;
}
```

集合的赋值

```
int assignLink (LinkSet s0,LinkSet s1) {
  PNode  x;
  if(s0==NULL||s1==NULL){printf("no head node error");return 0;}
  s0->link=NULL;                                          /*将s0置成空集合*/
  s1=s1->link;
  while(s1!=NULL) {
    x=(PNode)malloc(sizeof(struct Node));          /*分配结点空间*/
    if(x= =NULL)  {printf("out of space");return 0;}
    x->info=s1->info; x->link=NULL; s0->link=x;  /*在s0中插入*/
    s1=s1->link; s0=s0->link;                             /*指针后推*/
  }
  return 1;
}
```

注意这一运算不能简单地将s0的表头结点置成s1的表头结点

集合的插入

```
int insertLink (LinkSet s0,DataType x) {
  PNode  temp;
  if(s0= =NULL){printf("no head node error");return 0;}
  temp=(PNode)malloc(sizeof(struct Node));                    /*分配结点空间*/
  if(temp= =NULL){printf("out of space");return 0;}
  while(s0->link!=NULL) {
    if(s0->link->info= =x) {printf("data already exist");return 1;}
    else if(s0->link->info<x) s0=s0->link;
    else if(s0->link->info>x) {                               /*找到插入位置*/
      temp->info=x; temp->link=s0->link; s0->link=temp;       /*插入*/
      return 1;
    }
  }
  if(s0->link= =NULL) {                                       /*插到最后*/
    temp->info=x;  temp->link=s0->link; s0->link=temp;
    return 1;
  }
}
```

本讲重点

- 集合和字典的概念，抽象数据类型，存储结构和操作的实现。
- 从逻辑结构上看，集合和字典属于两种最简单的数据结构，它们的元素之间没有任何确定的依赖关系。然而正是这个原因使得在具体表示集合和字典时，可以选择多种不同的存储结构。
- 集合的实现方法很多，当讨论的集合都是某个更大的公共超集的子集，并且这个公共超集不很大时，可以采用位向量的方式来表示。
- 字典的每个元素是一个二元组：分别称为关键码和属性，这种二元组称作关联。字典就是以关联为元素的集合。
- 字典顺序表示和顺序检索；有序顺序表表示和二分法检索。