# VEB 树的设计与实现


学　　院　计算机科学与技术

专　　业　计算机科学与技术

年　　级　**2003 级**

姓　　名　胡正军

指导教师　张坤龙

**2007 年 6 月 15 日**

# 摘　　要

在高效数据管理系统中，需要特殊的数据结构来支持快速查询和更新。在含有大量数据量的数据库系统中，查询速度显得尤为重要。查询除了简单的查询关键字外，还需要实现前驱查询或后继查询，往往一个数据结构可能在关键字查询上做的很好，但前驱和后继查询就不一定效率高。

VEB 树的主体数据结构用到了 van Emde Boas 递归数据结构，VEB 树就是为解决查询方面的问题设计的一个快速查询和更新的数据结构。它使用了哈希字典和 van Emde Boas 递归数据结构，来达到对数据操作的高效性。

本文先对实现 VEB 树的两个基本技术进行了分析和阐述，一个是哈希字典，另一个是 VEB 递归数据结构，并提出了两个优化技术：位映射和缓冲插入。接下来是 VEB 树具体实现的分析，最后通过实验的方式比较了 VEB 树和双向链表的性能。

本文得出的主要结论是：VEB 树远远优于双向链表的性能，它完成查询只需常量时间，前驱查询获得最坏的 $O(\log w)$ 时间复杂度，更新只需均摊的 $O(\log w)$ 时间复杂度。

**关键字**：前驱查询；VEB 树；哈希字典

# 目　　录

# 第一章　概述

## 1.1　研究背景

现代计算机的高速发展，使得计算机能够处理的数据量变得越来越大，例如一个几千万条记录的数据集。要在如此庞大的数据量中，查询某个数据元素，显然速度问题是最重要的，如何构建高效的数据管理系统就变得至关重要了。

通常数据集系统都是动态的，支持数据元素的查询、插入、更新和删除等一系列操作。本文将要设计的数据结构会体现这些数据特性，同时这些操作都要体现高效性，这就需要高效的数据结构和算法来实现。

数据集系统有两个基本问题。第一个基本问题就是把某个数据元素读出来，通常一个数据元素代表一个记录，给定一个数据元素，然后到数据集中去查询这个元素的是否存在，如果一条记录存了多项信息，为了方便查询与管理，数据集系统会指定一个关键字代表一条记录，那么这个基本问题就变成了查询一个关键字是否在数据集中。通常这个问题也叫成员关系或字典查询。另一个基本问题是前驱或后继问题。前驱问题和后继问题很相似，所以把它们归结到一起，通常是找到一个离给定元素最近的数据元素。比如一系列记录的关键字为：1，45，69，85，123，987，1562，1789，2563，这些关键字按升序排列，给定 987，它的前驱即是 123，后继就是 1562。

根据数据集的特性和两个基本问题，高效的数据集必然是能快速查询，快速更新，本文所设计和实现的 VEB 树使用 van Emde Boas 递归数据结构来实现快速更新与前驱后继查询，同时采用了哈希字典存储相应信息来加速查询。这个递归结构是 van Emde Boas 提出来的，在高速缓存参数无关算法里也起到重要作用。

## 1.2　主要概念术语

域：一个域表示一个关键字的范围，在本文中，限定关键字为整数，那么域表示为[0，N－1]，N 相当于域的大小。在计算机科学领域，计算机算法所遵循的计算模型为 RAM 模型，RAM 模型假设在加、减、乘、除、位逻辑运算，内存访问都在一个常数时间完成。用 w 表示 RAM 模型的机器字大小，w 通常为 2 的幂，现在普遍使用的机器字 w 为 32 和 64。由于关键字为整数，要让关键字在机器上用一个字表示，N 满足条件：$N<=2^w$。

关键字：一个关键字可以唯一确定一个数据元素。因此，在查询中可以只查询数据元素的关键字，只需要对关键字进行比较或处理，便可以得到数据元素的位置。

哈希表：哈希表在查找时不需要任何比较，就能一次存取得到所查的记录。因为记录的存储位置和它的关键字建立了一个确定的对应关系f，使得每个关键

字和结构中一个唯一的存储位置相对应。在查找时，根据对应关系*f*找到给定k的f(k)，如果结构中存在关键字和k相等的记录，则必定在f(k)的存储位置，由此不经过比较就便可直接得到所查的记录。本文称f为哈希函数，以这个思想建立的表为哈希表。

字典：一个字典维护了一个给定域的子集S，S里面的元素与某个值集合有一个映射关系，它支持三种操作insert(key,val)，delete(key)和 lookup(key)。前两种操作是更新，最后一种是查询，操作lookup(key)返回插入操作的val，否则返回nil。

前驱数据结构：一个前驱数据结构维护一个子集S，它支持三种操作insert(key,val)，delete(key)和 predecessor(key)，predecessor(key)返回小于等于key的最大元素。successor(key)相对的就是返回大于等于key的最小元素，所以本文假定前驱数据结构也支持后继查询。

## 1.3 论文内容

本文的第二章主要是 VEB 树的设计，包括哈希字典和 VEB 递归数据结构的设计两大部分。哈希字典提出了两种实现方法：两级完美哈希和 Cuckoo 哈希。第二章将对这两种哈希详细说明。

第三章是 VEB 树的具体实现，首先将对哈希字典和 VEB 树递归数据结构的细节进行分析。其次，介绍怎么用位映射节省空间和用缓冲插入减小更新开销。最后，是对查询、插入、删除和前驱后继查询等操作的详细实现。

第四章是对整个数据结构性能的理论分析，并把 VEB 树和双向链表用实验的方式进行了性能比较。

# 第二章　　VEB 树的设计

## 2.1　哈希字典

### 2.1.1　哈希字典和完美哈希

前面有对哈希表的解释，通过使用哈希策略，减少了对关键字的比较，使得查询将在常量时间内完成。构建哈希表的最重要部分是哈希函数，哈希函数让一组关键字映射到一个地址空间，比较普遍的构造哈希函数的方法有：

1 直接定址法

取关键字或某个线性函数值为哈希地址，即：H(key)=key 或 H(key)=a*key+b。这种构造哈希函数所得的地址集合与关键字集合大小相同，关键字也不会发生冲突，缺点是需要的地址空间太大。

2 除留余数法

取关键字被某个不大于哈希表长度 m 的数 p 做模运算所得的结果作为哈希地址。即：H(key) = key MOD p,p<=m.

3 平方取中法

取关键字平方后的中间几位作为哈希地址，取的位数由表长决定。

4 随机数法

选择一个随机函数，取关键字的随机函数值作为哈希地址，即 H(key) = random(key),这适合关键字长度不等时采用这种方法构造哈希函数比较好。

字典的一个简单实现是建立一个数组，这个数组大小为 N，数据类型为 V，用一个特殊值 nil 表示数据元素的不存在。用 lookup 查询时，只要不是 nil 就表示查询到了；用 insert 插入时，找到适当的位置插入；用 delete 删除时，只需要把那个关键字值设成 nil。这个字典的查询和更新的复杂度都是常量时间，缺点是就算字典所维护的子集 S 比域要小，同样需要跟整个域一样大的线性空间。当要维护的数据元素集合 S 的大小接近整个域时，这种实现才比较好。通常 S 比整个域小，那么用数组就浪费了很多地址空间。

为了减少空间使用，而且查询和更新都在常量时间内完成，使用哈希函数，建立哈希表来构建字典，本文称之为哈希字典。下面来看一下哈希字典怎样节省空间的使用。关键字域的范围是[0，N－1]，N 为 $2^w$，把[0，N－1]映射到一个小点的范围[0，s－1]。在[0，s－1]这个域里面，可以用上面提到的数组进行快速查询和更新。用用(key,val)作为数组元素值。当查询一个数据元素时，给定一个 key,用哈希函数 H(key)计算出一个值，假设字典数组 dict[H(key)]表示的数据元素是（key′, val′），如果 key= key′,表示查询成功，返回 val′，否则返回 nil。

这里考虑这样一个问题，把[0，N－1]用哈希函数映射到较小的关键字域[0，

s－1]，无法使域里每一个元素映射到一个唯一的值，这势必会产生冲突。举个简单的例子，把域[0，99]映射到域[0，99]，如果用取模的方法，关键字 27 MOD 20 与关键字 47 MOD 20 得到的结果是一样的，这样两个关键字表示的是同一个元素了，产生冲突。如果哈希函数 h 在某个域内是一一映射的，本文称这个函数 h 为完美哈希函数。当关键字在[0，N－1]这个域内，刚开始所用到的关键字比较少，用某个哈希函数 h 可能能满足一一映射，这时这个 h 是完美哈希函数。随着使用的关键字越来越多，冲突产生后，这个哈希函数 h 已经不能正常使用了，必须改变哈希函数 h，使得新的哈希函数成为完美哈希函数。为了节省空间，小的那个关键字域刚开始比较小，随着冲突产生，这个关键字域得调整大小，那么所使用的哈希函数也得随之改变。

构建哈希表的目标是找到完美哈希函数，使得整个哈希字典应用正常，下面将描述两种哈希方法实现本文哈希字典的设计。这两种方法都有常量的查询时间，使用线性空间，有常量分摊更新开销。这里用分摊的更新开销是因为在更新操作时引发改变哈希函数所带来的开销，并不是每一次更新操作都会引发哈希函数的改变，改变哈希函数的开销平均分摊到好几次更新操作上就获得了分摊的更新开销。

### 2.1.2 两级完美哈希

经典两级完美哈希的思想是：当发现为一个大子集S找一个简单的完美哈希函数很难时，很容易找到一个简单的第一级哈希函数，它能把大子集S的值用一级哈希函数映射到很多小集合中，每一个小集合可能包含好几个关键字的映射，这样再为每个小集合找到一个完美哈希函数，这样的小集合把它们称作关键字“桶”。第一级哈希函数的选择得使得映射出来的小集合足够小，小到本文能为每一个关键字“桶”找到一个完美哈希，见图2-1。这里选择的哈希函数的形式都是这样的：$\mathrm{H}_{p,s}\{x \to ((ax \bmod p) \bmod s) \mid 0 < a < p\}$ 每个函数被$a$参数化，不同的$a$代表不同的哈希函数，p是确定的，s就是哈希表元素的个数，如果是一级哈希函数，那么s就代表了关键字“桶”的个数，在每个关键字“桶”，重新选择一个$a$充当第二级哈希函数。

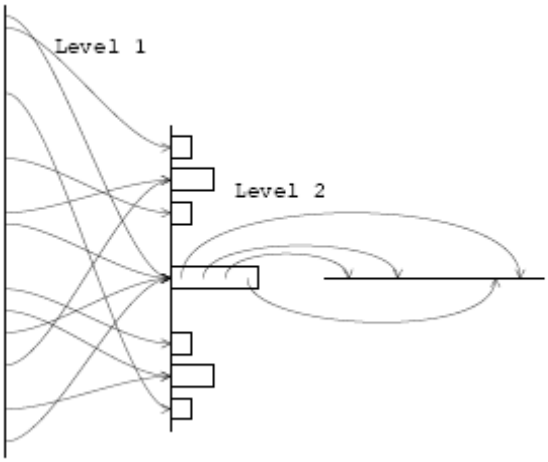


图2-1　　两级完美哈希

两级哈希方案中的关键字“桶”包含了相应关键字的完美哈希结构。当查找某个关键字key时，先用第一级哈希获得一个值h1(key)，根据这个结果找到相应的关键字“桶”，对应的第二级哈希函数为h2[h1(key)]，用h2[h1(key)](key)就可以找到关键字所映射的数据元素地址了。

### 2.1.3　Cuckoo 哈希

Cuckoo 哈希相比两级完美哈希的实现要简单，但和两级完美哈希有相同渐近的时间和空间的复杂度。两级完美哈希中，第一级哈希函数只有一个，但第二级哈希函数就有多个，这跟第一级哈希函数映射出来的关键字“桶”数有关，每个哈希桶对应一个自己的哈希函数。Cuckoo 哈希只使用了两个哈希函数，相应的也只构建了两个哈希数组，这样空间的使用将更小。

Cuckoo 哈希的原理如下：查询一个元素时，先用第一个哈希函数计算，若找到了这个关键字，就成功返回。若没找到，就再用第二个哈希函数计算，找到便成功返回。用第二个哈希若仍没找到，则说明这个关键字不存在字典中，返回nil。插入一个元素到 Cuckoo 哈希字典中时，要复杂一点，这也是 Cuckoo 哈希的关键。首先，用第一个哈希函数计算地址，检查这个关键字已经存在，若存在，更新数据即可。若不存在，检查计算得到的地址中包含的数据元素里面的关键字是否为空，如果为空，就用第一个哈希函数把这个关键字插入到第一个哈希表中，然后返回成功，如果不为空，说明这个位置已经被其他关键字占据了，将要插入的关键字 $key_1$ 就把这个已经存在的关键字 $key_2$ 挤出去了，然后将 $key_2$ 应用第二个哈希函数计算，看是否能在第二个哈希表中找到适当的位置插入，如果继续产生冲突，这将会导致一个一个 $key_3$ 的出现，这个 $key_3$ 又将用第一个哈希函数从第一个哈希表中找到适当位置插入，这样一直循环下去，直到 $key_n$ 找到位置插入后不再产生冲突，见图 2-2。当两个哈希表比较满时，有可能这个循环会一直

进行下去，这样开销将很大，为了避免这样的情况发生，规定了一个最大的循环次数，如果超过了这个最大的循环次数，还有关键字找不到自己的插入位置，本文就用新的哈希函数重新构建哈希表。

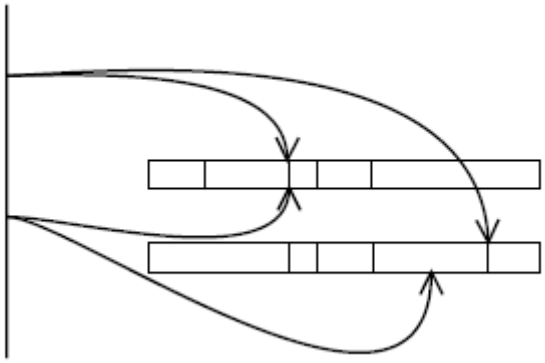

图 2−2　　Cuckoo 哈希

## 2.2　VEB 递归数据结构

前面提到的前驱数据还支持了另外两种操作：predecessor(key), successor(key)，现在设计本文的前驱数据结构，考虑如下几种设计：如果用一个无序的关键字链表来实现前驱数据结构，那么更新的开销将是常量的时间，但查询需要从表头开始一一比较，这需要 O(n)的查询时间。如果使用平衡二叉树来实现，更新和查询都需要$O(\log n)$的时间开销。

下面将描述一种使用 VEB 树的前驱数据结构，查询的开销为$O(\log w)$，均摊的更新开销为$O(\log w)$。

VEB 递归数据结构的原理是这样的：假设树的深度为 h，并且它为 2 的幂。概念上本文把树从边中层处分开，也就是在结点深度的 h/2 到 h/2+1 处分开。这样就把树分为深度为 h/2 的顶递归子树 A 和深度为 h/2 的底递归子树$B_1.....B_k$，然后对$A, B_1.....B_k$这些子树以同样的方式展开下去，直到一颗树只包含一个结点。顶递归子树 A 称作顶层递归树，$B_1.....B_k$称作底层递归树, 见图 2−3。

本文限定关键字都是整数，w表示RAM模型的机器字大小，b小于w，为2的幂。VEB树存储一个子集 S，S表示域[0，$2^b-1$]，用一种能有利于前驱查询的方法存储。那么域中每个关键字都有b位，把b位关键字的每一位映射到一棵树的路径上，这个路径从根到叶子节点代表了关键字的从高位到低位。如果用van

Emde Boas递归数据结构，意味着把[0，$2^b-1$]分为$2^{b/2}$个[0，$2^{b/2}-1$]，每一个[0，$2^{b/2}-1$]就成了一个块，每个块有有一个相同的b/2位前缀。VEB树的前驱查询的基本原理是域S[0，$2^w-1$]中的i的前驱将是j，j与i有最长的共同前缀。现在来找i的前驱，在包含i的块S中查找它的前驱，如果有这样一个前驱u′，那么i的前驱就是这个b/2的前缀加上u′，如果没找到前驱，i肯定比这个块中所有元素都要小，向顶层找到最近的一个非空块S′，提取S′中最大的元素m′， 如果最后没找到这个一个S′块，i就没有前驱了。

上面描述的程序在i的前驱在自己那个块中时是很快就完成了，但一个线性查找先前一个非空块的耗费很大。解决办法是建立一个辅助前驱结构，通过辅助前驱结构来快速找到一个非空前块，这个辅助的前驱结构存储S中元素的b/2位前缀。让i′′表示这个前缀，本文能很快找到S′，本文把前驱结构叫底层结构，辅助前驱结构叫顶层结构。

假设一个8位的关键字，前四位就是本文的顶层结构树，后面四位将构成底层结构树$B_1$……$B_k$，因为前四位共有16个不同的前缀，这将会有16颗底层结构树，即：k=16。然后考虑A，A现在代表了四位的一个整数，它又将递归划分成一个顶层结构和底层结构，这时它的底层结构树就只有4颗了。这样递归划分下去，直到最后b=1，那就是{0，1}，然后不能再细分了。VEB树能做到所有的前驱查询，所以递归也就到此为止。

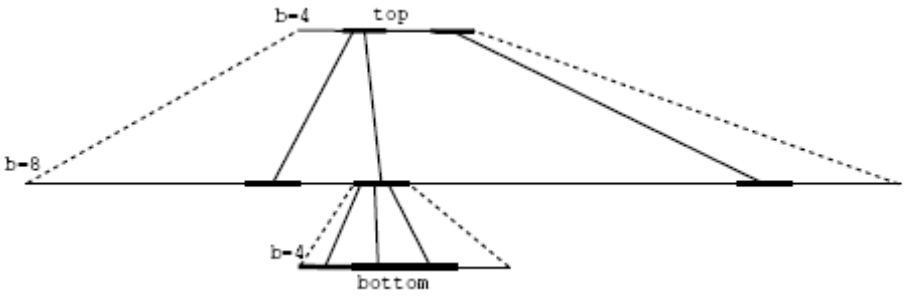


图2−3　　VEB递归数据结构

这种用VEB递归数据结构构建的树，查找时的效率是$O(\log w)$，因为整个树的路径长度为w。VEB树通过存储变量minimum 和 maximum来减少最坏的查询数。i比较i本身的块中的minimum，就能迅速知道这个块是否包含i的前驱，如果i大于minimum，只要一个底层结构树的查询，如果i小于minimum，只需要顶层结构树一次访问，把最大值找出来即可。

插入数据元素时，只需要把后缀插入到底层递归树，如果前缀已经存在在顶

层递归树中，那前缀就不需要插入了，否则也要在顶层递归树中插入前缀。删除时类似，把后缀从底层递归树中删除，如果在整个树中没有其他元素与要删除关键字的前缀相同时，也需要在顶层递归树删除前缀。

在VEB树中存储底层结构一个很明显的方法是存储在一个数组中，为所有可能的底层结构分配空间，这样既浪费空间又不利于查询。上面介绍的哈希字典正是为解决这个问题设计的，利用哈希字典来节省空间和获得查询常量时间。

# 第三章　　VEB 树的实现

根据前面一章设计的 VEB 树，关键字简单的映射到一个 VEB 树中，本文用一个双向链表来存储树的元素，这个链表体现了前驱后继关系，再用哈希字典存储关键字在链表中的地址，哈希字典是 VEB 树结构中的一部分。这个结构可以通过位映射和缓冲插入来节省空间和减少更新开销，下面将有介绍，最后是 VEB 树的具体实现。

## 3.1　效率优化

### 3.1.1　利用位映射节省空间

在第二章设计 VEB 递归树时，本文的递归中止条件是 b=1，也就是一个结点，事实上这样很浪费空间，必须为每个关键字分配一个字大小来存储。b 随着不断的递归二分，最后当域[0，$2^b$－1]的大小小于一个机器字时，可以利用机器字的每一位映射一个关键字，例如，假设机器字为 32 位，b=5 时，[0，$2^b$－1]<32，机器字的第 0 位映射 0，第一位映射 1，……，第 31 位映射 31，这样 32 个关键字本来需要 32 个字来存储，现在只需要一个字存储了。

### 3.1.2　通过缓冲插入减少更新开销

考虑在 VEB 树中的更新，一个插入肯定会在底层递归树中递归插入，也有可能要引发顶层递归树的调用，这样一个调用就引发了两个递归调用，然后四个等等。 当本文插入第一个元素到 VEB 树中就肯定会引发这种情况，因为最开始顶层递归树中是没有这个元素的前缀的。由于递归的次数最多为 $\log w$，这样更新的复杂度就是 $2+4+8+\cdots\cdots+2^{\log w}$ 约等于 $w$。

本文用一个缓冲插入的办法来减少这个更新的开销，在创建底层递归结构时偷点懒，在开始要插入到底层递归结构时，仅仅记录这个元素，而不真正插入到底层递归树里，在这里不递归下去，这一个变量叫buffer。当下一个元素要插入时，为它创建底层递归结构，插入到底层递归结构时，也把它作为底层递归树里面的buffer插入，同时也要在顶层递归树中插入一个入口，这也将成为顶层递归树的buffer。因此无论何时必须在顶层递归结构中插入，说明这个元素在相对应的底层递归结构是第一个元素，那么相应底层结构中的插入都将是一个buffer，不需要递归。因此，在每一级最多有一次插入。

对于删除，当一个删除导致两个递归调用时，肯定是底层结构的最后一个元素将要移除掉，否则不需要在顶层递归结构中删除。但用buffer可以解决这个问题，这种情况下，底层结构的删除不会递归下去，删除时仅仅删掉buffer。注意

到如果从一个结构中删除一个元素，而且这个元素刚好就是buffer这个元素的值，并且结构中还有别的值，仅仅是把其他非buffer元素移除掉，新的buffer会产生。

通过使用buffer缓冲插入使得更新的时间负责度为O(log w)，查询复杂度还是O(log w)，对空间复杂度没有影响。而且没有必要为buffer提供额外的空间，只需要让buffer保存min的值，这样min就代表buffer了。如果插入的元素比buffer要小，则buffer要用这个新的元素替代，而不是先前的buffer了。

## 3.2 具体数据结构

### 3.2.1 哈希字典的实现

下面是哈希字典的两种实现：两级完美哈希和 cuckoo 哈希。

```
typedef struct dict_data {
 int id;
 int count;
 int M;
 int s_M;
 int degeneracy_limit;
 int S;
 int k1;
 struct hashtable_elem **T;
 int *b;
 int *m;
 int *s;
 int *k;
 }dict_data;
```

在两级完美哈希中用到的哈希函数的形式是：

$$hash(x,k,s)=(((x \& INDMASK)*k) \% P) \% s$$

k 就是哈希函数的参数，k 用随机函数产生的: 1+(rand() % (P-1))

这里解释一下 S，degeneracy_limit，这两个量作用是限制第二级哈希表总大小，S 代表第二级哈希表的总大小，只要 S 超过了 degeneracy_limit，将要对整个两级哈希重新构建。

```
typedef struct dict{
 int size;
 int shift;
 int tablesize;
 int minsize,meansize;
```

```
  int maxchain;
  cell *T1;
  cell *T2;
  int a1[3];
  int a2[3];
  int id;
  int contain0;
  unsigned int val0;
}dict;
```

Cuckoo 哈希选择的哈希函数是：

```
#define hashcuckoo(h,a,shift,key) \
{\
  h = (a[0]*key)^(a[1]*key)^(a[2]*key); \
  h = (unsigned int)h >> shift; \
}
```

Cuckoo 哈希中有两个变量 minsize,meansize，在插入时，如果元素超过 meansize 会扩张成哈希表原来的两倍，删除时如果元素小于 minsize，哈希表缩小到原来的一半。这个结构中有两个变量，contain0,val0，因为 Cuckoo 哈希中关键字为并不代表真正的关键字 0，而是表示这个关键字不存在，所以 contain0 为真时，val0 就表示关键字为 0 的数据值。

这两个字典存储的都只是数据元素的指针，实际的数据元素用了一个双向链表，链表的 prep 指向数据元素的前驱，succ 指向数据元素的后继。

### 3.2.2 VEB 递归数据结构

本文用如下结构体实现了 veb 树：

```
typedef struct vebtree {
  unsigned int n;
  unsigned int w0;
  struct vebtree_i * vti;
  void * items;
}vebtree;
```

内部包含了一个哈希字典 items，方便查询，$w_0$ 代表关键字域的位数，因为 $w_0$ 都是 2 的幂，所以有可能关键字域小于[0，$2^{w0}-1$]，这没关系，相当于不使用的位都为 0。这是一个外部树结构，外部树包含了一个内部树，内部树封装了有一颗真正的 VEB 树：

```
typedef struct vebtree_i {
  unsigned int size;
  unsigned int max;
  unsigned int min;
  int id;
  struct vebtree_i * top;
  void * bottom;
  void * b2;
  unsigned short b2_sz;
  unsigned short b2_head;
  int * b2_aux;
  unsigned int bitmap;
}vebtree_i;
```

在这个结构中，min 相当于 buffer，用来缓冲插入。bottom 的作用只是存储了底层递归树的结构，比如底层递归树没有元素，字典里会有一个元素 VEB_NOMEMBERS，这个字典是以关键字的前一半高位作为关键字来存储的。以 8 位为例子，11010110，1101 为关键字，这个 1101 是要放到 top 结构中去的，它是底层结构的一个索引。如果底层递归树有一个 buffer，说明底层结构树还没有建立一颗树，只是用缓冲插入存储了一个 min 值，这就是 buffer，那么 bottom 字典里会存上 VEB_BUFFER。最后一种情况是底层结构树已经建立好了，bottom 字典里会存 VEB_REALTREE，这时 b2 已经不是空指针了，指向了一颗 vebtree_i。

veb 树的程序实现是在 32 位机器上，如果域也为［0，$2^{32}-1$］，那么 b2 数组最多有 $2^{16}$ 个，b2_sz 只是记录已经到的底层递归树个数。刚开始时，b2_head=0，当第一个元素插入到底层递归树时，用 b2[0]表示这颗树，这时 b2_head 要加 1，意味着下一个元素如果要新建一颗底层递归树时，用 b2[b2_head]来指向它。bitmap 用于节省空间，可以较早结束递归，在域的位数小于 MICROSIZE，时就结束递归，而不是到达域［0，1］。

## 3.3 对数据的各种操作

### 3.3.1 查询

哈希字典的查询很简单，两级完美哈希的查询中用两次哈希函数即可。

j = hash(a, dict->k1, dict->s_M);

i = hash(a, dict->k[j], dict->s[j]);

Cuckoo 哈希中，如果在第一哈希表中查询不到则需要在第二个哈希表再查

询一次。

VEB 树的查询分为前驱查询和后继查询，两者的原理是一样的。查询开销为 $O(\log w)$ 。

查询关键字 i 的前驱的主要流程见图 3-1：

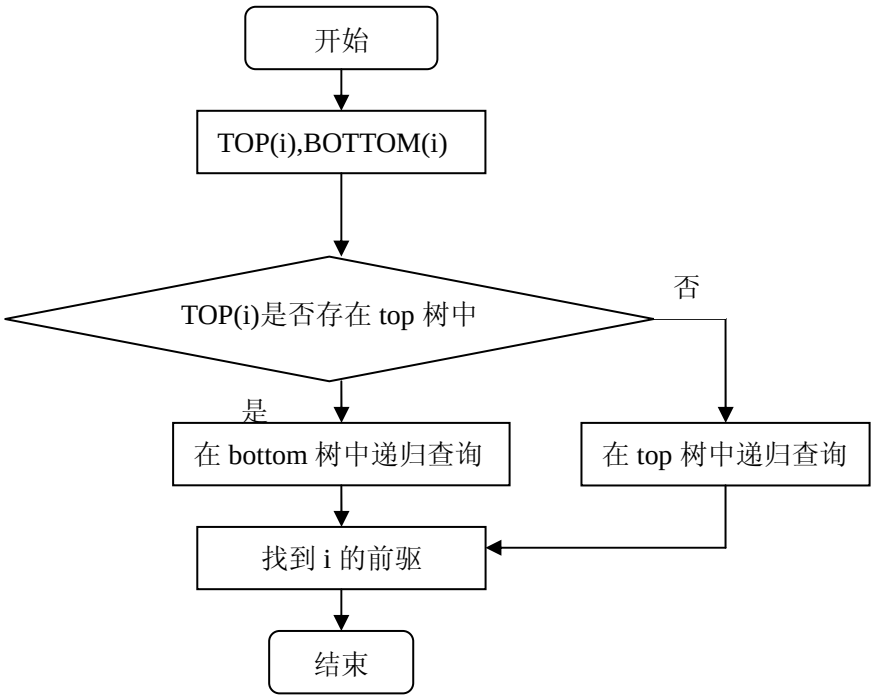


图 3-1　　VEB 树前驱查询流程图

查询一个给定关键字 i 前驱时，开始会做一些特殊处理，树为空时，返回 VEB_TREE_EMPTY，如果 i 大于树的 max，前驱就为 max，如果 i 小于树的 min，说明没有前驱，返回 VEB_NO_PREDECESSOR。当 vt->min <= i < vt->max 时，意味着树中存在 i 的前驱和后继，将要在树中查询。用 TOP(i)获得 i 的高半位，以 TOP(i)作为关键字在哈希字典中查询，查询得到的结果有三种情况，一种是查不到东西，说明这个块还没有数据；一种是得到一个 VEB_BUFFER 值，它是这个块中的最小值，如果 VEB_BUFFER<BOTTOM(i)，把这个 VEB_BUFFER 和 TOP(i)连接起来即为要找的前驱；最后一种情况是 TOP(i)对应这颗底层树已经建立起来，根据查询的结果得到 vt->b2 数组的那棵底层结构树，记为 botq，然后在 botq 中递归查询 BOTTOM(i)的前驱，根据 botq 中查到的前驱，再与 TOP(i)连接起来，就是关键字 i 的前驱。这三种情况都有可能找不到前驱，这说明前驱不在 TOP(i)对应的这个块中，需要找到一个前驱块，这必须在 vt->top 树中查找，找到这个前驱块，取出 max 值即可。在递归查询中，当表示关键字域的位数小于等于 MICROSIZE，就用位映射查询。

查询给定关键字的后继与查询前驱原理相同，这里不再详叙了。

### 3.3.2 插入

首先来分析一下在哈希字典中的插入，分为两部分，一部分为两级完美哈希字典的插入，另一部分为 Cuckoo 哈希字典中的插入。

对于两级完美哈希字典中的插入，通过：

j = hash(a, dict->k1, dict->s_M);i = hash(a, dict->k[j], dict->s[j])

可以迅速找到关键字 a 需要插入的位置 T[j][i]。如果这个位置中关键字已经存在，就更新一下元素状态与数据即可。检测到关键字不存在，整个哈希表的更新数 dict->count 需要增加 1，这时需要检查是否超出哈希字典的最大容量，如果更新次数超出容量整个哈希字典都要重哈希，这包括第一级和第二级哈希表，下一段落将会介绍重哈希。如果哈希表更新计数器 dict->count 不超过哈希字典容量且关键字为一个新关键字，那么关键字将插入到 T[j][i]，也就是插入到第 j 个哈希桶的第 i 个位置。这里又分两种情况，一种是插入这个元素后，元素数量没超过第 j 个关键字“桶”的容量，一种是超过了最大容量。对于第一种情况，如果 T[j][i]这个位置被另外一个关键字占据了，说明在第 j 个关键字“桶”发生了冲突，必须为第 j 个关键字“桶”重新选择哈希函数，这里会调用一个函数 int select_hash _function(int s, int b, struct 哈希 table_elem * Lj)，这个函数实现了从域[0,b]到域[0,s] 为给出的数组 L[j](大小为 b)选择新的完美哈希函数。对于后一种情况，需要对第 j 个关键字“桶”的容量 B[j]加倍。对第 j 个关键字“桶”的最大容量加倍后，对整个哈希字典的 sum(s_j)会增加，这时对 sum(s_j)与 degeneracy_limit 做个比较，超过了 degeneracy_limit，对整个哈希字典重哈希。否则为第 j 个关键字“桶”选择新的哈希函数，这时最大容量 B[j]已经是原来的两倍了，哈希表大小将是 2B[j]*(B[j]-1)。

现在介绍一下整个两级哈希的重哈希。重哈希主要对第一级哈希表的最大更新次数 M 更改为当前数据元素数量的(1+C)倍（C=0.5），哈希表大小更改为 S_FACTOR*M，S_FACTOR=8.0*sqrt(30)/15.0，在第四章将会分析这个问题。相应的第二级哈希表最大更新次数 m[j]更改为第二级哈希表每个表元素数量的两倍，表大小改为 2*m*(m-1),然后把所有的元素重新分散到新的哈希表中去。

对于 Cuckoo 哈希中的插入就要简单得多了。如果关键字为 0，将用到结构的 contain0,val0 做特殊处理。同样验证关键字是否已经存在于两个哈希表中，如果存在，更新数据即可；如果不存在，将会陷入一个找空位的循环之中，循环的最大次数已经在结构中用 maxchain 定义。最坏的一种情况是在循环 maxchain 次后，有一个关键字仍然没位置插入，需要重哈希，如果字典的大小小于 meansize,仍然用原来的表大小重哈希：rehash(D,D->tablesize)：否则将用两倍于原来表大小重哈希：rehash(D,2*D->tablesize)。重哈希后，再重新调用插入函数。

接下来从整个 veb 树来看看插入的实现。主体流程图见图 3-2：

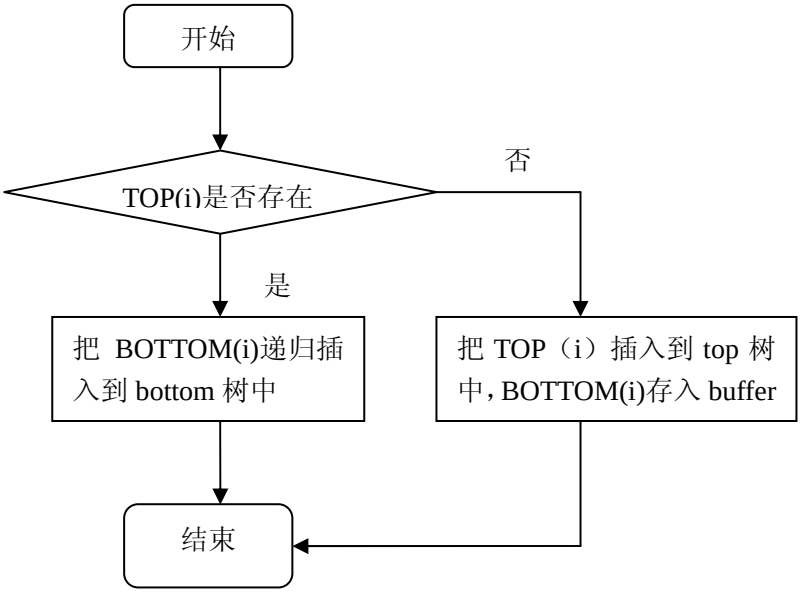


图 3-2　　VEB 树插入算法流程图

如果插入的是第一个元素，那么这时只是创建了一颗外部 veb 树，所以在插入时要检测一下内部树是否为空，若为空需要在插入算法里调用创建内部树的函数。内部树的创建主要把 vebtree_i 的各个变量分配值，并把一个 buffer 赋值给 min 和 max，这就用到了缓冲插入，只是用 min 纪录一下。如果内部树已经创建好了，将把值插入到一个内部树中，这里调用了一个内部插入的函数。内部插入函数才是在 veb 树中插入的主要实现，在插入值到内部树之前，需要更新树的 min 和 max 值，有一种情况就是插入的 i 小于 min，那么实际插入的值就不是 i，而是原来的 min 值。把要插入的 i 分为两部分 TOP(i),BOTTOM(i)，对 TOP(i)作为哈希字典的关键字进行哈希查询，得到一个结果 c。c 可能有三个结果，一个是为 0，这表示共同前缀为 TOP(i)的这颗顶层结构树还没建立，需要建立，a 为 buffer，同时把 VEB_BUFFER 的信息插入到哈希字典中；第二个可能是 VEB_BUFFER，这意味着树中已经有一个元素作为 buffer，这时再插入一个元素，就必须建立一颗树了，调用函数创建一颗底层结构树，然后在哈希字典中插入一个以 TOP(i)为关键字，VEB_REALTREE 与刚建立的底层结构树指针为数据的元素；第三个结果就是 VEB_REALTREE，这时 c 的下半部分已经包含了对应树的指针，这时递归调用插入函数，把 BOTTOM(i)插入到底层结构树中去。实际上 c 一个字分为两部分，前半个字指示树的状态，后半个字存一些辅助信息。内部插入函数有递归调用，当表示域的关键字位数小于 MICROSIZE 时，递归会结束，这里用的是位映射技术。内部插入函数的功能就是上面所分析的，在内部插入函数返回后，把要插入的 i 放到双向链表的适当位置，这里要调用找前驱的函数，

找到 i 的前驱，然后更新双向链表指针。

### 3.3.3 删除

删除算法也分两个方面：哈希字典的删除和 VEB 树的删除。两级完美哈希中的删除很简单，利用 j = hash(a, dict->k1, dict->s_M)；i = hash(a, dict->k[j], dict->s[j]);找到数据元素后，置状态为 HTE_DELETED。这里仍需要检查一下操作数 dict->count 是否大于 dict->M,超过了就要哈希。Cuckoo 哈希中的删除要把相应关键字置 0 就行，当字典大小小于 minsize 时，调用 rehash(dict,dict->tablesize/2)重哈希。

下面分析在 VEB 树中的删除。程序流程图见图 3-3：

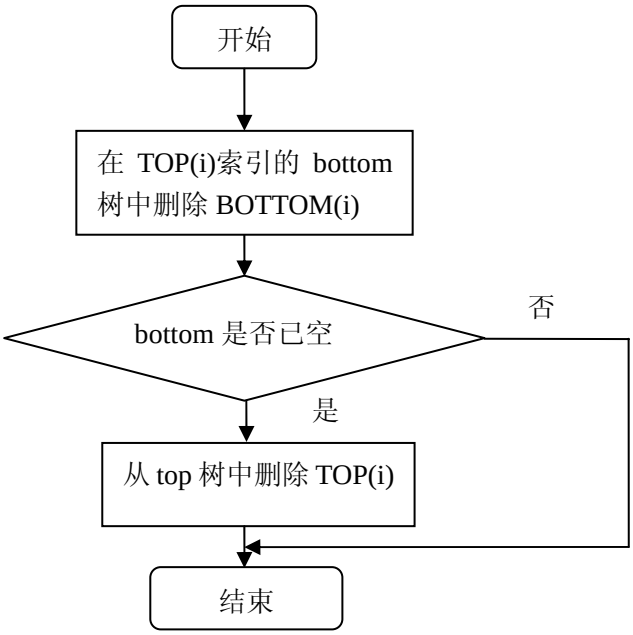


图 3-3　　VEB 树删除算法流程图

VEB 树在进入内部树删除前，将查询字典中元素是否存在或者树是否为空。进入内部树的删除函数，将检测当前域的位数是否小于 MICROSIZE，若小于，则置表示元素的映射位为 0 就完成删除了。

如果要删除的元素正好是 min，要从余下的元素中找出新的 min 值。Top->min 赋值给 a，然后以 a 为关键字作哈希查询得到 c，BLOCKTYPE(c)将得到两个结果要么是 VEB_BUFFER，要么是 VEB_REALTREE，而不可能为 0。当 BLOCKTYPE(c)是 VEB_BUFFER 时，表示 a 对应的块中已经没有元素，这会在 Top 树中引发一个递归删除，把 a 删除掉，并更新 min 值为 a 与 BLOCKDATA(c)的联合。当 BLOCKTYPE(c)是 VEB_REALTREE 时，BLOCKDATA(c)存有相应底层递归树的指针 botq，更新 min 为 a 和 botq->min 的联合。然后要在底层结构树中递归删除当前新 min 的底半部分 BOTTOM(min)。

当删除元素是 max 时，大致流程与删除 min 差不多，要是其他元素就不要更新 min 或 max 值。在删除时，可能导致一些指针要释放，这时一定要用 free 函数释放掉。

在内部树结束删除后，和插入一样，要在双向链表中修改指针，删掉一个数据元素。

# 第四章　　性能分析

## 4.1　VEB 树的性能

根据前面两章的分析已经很明显的得出了 VEB 树的性能：前驱后继查询需要$O(\log w)$的时间，通过缓冲插入把更新的时间开销从 O($w$)降到$O(\log w)$。

## 4.2　哈希字典的性能

两级完美哈希字典查询的负责度为 O(1)，如果插入和删除引发哈希字典重构这是哈希字典的最主要开销。一种解决方法是在一定的更新操作次数$\Omega(n)$后重建整个哈希字典，那么重建哈希字典的时间开销将分摊到所有更新操作上。两级完美哈希的哈希桶大小要满足如下不等式：

$$\sum_{j=0}^{s-1}\frac{B_j(B_j-1)}{2}<\frac{2n(n-1)}{s}\ldots\ldots\ldots(4\text{-}1)$$

在重建哈希字典时，让第二级哈希表的大小大约为原来实际大小的 4 倍，当插入一个关键字到哈希表中产生冲突时，如果发现哈希桶的大小仍然小于原来大小的 2 倍，意味着第二级哈希表的数据元素仍然比较稀松，对冲突的处理方法只是重新选择第二级哈希函数。当哈希表变得比较紧时，就把哈希表大小扩大四倍，然后找一个新的第二级哈希函数。第二级哈希表扩大四倍后，发现过大，将引发一个整个哈希字典的重建。

接下来看看两级完美哈希是如何控制第二级重函数与扩大四倍这个方法的。选择两个常量 A，B，重哈希时假设有 n 个元素，第一级哈希表有 An 个元素，Bn 表示所有第二级哈希表长度之和。选择一个 c，在[0，1]之间，设计时用的是 0.5，整个哈希字典重建将会在如下两种情况下发生，一是在 cn 个更新操作后，一是在第二级哈希表长度之和超出了 Bn，假设某个哈希桶的大小为$B_j^{`}$，在下一次哈希表扩大或重建哈希字典之前，哈希桶的大小不会超过$2B_j^{`}$，那么第二级哈希表的总大小是$\sum_j 4B_j^{`}(2B_j^{`}-1)$。有:

$$\begin{aligned}&\sum_j 4B_j^{`}(2B_j^{`}-1)\le 16\sum_j\frac{B_j^{`}(B_j^{`}-1)}{2}+8\sum_j B_j^{`}\\&\le 32\frac{2(n+cn)(n+cn-1)}{An}+8(n+cn)\\&\le(32(1+c)^2/A+8(1+c))n\end{aligned}$$

让 B=$32(1+c)^2/A+8(1+c)$，取 A=(1+c)$8\sqrt{30}$/15。在 cn 次更新操作后将会

有一半的几率不会重建整个哈希字典。

Cuckoo 哈希字典中,m 个元素映射到 Cuckoo 的两个哈希表中就得到 2m 个哈希值，因此让表维持在大约半空的状态，这样在关键字找空位时就有 1/2 的几率找到，只要 O(1)的时间。

## 4.3 实验结果

为了与 VEB 树的性能进行比较，本人实现了一个有序双向链表，下面是对两个重要操作的一些数据比较。

在插入操作上的实验结果见图 4-1。

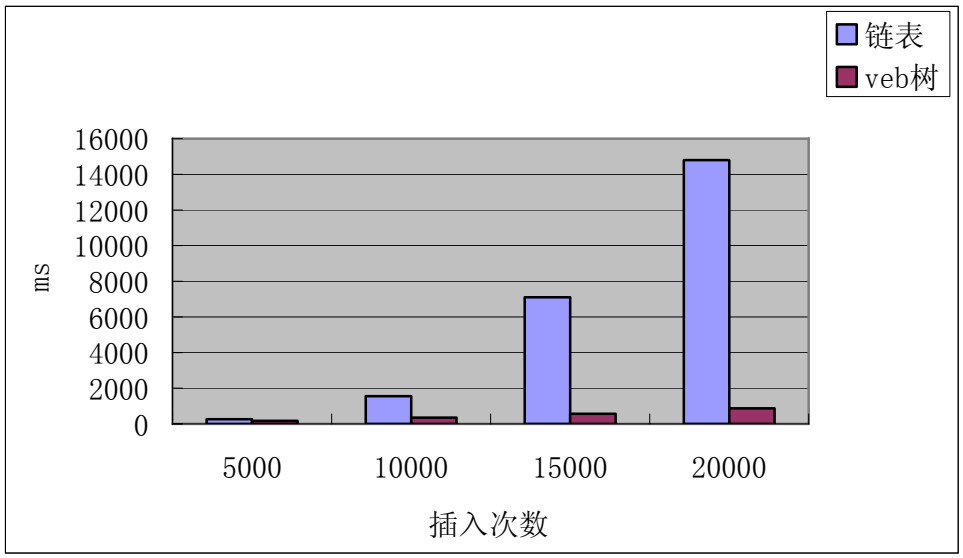


图 4-1　　插入操作的比较

实验中分别选取的插入次数为 5000，1000，15000，20000。在插入操作中，VEB 树的时间开销明显比链表次数少很多，在插入次数较少的情况下，VEB 树的时间复杂度和链表相比优势并不是很大，当插入次数成倍增加时，VEB 树的时间损耗并不增大多少，而链表的时间损耗直线增长。链表的插入时间复杂度与表长成正比，而 VEB 树的插入时间复杂度与已经插入的元素数量没有联系，只跟关键字域有关，为 $O(\log w)$ 。

查找前驱的实验结果见图 4-2：

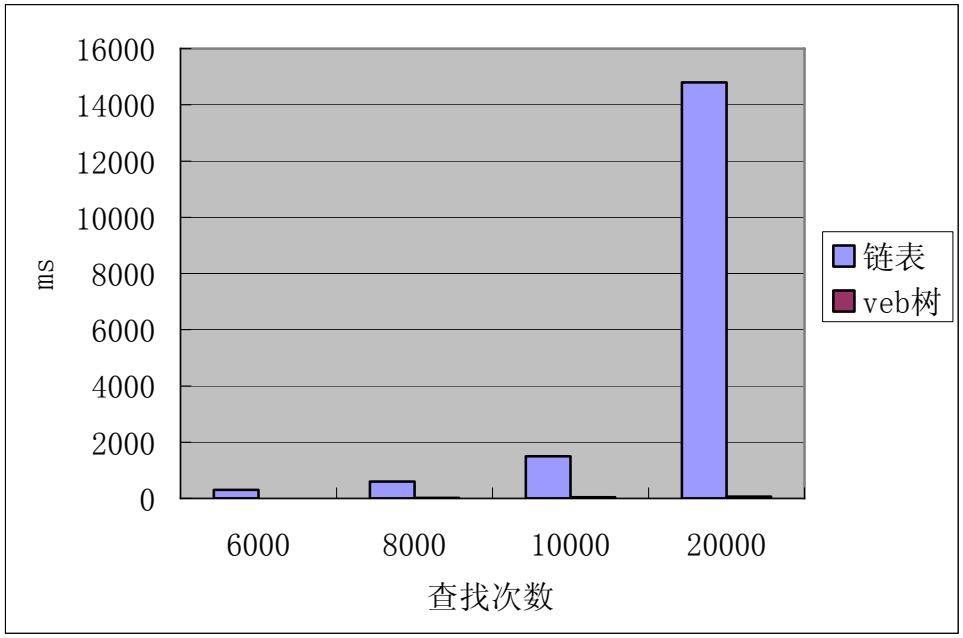


图 4-2　　前驱查找

选取查询的次数分别为 6000，8000，10000，20000，链表的查询复杂度跟表长有关，可以看出链表的前驱查询和插入时间复杂度一样，VEB 前驱查询和插入的时间复杂度也一样，都为 $O(\log w)$ 。

# 第五章　　结论

本文详细设计和实现了 VEB 树，VEB 树的成员关系查询时间复杂度为 O(1)，前驱查询的时间为 O(log *w*)，更新的时间负责度为 O(log *w*)。VEB 树应用到高效的数据集系统中会加速查询和更新。不过 VEB 树的一个缺陷就是限定了关键字为整数，并且关键字域跟机器字位数有关，关键字域范围为[0，$2^w-1$]。

VEB 树中采用 van Emde Boas 提出的递归数据结构，树已经不是原来意义上从跟节点到叶子节点的祖先和子孙关系，而是在整个树层面上分 top 树和 bottom 树，这种创新思想可以应用到很多结构中，这种递归布局应用到高速缓存参数无关树上的设计有重要作用[9]。

VEB 树的实现是用 C 语言实现，在 VC6.0 环境下测试通过，由于程序涉及指针操作比较多，故错误隐藏很深，带来调试困难，需要一步步跟踪执行。

# 参考文献


[1]Paul Beame and Faith Fich, Optimal bounds for the predecessor problem and related Problems[Z].New York: ACM Press,1999.

[2]Brodal G, Fagerberg R and Jacob R, Cache oblivious search trees via binary trees of small height[R]. New York:Proc. ACM-SIAM Symp. on Discrete Algorithms, 2002.

[3]Cormen T H, Leiserson C E, Rivest R L,et al. Introduction to algorithms[M]. MIT Press, 2001.

[4]Dietzfelbinger M, Karlin A R, Mehlhorn K, et al. Dynamic perfect hashing: Upper and lower bounds[J]. IEEE Symposium on Foundations of Computer Science, 1988, 5: 524—531.

[5]Michael L. Fredman, J´anos Koml´os, and Endre Szemer´edi, Storing a sparse table with O(1) worst case access time[J], Journal of the Association for Computing Machinery 31 1984, 31( 3): 538—544..

[6]Motwani R, Raghavan P. Randomized algorithms[M], Oxford: Cambridge University Press, 1995,320—362. [7] Rasmus Pagh and Flemming Friche Rodler, Cuckoo hashing[J]. Lecture Notes in Computer Science , 2001, 2161:121—135.

[7]Overmars M H. The Design of Dynamic Data Structures[M], New York.:Springer-Verlag, 1983,345—361.

[8]P. van Emde Boas, Kaas R, and Zijlstra E. Design and implementation of an efficient priority queue[J], Mathematical Systems Theory, 1977, 10:99—127.

[9]Bender M A, Erik Demaine D, Cache-Oblivious B-Trees[Z]. National Institute of Standards and Technology: Gaitherburg, 2000.

[10]严蔚敏，吴伟民，数据结构（C语言版）[M].北京：清华大学出版社，2004，251—262.

[11]Dietz P F, Sleator D D. Two algorithms for maintaining order in a list[R]. New York: Theory of Computing, 1987.

[12]Mehlhorn K.. Data Structures and Algorithms[M]. Sorting and Searching, theorem 1984, 198—199.

[14]Willard D E. Good worst-case algorithms for inserting and deleting records in dense sequential files[Z]. Washington, D.C: In Proc. 1986 ACM SIGMOD Internat. Conf. Management of Data, 1986.

# 致　　谢

首先，我要感谢我的导师张坤龙副教授，感谢他对我毕业设计的多方面指导与帮助。从他那里我学到了治学的严谨作风，分析问题的全面性，踏踏实实的攻克每一个问题，以及坚持不懈的学习新知识。我将在以后的学习道路上以导师为榜样，不断努力。

同时，我还要感谢同组的成员：包武，吴宗远，王乐。感谢他们一起对本毕设题目提出的建议与帮助，通过小组的努力才有本论文的顺利完成，再次感谢他们。