DIRECT PERFECT HASHING FUNCTIONS FOR EXTERNAL FILES

By

Yuichi Bannai

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

1988

ABSTRACT

DIRECT PERFECT HASHING FUNCTIONS

FOR EXTERNAL FILES

By

Yuichi Bannai

A composite perfect hashing scheme for large external files which guarantees single disk access retrieval has been proposed in [RL88]. It was suggested that direct perfect hashing functions be found by trial and error methods. In this thesis, we explore systematic methods of finding direct perfect hashing functions. Extensions of Sprugnoli's Quotient Reduction and Remainder Reduction methods are proposed. The experimental results indicate that these methods are practical. Also, the performance of different $universal_2$ classes of hashing functions are studied.

To my loving wife, *Namiko*

ACKNOWLEDGEMENTS

I wish to express my thanks to my thesis advisor M.V. Ramakrishna for his guidance and encouragement. The other members of my thesis committee S. Pramanik and M. Chung provided valuable comments. My thanks are due to E. Vincent who carefully read this thesis and made many suggestions for improving readability. I would like to thank my wife Namiko for her patience and love that made it possible for me to complete this thesis. Finally, I would like to thank my company Canon Inc. for their constant support during my stay in the United States.

TABLE OF CONTENTS

1. Introduction	1
2. Background	3
3. External Perfect Hashing Scheme	5
4. Quotient Reduction Method	7
5. Remainder Reduction Method	23
6. Universal Classes of Hashing Functions	28
7. Dynamic Behavior	36
8. Conclusions and Future Work	43
Appendix	44
Bibliography	46

LIST OF TABLES

Table 4.1	Hash table of Example 4.2	xvii
Table 4.2	Average load factor of nine groups of file (A)	xx
Table 4.3	Load factor for file (B)	xxi
Table 5.1	Probability of having no collisions for a given M and n	xxiv
Table 5.2	Average load factor of nine groups of file (A) obtained using RR	xxvi
Table 5.3	Load factor for file (B) obtained using RR	xxvi
Table 6.1	A hash table (example of failure)	xxx
Table 6.2	A hash table (example of success)	xxx
Table 7.1	Probability of an insertion causing rehashing using RR	xxxvii
Table 7.2	Probability of an insertion causing rehashing (n=250)	xxxvii
Table 7.3	Comparison of probability of rehashing	xxxviii

LIST OF FIGURES

Figure 4.1	Admissible increments when $b = 2$	viii
Figure 4.2	Set of keys for Example 4.1	ix
Figure 4.3	Solution space	xiii
Figure 4.4	Solution space for Example 4.2	xvi
Figure 6.1.1	Observed and computed probability of a trial succeeding	xxxii
Figure 6.1.2	Observed and computed probability of a trial succeeding	xxxiii
Figure 6.2.2	Probability of a randomly chosen function being perfect ($b = 10$)	xxxiv
Figure 6.2.2	Probability of a randomly chosen function being perfect ($b = 20$)	xxxiv
Figure 7.1.1	Load factor of a group ($b = 10$)	xl
Figure 7.1.2	Load factor of a group ($b = 40$)	xl
Figure 7.2.1	Overall load factor ($b = 10$)	xli
Figure 7.2.2	Overall load factor ($b = 40$)	xlii

1. Introduction

Hashing is an important and widely used technique for organizing files. The storage area is partitioned into a finite number of buckets. Hashing involves the computation of the address of the data item directly by evaluating a function (hashing function) on the search-key of the desired record. The record is then stored in the corresponding bucket. If the bucket is already full, we say an 'overflow' has occurred. A hashing function is said to be 'perfect' if no overflows occur for the given data. There is no need to handle overflows in a perfect hashing scheme, and hence we obtain ideal retrieval performance.

A composite perfect hashing scheme for large external files was proposed in [RL88]. An ordinary hashing function is used to divide records of the file into a number of groups. The records in each group are stored using direct perfect hashing functions. A simple trial and error method for finding perfect hashing functions is proposed and analyzed. In this scheme, any record can be retrieved in a single disk access, and the analysis shows that records can be inserted and deleted at an average cost comparable to that of many traditional hashing schemes. One drawback of this scheme is that there is no guarantee of finding a direct perfect hashing function for a given set of keys, which motivated this thesis. The proposed methods here are an extension of Sprugnoli's Quotient Reduction and Remainder Reduction methods [SP77].

In [RL88], Ramakrishna *et al.* experimentally showed that *universal*$_2$ class H_1 of hashing functions could be used for making trials: the relative frequency of perfect hashing functions within *universal*$_2$ class H_1 is the same as that predicted by the analysis for the set of all functions. However, they did not study other *universal*$_2$ functions: classes H_2 and H_3 of hashing functions. In this thesis, we examine if classes H_2 and H_3 behave similar to the class H_1.

Outline of the Thesis

The next section presents the background of research into perfect hashing. The external perfect hashing scheme proposed in [RL88] is introduced in section 3. In section 4, the Quotient

Reduction method for generating perfect hashing functions is introduced and analyzed. The Remainder Reduction method, which is superior to the Quotient Reduction method, is presented in section 5. The performance of *universal*$_2$ classes H_2 and H_3 functions are studied in section 6. Section 7 discusses the dynamic behavior of the proposed scheme.

Definitions and Assumptions

A hash table consists of m buckets, and each bucket has a capacity of b records. A set $I = \{x_1, x_2, ..., x_n\}$ of n search keys are to be stored in the table. A hashing function $h : I \rightarrow [\, 0 \,..\, m-1 \,]$ assigns an integer 0 through $m-1$ to each key. We will consider functions of integer keys in this paper. A hashing function h is a **perfect hashing** function for a given key set and table if no bucket receives more than b keys under the function h. We use $[p \,..\, q\,]$ to denote the interval $[p, p+1, p+2, ..., q-1, q\,]$ $(p < q)$, the length of which is $q-p+1$. The load factor (α) is the ratio of the number of keys to the total capacity of the hash table, $\alpha = n/mb$.

2. Background

The perfect hashing schemes dealt with in the literature may be grouped into two classes [RL88]:

a) Direct perfect hashing;

b) Composite perfect hashing.

Direct perfect hashing functions map a given key into an address where the key and its associated record are stored. The composite perfect hashing scheme involves additional table look-up to find the desired address. This means that a retrieval involves more than one level of access.

Direct Perfect Hashing

Sprugnoli, who was the first to formally define perfect hashing, proposed a systematic method of finding perfect hashing functions, called the **Quotient Reduction method** [SP77]. The functions are of the form $h(x) = \lfloor (x+s)/N \rfloor$, where s and N are constants, and are called the **admissible increment** and **quotient**, respectively. He gave a constructive proof of the existence of the Quotient Reduction perfect hashing function for any given key set. His algorithm finds s and N for a given set of keys $I = \{x_1, x_2, ..., x_n\}$. However, his algorithm was restricted to the special case of the bucket size being one. He also proposed the **Remainder Reduction** method, which yields better storage utilization especially when the keys are not uniformly distributed. The functions are of the form $h(x) = \lfloor ((xq+d) \bmod m) / N \rfloor$.

Jaeschke [JS81] proposed the **Reciprocal Hashing** function of the form $h(x) = \lfloor C/(Dx+E) \rfloor \bmod n$, where C, D, and E are constants. He proved that there always exists a perfect hashing function of this form for any given I. Chang [CH84] proposed a similar scheme based on the Chinese Remainder Theorem. However, these schemes are not of practical value. Please refer to [RM86, MR84] for further details.

Composite Perfect Hashing

Composite hashing is necessary to handle for large sets of data. Sprugnoli suggested the idea of segmentation to handle large sets of keys [SP77]. The keys are divided into a number of groups by an ordinary hashing function so that each group can be handled by direct perfect hashing. Fredman *et al.* [FR82] constructed perfect hash tables using Sprugnoli's idea. Their scheme guarantees constant time retrieval with $O(n)$ space requirement. However, they did not discuss how to update the table. Cormack *et al.* [CR85] proposed a similar scheme including algorithms for insertion and deletion.

3. External Perfect Hashing Scheme

The external perfect hashing scheme proposed by Ramakrishna *et al.* [RL88] attempts to achieve single disk access retrieval at the cost of a small header table in internal memory. They used a composite data structure similar to that proposed by Cormack *et al.* [CR85]. Direct perfect hashing functions are found by a simple trial and error method. One drawback of this method is that there is no guarantee that perfect hashing functions can always be found for any given key set although the expected number of trials is small and can be controlled. We investigate systematic methods for finding direct perfect hashing functions for external files, and study properties of the file constructed by the proposed method such as the storage utilization and frequency of rehashing.

The details of the external perfect hashing scheme are described below. The set of keys is divided into several groups. The keys in each group are stored in a number of contiguous pages in the secondary memory using a perfect hashing function. An ordinary hashing function H is used to accomplish the grouping. Let key group t denote the set of keys $\{x \mid x \in$ the given key set and $t = H(x)\}$. Each entry of the header table in the internal memory is of the form (p_t, m_t, R_t), where (p_t) denotes pointers to starting addresses of each group, (m_t) is the group size, and (R_t) denotes the parameters of the perfect hashing function for group t. Any record can be retrieved in a single access of the secondary memory in this scheme. Algorithms for retrieval, insertion, and deletion are given below.

Retrieval of a record with key x:
- Compute the group t to which x belongs by $t := H(x)$.
- Extract $<p_t, m_t, R_t>$ from entry t of the header table.
- Compute the page address of x by $A_x := p_t + h(x, R_t)$.
- Read in page A_x.
- Search page A_x for key x. If key x is not found, the search has failed.

Insertion of a record with key x:
- Compute A_x as above and read in page A_x .
- If the page is not full then
  - Insert the record into page A_x, and write back the page,

  else { **Rehashing** }
  - Read in all pages of the group t. The number of pages is m_t.

- Find a new perfect hashing function for all the keys in the group, including the new key x. The number of pages obtained by the new perfect hashing function, m_u, may or may not be equal to m_t, and let R_u be the parameter of the new perfect hashing function.
- Find an address p_u in the secondary storage having space for m_u contiguous pages (p_u may be equal to p_t).
- Restore the records on pages p_u, p_u+1, p_u+2,.....,p_u+m_u-1 using the new perfect hashing function.
- Update the header table entry of group t to $< p_u, m_u, R_u >$.

Deletion of a record with key x:

- Compute A_u as above and read in the page.
- Search page A_u for key x.
- If key x is found then delete the record and write back the page,
  else the desired record is not in the file.

It may be necessary to rehash the group to avoid low storage utilization. The procedure of rehashing is similar to that for insertion.

4. Quotient Reduction Method

Quotient Reduction hashing functions are of the form $h(x) = \lfloor (x+s)/N \rfloor$, where s and N are constants. We will show that functions of this form, which are perfect, exist for any given bucket size, by giving a constructive proof.

The basic idea of this method is to divide the given set of keys into a number of intervals whose length is N, and to move the boundaries between the intervals by adding the admissible increment s so that each interval, corresponding to a bucket, contains no more than b keys. The following is an extension, of Sprugnoli's construction, to the general case $b > 1$.

Admissible increments

Let $I = \{x_1, x_2, \ldots, x_n\}$ be the set of keys to be hashed, in sorted order. When the bucket size is b, keys x_i and x_{i+b} must not fall in the same bucket. A set of admissible increments J_i of the key x_i for a given N is a set of integers for which the condition

$$\left\lfloor (x_i+t_i)/N \right\rfloor < \left\lfloor (x_{i+b}+t_i)/N \right\rfloor \qquad 1 \le i \le n-b,\ t_i \in J_i$$

holds. In other words, an admissible increment for x_i is any translation value which adjusts x_i and x_{i+b} to two different intervals $[(p-1)N \,..\, pN-1]$ and $[(q-1)N \,..\, qN-1]$, where p and q are integers, and $p<q$. There are $n-b$ sets of admissible increments for a given I and b.

Proposition 1

There exists a quotient reduction perfect hashing function for a given I, b, and N if and only if the set of admissible increments J_i, $1 \le i \le n-b$, have at least one element in common: *i.e.*, $\bigcap_{i=1}^{n-b} J_i \neq \Phi$.

Proof

The "only if" part is obvious. The "if" part can be proved as follows.

The condition $h(x_i)<h(x_{i+b})$ must hold for every x_i, $1 \le i \le n-b$, for h to be a perfect hashing function. Hence,

$$\left\lfloor (x_i+t_i)/N \right\rfloor < \left\lfloor (x_{i+b}+t_i)/N \right\rfloor \qquad \text{for } \textit{every } x_i,\ 1 \le i \le n-b \tag{4.1}$$

Let $k=\left\lfloor (x_{i+b}+t_i)/N \right\rfloor$ and $\delta_i = x_{i+1} - x_i$. It follows that $x_i + t_i < kN$ and $x_{i+b} + t_i \ge kN$. The set of all admissible increments J_i of x_i is given by

$$J_i = \{\ t_i \mid kN - x_{i+b} \le t_i \le kN - x_i\ \}. \tag{4.2}$$

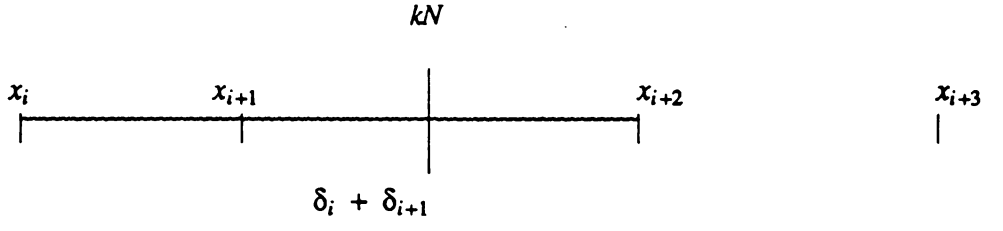
Let $u_i = t_i + x_{i+b} - kN$. J_i can be represented in terms of δ_i as follows.

$$J_i = \{\ t_i \mid t_i = u_i - x_{i+b} + kN\ ,\ 0 \le u_i < \sum_{j=i}^{i+b-1} \delta_j\ \}. \tag{4.3}$$

In fact, $x_i + t_i = x_i + u_i - x_{i+b} + kN = u_i - \Sigma\delta_j + kN$, and

$x_{i+b} + t_i = x_{i+b} + u_i - x_{i+b} + kN = u_i + kN$. Thus, two keys x_{i+b} and x_i fall in two different intervals $[(p-1)N\ ..\ pN-1]$ and $[(q-1)N\ ..\ qN-1]$, $(p < k,\ k \le p)$, if and only if $0 \le u < \sum_{j=i}^{i+b-1} \delta_j$.

Figure 4.1 shows the admissible increments for x_i with $b = 2$.



The keys can be moved to the left by $x_{i+2} - kN$, and to the right by $kN - x_i$.
The range of the admissible increments is given by

$$x_{i+2} - kN + kN - x_i = x_{i+2} - x_i = \delta_i + \delta_{i+1}.$$

Figure 4.1 Admissible increments when $b = 2$.

Taking *modulo N*, the reduced admissible increments for x_i is given by

$$J_i = \{\, t_i \mid t_i = u_i - x_{i+b} \bmod N \mid 0 \le u_i < \sum_{j=i}^{i+b-1} \delta_j \}. \tag{4.4}$$

Each J_i is a subset of integers $[0 .. N-1]$. For each interval $[x_i .. x_{i+b}]$, $1 \le i \le n-b$, the set of admissible increments, J_i is computed by (4.4). If an element s common to all J_i, $1 \le i \le n-b$ exists, then the relation

$$\left\lfloor (x_i+s)/N \right\rfloor < \left\lfloor (x_{i+b}+s)/N \right\rfloor$$

holds for every x_i, $1 \le i \le n-b$. The corresponding s and N define a perfect hashing function $h(x_i) = \left\lfloor (x_i+s)/N \right\rfloor$. Thus, there exists a quotient reduction perfect hashing function if and only if $\bigcap_{i=1}^{n-b} J_i \neq \Phi$. □

In order to find admissible increments for a given I, N, and b, we compute J_i, $1 \le i \le n-b$, and take their intersection. If the length of the interval between x_{i+b} and x_i is equal to or greater than the quotient N, $\sum_{j=i}^{i+b-1} \delta_j \ge N$, the set of admissible increments is simply $[0 .. N-1]$. This means that any integer can be taken as an element of J_i, and we can ignore this set for the intersection operation mentioned above. Example 4.1 illustrates the computation of the set of admissible increments with $b = 3$.

Example 4.1

	31	58	67	123	142	146	154	187	198	220
1st difference		27	9	56	19	4	8	33	11	22
2nd difference			36	65	75	23	12	41	44	33
3rd difference				92	84	79	31	45	52	66

Figure 4.2 Set of keys for Example 4.1

Suppose $N = 74$ (later we will see how to choose the value of N), it follows from (4.4) that

$$J_4 = [0,24] \cup [68,73],$$

$$J_5 = [0, 5] \cup [35,73]$$

$$J_6 = [0, 1] \cup [24,73]$$

$$J_7 = [2,67]$$

By taking the intersection of all the J_i J_1, J_2, J_3 since they are all [0..73]), we obtain

$$J_4 \cap J_5 \cap J_6 \cap J_7 = \Phi.$$

It follows from the previous proposition 1 that a Quotient Reduction perfect hashing function does not exist for the given key set with $N = 74$. However, with $N = 73$,

$$J_4 = [0,22] \cup [65,72]$$

$$J_5 = [0, 3] \cup [32,72]$$

$$J_6 = [21,72]$$

$$J_7 = [0,64] \cup [72]$$

$$J_4 \cap J_5 \cap J_6 \cap J_7 = [72].$$

Thus, there exists a Quotient Reduction perfect hashing function for $N = 73$. □

The number of buckets m for a given N and s

The number of buckets m for a given set of keys I, a quotient N, and an admissible increment s is given by

$$m = \begin{cases} \lfloor r/N \rfloor + 1 & \text{if } \{r \bmod N + (x_1 + s) \bmod N\} \le N-1 \\ \lfloor r/N \rfloor + 2 & \textit{otherwise} \end{cases}$$

where r is the range of keys, $r = x_n - x_1$. This can be readily shown as follows:

$$m = h(x_n) - h(x_1) + 1$$

$$= \left\lfloor (x_n+s)/N \right\rfloor - \left\lfloor (x_1+s)/N \right\rfloor + 1$$

$$= \left\lfloor (x_1+s+r)/N \right\rfloor - \left\lfloor (x_1+s)/N \right\rfloor + 1. \tag{4.5}$$

The first term is reduced to

$$\left\lfloor (x_1+s+r)/N \right\rfloor = \begin{cases} \left\lfloor (x_1+s)/N \right\rfloor + \lfloor r/N \rfloor & \text{if } \{(x_i+s) \bmod N + r \bmod N\} \le N-1 \\ \left\lfloor (x_1+s)/N \right\rfloor + \lfloor r/N \rfloor + 1 & \textit{otherwise} \end{cases}$$

Thus, it follows from (4.5) that

$$m = \begin{cases} \lfloor r/N \rfloor + 1 & \text{if } \{r \bmod N + (x_1 + s) \bmod N\} \le N-1 \\ \lfloor r/N \rfloor + 2 & \textit{otherwise} \end{cases} \tag{4.6}$$

The upper bound of N

The previous result leads to the upper bound of N for given I and b. The minimum number of buckets m^* is given by

$$m^* = \lceil n/b \rceil .$$

The upper bound of N which yields the minimum number of buckets m^* is given by (4.6),

$$m^* = \lfloor r/N \rfloor + 2 \quad (\textit{where } r \bmod N \neq 0). \tag{4.7}$$

Therefore,

$$m^* - 2 = \lfloor r/N \rfloor$$

$$r = (m^*-2)N + r \bmod N.$$

Since $1 \le r \bmod N \le N-1$,

$$(m^*-2)N + 1 \le r \le (m^*-1)N + N - 1$$

$$\frac{r+1}{m^*-1} \le N \le \frac{r-1}{m^*-2}.$$

Since N is an integer, the upper bound of N is given by

$$N^* = \left\lfloor \frac{r-1}{m^*-2} \right\rfloor = \left\lfloor \frac{r-1}{\lceil n/b \rceil - 2} \right\rfloor . \tag{4.8}$$

Solution space of the Quotient Reduction method

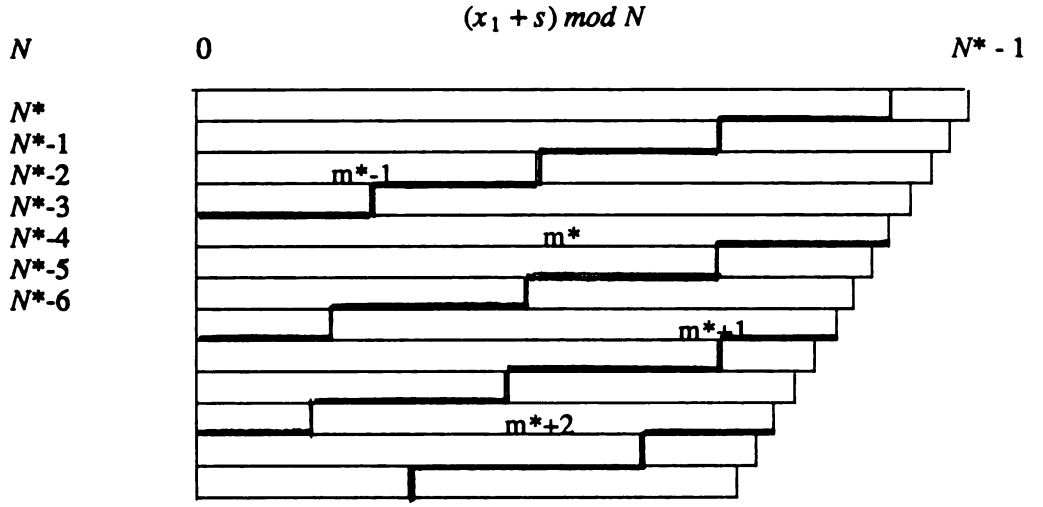
Figure 4.3 illustrates the relationship between quotient N, admissible increment s, and the number of buckets m for a given set of keys. The horizontal axis represents the value of $(x_1 + s) \bmod N$, with s as a variable. The vertical axis represents the value of the quotient N. The first row of the figure corresponds to $N = N^*$. The left hand side of the boundary in the first row corresponds to the space in which the following inequality holds,

$$(x_1 + s) \bmod N^* \leq (N^* - 1) - r \bmod N^*.$$

The values N^* and s in this space yields $m^* - 1$ buckets. The space in which the following inequality

$$(x_1 + s) \bmod N^* > (N^* - 1) - r \bmod N^*$$

holds is in the right hand side of the boundary. The values s and N $(= N^*)$ in this space yields m^* buckets. Similarly, each row represents the solution space for a different N.



N^* denotes the upper bound of N.
m^* denotes the minimum number of buckets.

$$N^* = \lfloor (r-1)/(m^*-2) \rfloor$$

$$m^* = \lceil n/b \rceil$$

Figure 4.3 Solution space

Consider the range of possible N which give a fixed number of buckets m. Let NL_1 and NU_1 denote the lower bound and the upper bound of N respectively, when $\{r\ mod\ N + (x_1+s)\ mod\ N\} \le N-1$, NL_1. They can be obtained as follows.

$$m = \lfloor r/N \rfloor + 1,$$

$$r = (m-1)N + r\ mod\ N,\ since\ 0 \le r\ mod\ N \le N-1$$

$$(m-1)N \le r \le (m-1)N + N-1$$

$$(r+1)/m \le N \le r/(m-1).$$

Since N is an integer, the lower bound NL_1 and upper bound NU_1 are

$$NL_1 = \lceil (r+1)/m \rceil, \quad NU_1 = \lfloor r/(m-1) \rfloor .$$

Similarly, when $\{r\ mod\ N + (w_1+s)\ mod\ N\} \ge N$, the lower bound NL_2 and the upper bound NU_2 are given by

$$m = \lfloor r/N \rfloor + 2,$$

$$(r+1)/(m-1) \le N \le r/(m-2),$$

and hence,

$$NL_2 = \lceil (r+1)/(m-1) \rceil , \quad NU_2 = \lfloor r/(m-2) \rfloor .$$

Algorithm QR (Quotient Reduction)

We are now ready to introduce our algorithm QR to generate a perfect hashing function for the given set of keys I. The basic idea of this algorithm is to search the solution space of N and s to obtain m^* buckets, m^*+1 buckets and so on.

Algorithm QR

STEP 1 { Initialization }
Compute the range of the key: $r := x_n - x_1$.
Compute the minimum number of buckets: $m := \lceil n/b \rceil$.

STEP 2
Compute the upper bounds and lower bounds of N corresponding to m buckets.

$NU_2 := \lfloor r/(m-2) \rfloor$

$NL_2 := \lceil (r+1)/(m-1) \rceil$

$NU_1 := \lfloor r/(m-1) \rfloor$

$NL_1 := \lceil (r+1)/m \rceil$

for $N := NL_1$ **to** NU_1 **do**
Compute the set of the admissible increments J_i (as described in proposition 1) with N including only those elements t in the J_i's such that the condition
$r \bmod N + (x_1 + t) \bmod N \ge N$ holds.

Take the intersection of J_i' s.
if *not empty* **then goto** STEP 3
end for

for $N := NL_2$ **to** NU_2 **do**
Compute the set of the admissible increments J_i (as described in proposition 1) with N including only those elements t in the J_i's such that the condition
$r \bmod N + (x_1 + t) \bmod N \le N-1$ holds.

Take the intersection of J_i' s.
if *not empty* **then goto** STEP 3

end for
$m := m + 1$
goto STEP 2

STEP 3
s := Choose a value from intersection set.
The perfect hashing function $h(x) = \lfloor (x + s)/N \rfloor$

The range of N that corresponds to m buckets is found as follows. When the range of N is $NU_2 \geq N \geq NL_2$, the condition $\{r \bmod N + (x_1 + J) \bmod N\} \geq N$ holds. When the range of N is $NU_1 \geq N \geq NL_1$, the condition $\{r \bmod N + (x_1 + J) \bmod N\} \leq N-1$ holds.

If the intersection is not empty, any element of the set is acceptable (all give the same load factor). However, to obtain more uniform distribution of hashed keys, choose s^* from the intersection set such that,

$$s^* = \min_{j \in J} | N - (x_1 + j) \bmod N - (x_n + j) \bmod N |.$$

This choice of s^* makes the interval covered by the first bucket, $[x_1 .. p(N-1)]$, as close as possible with that of the last bucket $[qN .. x_n]$, where $p = h(x_1) + 1$ and $q = h(x_n)$. Finally, we need the transformation

$$s = s^* - N\{(x_1 + s^*) \operatorname{div} N\}$$

so that x_1 hashes to bucket zero, *i.e.*, $\left\lfloor (x_1 + s)/N \right\rfloor = 0$.

Example 4.2

Consider the key set I = {31, 58, 67, 123, 142, 146, 154, 187, 198, 220} with $b = 3$, $n = 10$ and the range of keys, $r = x_{10} - x_1 = 189$. The minimum number of buckets is $m^* = \lceil n/b \rceil = 4$.

The upper bound $NU_2 := \lfloor r/(m-2) \rfloor = \lfloor 189/(4-2) \rfloor = 94$.

The lower bound $NL_2 := \lceil (r+1)/(m-1) \rceil = \lceil (189+1)/(4-1) \rceil = 64$.

The upper bound $NU_1 := \lfloor r/(m-1) \rfloor = \lfloor 189/(4-1) \rfloor = 63$.

The lower bound $NL_1 := \lceil (r+1)/m \rceil = \lceil (189+1)/4 \rceil = 48$.

The solution space is shown in Figure 4.4.

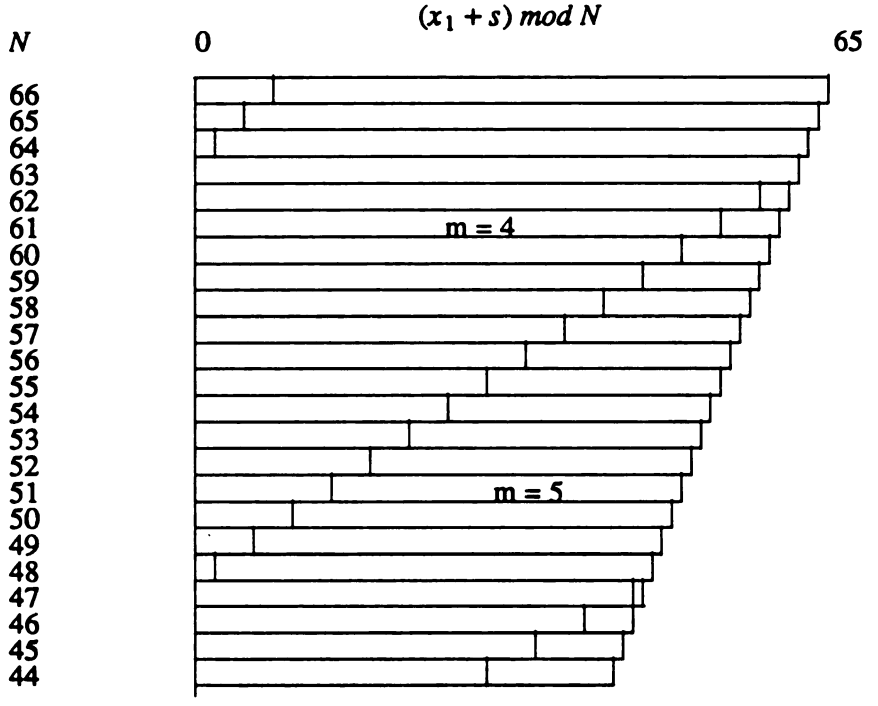


Figure 4.4 Solution space for Example 4.2

The search starts with the value $N = NL_1 = 48$, and the set of admissible increments J = [17,18,19] is found with N = 48 in STEP 2. We move to STEP 3, and the element [18] is chosen which is transformed as $s = 18 - 48\{(31 + 18)\ div\ 48\} = -30$, so that $h(x_1) = 0$. We thus have the Quotient Reduction perfect hashing function,

$$h(x_i) = \left\lfloor \frac{(x_i - 30)}{48} \right\rfloor$$

The hash table for this example is shown below.

Table 4.1 Hash table of Example 4.2

bucket #	0			1	2			3		
x_i	31	58	67	123	142	146	154	187	198	220

Efficient QR Algorithm (EQR)

We can see from (4.8) that the upper bound N^* of the quotient N is proportional to the range of keys r. It is obvious that the algorithm QR is impractical for a large r due to the large computation time. This is because every N is examined sequentially in the algorithm QR.

In this section, we aim to reduce the computation time. The basic idea is to consider the values of N such that $N = \lfloor r/(m-1) \rfloor$ to determine if an admissible increment exists. We begin with $m^* + 1$ buckets. If the search fails, we proceed with $m^* + 2$, and so on. If the search succeeds with $N=N_s$ and m_s buckets, we proceed with $m_s - 1$ buckets and for $N = N_s + 1$ to $\lfloor r/(m-2) \rfloor$. (This is to achieve better storage utilization due to less buckets.) Also, attempt is made to find the smallest N for the value in obtained, so that we have as uniform a distribution as possible.

Algorithm EQR

STEP 1 { Initialization }
Compute the range of the keys: $r := x_n - x_1$.
Compute the minimum number of buckets: $m := \lceil n/b \rceil$. { The value m* }
Compute the upper bound NU for the search: $NU := \lfloor r/(m-1) \rfloor$.
$m := m + 1$ { Starting with m*+1 }

STEP 2
Compute the lower bound NL for the search: $NL = \lfloor r/(m-1) \rfloor$.
Compute the set of admissible increments J_i with NL including only those elements t in the J_i's such that the condition,
$r \bmod NL + (w_1 + t) \bmod NL \le NL-1$, holds.

STEP 3
Take the intersection of J_i's.
if *not empty* **then** { next searches }

```
        Store the result temporarily
        for N := NL+1 to NU do
            Compute the set of admissible increments J_i with N including only those elements t in the J_i's such that the condition,
             r mod N +(x_1 + t) mod N ≤ N−1, holds.
            Take the intersection of J_i's.
            if not empty then      { Search Complete }
                goto STEP 4
        end for
            { Another set of searches }
        Compute the lower bound NL_2 for the search:   NL_2 := ⌈(r + 1)/m⌉ .
        for N := NL_2 to NL−1 do
            Compute the set of admissible increments J_i with N including only those elements t in the J_i's such that the condition
            r mod N +(x_1 + t) mod N ≤ N−1, holds.
            Take the intersection of J_i's.
            if not empty then      { Search Complete }
                goto STEP 4
            else
                Return the result of intersection & NL.
                goto STEP 4
    else
        m := m + 1            { Next bucket size }
        NU := ⌊r/(m−2)⌋       { Next upper bound }
        goto STEP 2
STEP 4
    s := Choose a value from intersection set
    The perfect hashing function h(x) = ⌊(x +s)/N⌋
```

In STEP 2, NL is set to the maximum value of N corresponding to $(m+1)$ buckets. The range of the admissible increments to be examined is calculated by the same method as in the algorithm QR. In STEP 3, a search is made for the quotient NL. If admissible increments exist, they are stored temporarily and further searches in the range $NL+1$ through NU are made that yield m buckets. If the first search fails, the next bucket size and the next upper bound NU are set, and the control is transferred to STEP 2. If one of the searches succeeds, we go to STEP 4 and the result is used in STEP 4. If none of the searches succeeds, another set of searches is made for $N = NL_2$ to $NL - 1$. (Although we have found admissible increments for NL, we need the smallest N that yields the same number of buckets as NL.) The lower bound of these searches NU_2 which corresponds to m buckets is computed by $NU_2 := \lceil (r + 1)/m \rceil$. If none of these searches succeeds, the result of the very first successful search ($N = NL$) is returned, or the result of the first successful search of this set is used in STEP 4. In STEP 4, the value of s is chosen by the

same method as in the algorithm QR.

Example 4.3

For the same set of keys as in example 4.2, the EQR examines if there exist admissible increments with

$$NL = \lfloor r/(m-1) \rfloor = \lfloor 189/(4-1) \rfloor = 69.$$

The search failed for NL=69 in this example, and the next value of NL is set as

$$\lfloor r/(m-1) \rfloor = \lfloor 189/(5-1) \rfloor = 47.$$

This search succeeded, and we obtained the perfect hashing function which requires five buckets. The next set of searches for the range of N that correspond to four buckets were done from N=48 to 68. A set of admissible increments J=[17,18,19] was found in STEP 3, and the perfect hashing function $H(x) = \lfloor (x-30)/48 \rfloor$, was obtained. (This hashing function is the same as that in example 4.2.) □

Complexity of the Algorithms QR and EQR

The taking intersection in the QR and EQR is rather straightforward, and its complexity is $O(n^2)$. (Please refer to the Appendix). In the algorithms QR and EQR, the time complexity is determined by the loop of N. In both algorithms, the total number of execution in these loops in the worst case is proportional to the upper bound of N, $\left\lfloor r/\lceil n/b \rceil - 2 \right\rfloor$. Thus, the complexity of these algorithms is $O(rbn)$. The difference between the two algorithms is the constant of proportionality. Although the constant is small, the complexity is not practically acceptable in many cases. In Remainer Reduction method discussed next, we control r, using modulo operation, and hence the complexity. It is clear that the space requirement of these algorithms is $O(n)$.

Results of the experiments

To test our algorithms, we conducted several experiments using the following real life files.

(A) Userids from IBM System at the University of Waterloo (12,000 keys);

(B) Userids from the msudoc at MSU (600 keys).

All the data was in alphanumeric form and the length of individual keys varied from 2 to 25 characters. The keys were converted into 2 byte long unique integers by the RADIX_Convert method. Details of this method is described in [RM86]. The keys in each file were divided into s groups using hashing functions of the form $H(x)=(((cx+d) \bmod p) \bmod g)$, where p is a large prime number. The hashing function used to separate groups is as follows.

$$H(x) = (((314559x+27182) \bmod 65521) \bmod 9) \tag{4.9}$$

For each of the nine groups obtained, direct perfect hashing functions were generated using algorithms QR and EQR. The average load factors obtained for the nine groups of file (A) are shown Table 4.2. The third row gives the ratio of the average load factor obtained by the EQR to that by the QR.

Table 4.2 Average load factor of nine groups of file (A) (percentage)

1) $n = 100$

Algorithm	b=10	b=20	b=30
QR	69.5	82.0	83.3
EQR	67.8	82.0	83.3
Ratio	97.6	100.0	100.0

2) $n = 250$

Algorithm	b=10	b=20	b=30	b=40	b=50
QR	55.1	69.6	80.1	80.6	82.0
EQR	52.1	68.1	78.5	80.6	82.0
Ratio	94.6	97.8	98.0	100.0	100.0

3) $n = 500$

Algorithm	b=10	b=20	b=30	b=40	b=50
QR	50.3	70.5	78.0	81.0	84.1
EQR	47.9	69.7	77.6	80.4	83.3
Ratio	95.2	98.9	99.5	99.3	99.0

Perfect hashing functions were generated for the keys in file (B) without any partitioning. Table 4.3 shows the load factor for these keys.

Table 4.3 Load factor for file (B) (percentage)

1) $n = 100$

Algorithm	b=10	b=20	b=30
QR	71.4	83.3	83.3
EQR	71.4	83.3	83.3
Ratio	100.0	100.0	100.0

2) $n = 250$

Algorithm	b=10	b=20	b=30	b=40	b=50
QR	61.0	65.8	69.4	78.1	83.3
EQR	61.0	65.8	69.4	78.1	83.3
Ratio	100.0	100.0	100.0	100.0	100.0

3) $n = 500$

Algorithm	b=10	b=20	b=30	b=40	b=50
QR	42.4	75.8	83.3	83.3	83.3
EQR	42.0	75.8	83.3	83.3	83.3
Ratio	99.1	100.0	100.0	100.0	100.0

These tables show that higher b gives better storage utilization. We obtained over 80% load factors for large bucket sizes, whereas about 40% to 70% for small bucket sizes. For large bucket sizes, the load factor remains almost the same as the group size increases, whereas the load factor

falls clearly for small bucket sizes as the group size increases. The ratio of the average load factor obtained by the EQR to that obtained by the QR is close to 100%. However, the computation time of the EQR is not practical even though the EQR is faster than the QR.

5. Remainder Reduction Method

The Quotient Reduction methods described in the previous section has two major problems.

a) The low storage utilization when the keys are not uniformly distributed;

b) Large computation time.

The Remainder Reduction method overcomes these two problems. The basic idea of this method is to scramble the set of keys to obtain a more uniform distribution within a narrow range, followed by Quotient Reduction perfect hashing. Since we can control r in the Remainder Reduction method, the computation time can be controlled. Remainder Reduction perfect hashing functions are of the form

$$h(x_i) = \left\lfloor \{(qx_i) \bmod M + s\}/N \right\rfloor , \tag{5.1}$$

where q, M, s, N are constants to be determined. The transformation

$$x'_i = (qx_i) \bmod M \tag{5.2}$$

accomplishes the scrambling mentioned above by choosing a q and M appropriately. The transformation (5.2) may cause a collision, *i.e.*, more than two distinct primary keys are transformed into identical keys. The probability p of no collision occurring by the transformation can be obtained as follows [FL68].

$$p = \frac{{}_M P_n}{M^n} = \frac{M(M-1)(M-2).....(M-n+1)}{M^n}$$

$$= (1-\frac{1}{M})(1-\frac{2}{M}).....(1-\frac{n-1}{M}).$$

For small positive x, we have $\log(1-x) \simeq -x$,

$$\log p \simeq \frac{1+2+.....+(n-1)}{M} = -\frac{n(n-1)}{2M}.$$

We obtain

$$p \simeq e^{\frac{-n(n-1)}{2M}} . \tag{5.3}$$

The probabilities of having no collisions for n=500, 250, and 100 are shown in Table 5.1.

M	n=500	n=250	n=100
2^{16} (65536)	0.15	0.62	0.93
2^{17}	0.39	0.79	0.96
2^{18}	0.62	0.89	0.98
2^{19}	0.79	0.94	0.99
2^{20}	0.89	0.99	1.00
2^{21} (2097152)	0.94	0.99	1.00

Table 5.1 Probability of having no collisions for given M and n.

If we want to avoid collisions, we have to select suitably large M. For example, M should be greater than 2^{21} when $n = 500$. However, we do not gain the advantages of the Remainder Reduction method mentioned above with this large M.

We can tolerate some collisions by the transformation (5.2). If the maximum number of identical keys is less than or equal to the bucket size b, then in general there exists a quotient reduction perfect hashing function.

On the other hand, the probability that more than b keys collide under this transformation is almost zero. Let $P(\alpha,m,b)$ denote the probability that none of the urns contain more than b balls, where n, $n = \alpha bm$, balls are tossed into m urns. An approximate formula is derived in [RL88].

$$P(\alpha,m,b) \simeq e^{-mPov(\alpha,b)},$$

where,

$$Pov(\alpha,b) \simeq \sum_{i=b+1}^{n} \frac{e^{-b\alpha}(b\alpha)^i}{i!}$$

$$\simeq \frac{(b\alpha)^{b+1}}{(b+1)!} e^{-b\alpha} \left\{ \frac{b+2}{b(1-\alpha)+2} \right\}.$$

For example, when n=500, m=500, and b=10, The probability that more than b keys collide is given by

$$1 - P(0.1,500,10) \approx 1 - e^{-500Pov(0.1,10)}$$

$$\approx 1 - e^{-5.03\times10^{-6}} \approx a\ negligibly\ small\ quantity$$

Now we are ready to introduce our Remainder Reduction algorithm. To implement this method, it is necessary to apply the transformation $x'_i = (qx_i)\ mod\ M$ for each x_i, $1 \le i \le n$, and sort before applying Quotient Reduction method. Let $\{w_1, w_2, \ldots, w_n\}$ denote sorted keys from $\{x'_1, x'_2, \ldots, x'_n\}$. q and M are chosen to be prime numbers.

Algorithm RR

STEP 1

Transform keys $\{x_1, x_2, \ldots, x_n\}$ into $\{x'_1, x'_2, \ldots, x'_n\}$ using $x'_i = (qx_i)\ mod\ M$.

Sort the keys $\{x'_1, x'_2, \ldots, x'_n\}$ in non decreasing order, and obtain sorted keys $\{w_1, w_2, \ldots, w_n\}$.

STEP 2

Apply the algorithm QR to the sorted keys

QR(n, $\{w_1, w_2, \ldots, w_n\}$) *or*

Apply the algorithm EQR to the sorted keys

EQR(n, $\{w_1, w_2, \ldots, w_n\}$)

Perfect hashing function $h(x_i) = \left\lfloor \{(qx_i)\ mod\ M + s\}/N \right\rfloor$

The complexity of Remainder Reduction Algorithm is given by

$$O(nMb),$$

where M is the modulo of the transformation (5.2). As the range of keys is given by M in this method, r in the QR is substituted by M. The space requirement is $O(n)$.

Results of experiments

For each of nine groups obtained by the hashing function (4.9), perfect hashing functions are generated using algorithm RR with the QR and with the EQR. The average load factor of nine groups of file (A) is shown in Table 5.2. Several values of q were used in this experiment, and we observed that the value of q hardly affected the result.

Table 5.2 Average load factor of nine groups of file (A) obtained using RR (percentage)

1) n = 100, q=37, M=2039.

Algorithm	b=10	b=20	b=30
RR using QR	70.0	78.3	83.3
RR using EQR	69.3	78.3	83.3
Ratio	99.0	100.0	100.0

2) n = 250, q=71, M=4093.

Algorithm	b=10	b=20	b=30	b=40	b=50
RR using QR	65.2	78.9	80.8	84.3	85.2
RR using EQR	64.1	77.4	80.8	84.3	85.2
Ratio	98.3	98.1	100.0	100.0	100.0

3) n = 500, q=101, M=8191.

Algorithm	b=10	b=20	b=30	b=40	b=50
RR using QR	56.7	72.0	78.7	81.7	81.9
RR using EQR	56.1	71.8	78.2	81.7	81.9
Ratio	98.9	98.7	99.4	100.0	100.0

Table 5.3 shows the load factor for file (B)

Table 5.3 Load factor for file (B) obtained using RR (percentage)

1) n = 100, q=37, M=2039.

Algorithm	b=10	b=20	b=30
RR using QR	76.9	83.3	83.3
RR using EQR	76.9	83.3	83.3
Ratio	100.0	100.0	100.0

2) $n = 250$, q=71, M=4093.

Algorithm	b=10	b=20	b=30	b=40	b=50
RR using QR	71.4	78.1	83.3	89.3	83.3
RR using EQR	71.4	78.1	83.3	89.3	83.3
Ratio	100.0	100.0	100.0	100.0	100.0

3) $n = 500$, q=101, M=8191.

Algorithm	b=10	b=20	b=30	b=40	b=50
RR using QR	60.2	69.4	79.4	83.3	83.3
RR using EQR	57.5	69.4	79.4	83.3	83.3
Ratio	95.5	100.0	100.0	100.0	100.0

We observe that the load factors are almost above 70%. The average computation time for generating a perfect hashing function is reduced to only a few seconds on VAX 8600. Thus, we conclude that RR method of finding perfect hashing function is a practical technique.

6. Universal Classes of Hashing Functions

In order to compare the proposed method to other methods such as simple trial and error method, we will examine the probability of a randomly chosen function being perfect. Let $P(n,m,b)$ denote the the probability that a randomly chosen function is perfect, where n is the number of keys, m is the number of buckets, and b is the bucket size. A way of computing $P(n,m,b)$ using the following simple recurrence relation is described in [RL88]:

$$P(n+1,m,b) = P(n,m,b) - {}_nC_b P(n-b,m-1,b)\frac{(m-1)^{n-b}}{m^n} \tag{6.1}$$

However, choosing a function for a trial at random from the set of all functions is clearly impractical. Hence, *universal*$_2$ classes of hashing functions were introduced in [RL88].

Let H be a class of functions which map a set of integers $A = \{0,1,2,....,a-1\}$ into the set of integers $B = \{0,1,2,....,m-1\}$, where $a \gg m$. The set A corresponds to the set of keys and B corresponds to the set of addresses. Let x and y be distinct integers $x, y \in A$ and $h_i \in H$ a hashing function. We define

$$\delta_{h_i} = \begin{cases} 1 & \text{if } x \neq y \text{ and } h_i(x) = h_i(y) \\ 0 & \textit{otherwise} \end{cases}$$

If $\delta_{h_i} = 1$, then x and y are said to collide under h_i. The class H is said to be a *universal*$_2$ class of hashing functions if for all $x,y \in A$, $\sum_{h_i \in H} \delta_{h_i}(x,y) \leq |H|/m$. This means that H is a *universal*$_2$ class of hashing functions if for every pair $x,y \in A$, they collide no more than $|H|/m$ functions. This implies that if a function is chosen randomly from a *universal*$_2$ class, the probability of a pair of keys colliding is less than or equal to $1/m$.

In [RM86], Ramakrishna performed a series of experiments using several test files to determine if the *universal*$_2$ class H_1 functions could be used for selecting functions for trials. He showed that the relative frequency of perfect hashing functions within *universal*$_2$ class H_1 is the same as that predicted by the analysis for the set of all functions. However, he did not study other *universal*$_2$ hashing functions: class H_2 and H_3 functions. Since class H_2 and H_3 functions do

not require multiplication (only need Exclusive OR), they may be convenient for long keys. The experiments here are aimed at examining if class H_2 and class H_3 functions behave similar to the class H_1. The *universal*$_2$ class H_3 is defined as follows by Carter and Wegman [CW79]:

Let A be the set of keys of i bit binary numbers, and B be the set of j bit binary numbers. The set B corresponds to 2^j buckets, and the number of bucket has to be a power of two. Let Mt be the set of arrays of length i whose elements are from B. We can regard Mt as i by j boolean matrices. For $mt \in Mt$, let $mt(k)$ be the bit string which is the kth element(row) of the matrix mt, and for $x \in A$, let x_k be the kth bit of x. Define $f_{mt}(x) = x_1 mt(1) \oplus x_2 mt(2) \oplus \cdots \oplus x_i mt(i)$, where $\oplus$ denotes exclusive OR operation. The meaning of this function is to take exclusive ORs of those elements in mt that correspond to 1 bits of key x. The class H_3 is defined as a the set of functions $\{f_{mt} \mid mt \in Mt\}$, over all possible matrices mt. An example of class H_3 is shown below.

Example 6.1

When the number of buckets $m = 4$, the bucket size $b = 3$, for the set of keys, which consists of ten 8-bit keys, given in Figure 4.2, a hashing function is found as follows:
A 8×2 boolean matrix mt is obtained by generating eight random numbers in the range of 0 to 3. The following is one such matrix.

mt(1)	01
mt(2)	11
mt(3)	10
mt(4)	00
mt(5)	10
mt(6)	11
mt(7)	00
mt(8)	01

Keys 67 and 123 are converted to binary form $(01000011)_2$ and $(01111011)_2$, respectively. The hash addresses of these keys are given by

$$f_{mt}(67) = mt(2) \oplus mt(7) \oplus mt(8) = 11 \oplus 00 \oplus 01 = 10$$
$$f_{mt}(123) = mt(2) \oplus mt(3) \oplus mt(4) \oplus mt(5) \oplus mt(7) \oplus mt(8)$$
$$= 11 \oplus 10 \oplus 00 \oplus 10 \oplus 00 \oplus 01 = 10$$

Keys 67 and 123 are then stored in bucket 2. Similerly after computing all the hash addresses, we obtain the following hash table

Table 6.1 A hash table (example of failure)

bucket #	0	1	2	3
keys	31 58 142 187	146 198	67 123	154 220

Since the bucket size b is 3, this trial has failed. However, the following mt gives a perfect hashing function:

mt(1)	01
mt(2)	00
mt(3)	10
mt(4)	11
mt(5)	00
mt(6)	01
mt(7)	10
mt(8)	11

Using this matrix mt, we obtain the hash table given below.

Table 6.2 A hash table (example of success)

bucket #	0	1	2	3
keys	123 146 154	67 187	142 198	31 58 220

□

The definition of the class H_2 is similar to that of H_3 except that a key is mapped into a longer bit string. Let A be the set of keys of i-digit numbers written in base α. For $x \in A$, let x_k

denote the kth digit of x. B has the same definition as in H_3. Define g to be the function which maps x into the bit strings of length $i\alpha$ which has 1's in positions x_1+1, x_1+x_2+1, $x_1+x_2+x_3+1$, ···, etc, where x_k is the kth bit of x. Then, Mt is the set of arrays of length $i\alpha$ whose elements are from B. Mt can be regarded as $i\alpha$ by j matrices. We can apply the $i\alpha$ bit keys defined above to the same operation as in H_3. If H_3 is the class defined above for $i\alpha$ bit keys, then $H_2 = \{fg \mid f \in H_3\}$. Although H_2 needs more space than H_3, the computation time of H_2 is less than that of H_3, because converted keys in H_2 have fewer 1-bits.

Example 6.2

When the base $\alpha = 4$, the same keys as in *Example 6.1* are mapped into strings of length 16. For example, keys 67 and 123 are converted as follows.

$$g(67) = g((1003)_4) = 0100100000000000$$
$$g(123) = g((1323)_4) = 0100101001000000.$$

We can apply the same operation to these converted keys as in *Example 6.1*. mt is a 16 by 2 boolean matrix and is generated similarly. □

Results of experiments

A set of experiments was conducted as follows. A set of 100 hashing function was created randomly by generating random numbers in the range of 0 to $2^j - 1$. The first n keys, $n=\alpha mb$, in each of nine groups of file (A) were hashed by each of the 100 hashing functions, and the number of perfect hashing functions was recorded. n was selected so that the load factor varies from 50% to 100%. This experiment was repeated for various values of m and b. Figure 6.1.1 shows the results of one set of experiments with b = 10 and m = 16 using the group 8 of file (A). The solid line is a plot of $P(n, 16, 10)$ computed using the recurrence relation (6.1). The symbols 2 and 3 represent the experimental probabilities obtained with class H_2 and class H_3 of hashing functions respectively.

Figure 6.1.2 shows the results of another set of experiments with $b = 40$ and $m = 8$ using the fourth group of file (A). The solid line is a plot of $P(n, 8,40)$. The number of keys varied from 160 to 320. We used the same set of 100 hashing functions for all load factors and each of nine groups.

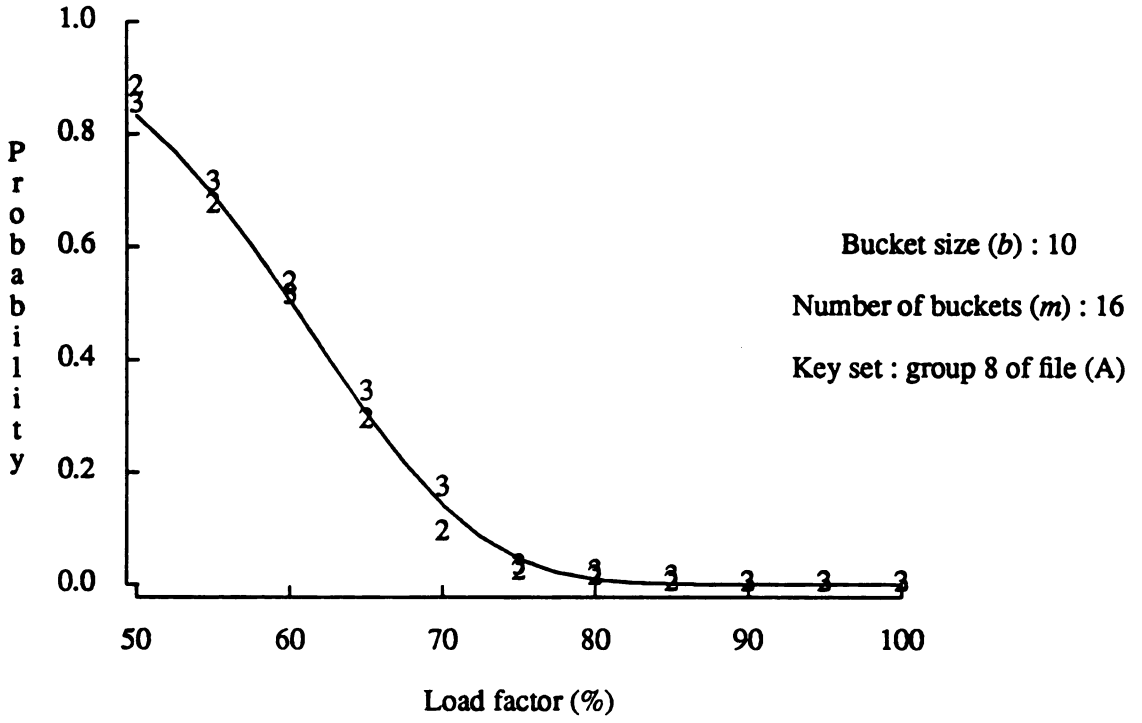


Figure 6.1.1 Observed and computed probability of a trial succeeding

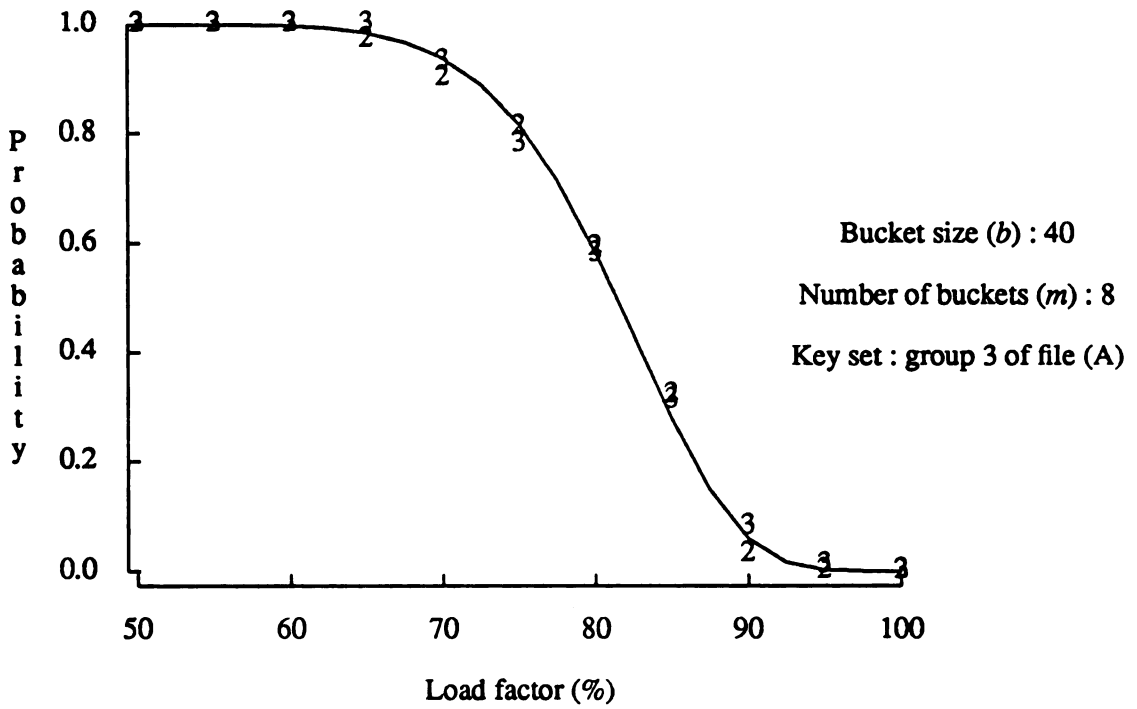


Figure 6.1.2 Observed and computed probability of a trial succeeding

We computed the 95% critical region for each point. The relative frequency of perfect hashing functions will fall within the region bounded by θ_L and θ_U with 95% probability. Let $P(\alpha mb,m,b) = \theta$, then the approximate values of θ_L and θ_U may be obtained using the relations $\theta_L = \theta - 1.96\sigma$ and $\theta_U = \theta + 1.96\sigma$, where $\sigma = \sqrt{\theta(1-\theta)/100}$. We observed that almost all the experimental values fall within the critical region. The result indicates that relative frequency of perfect hashing functions in class H_2 and class H_3 is the same as that predicted by the theoretical analysis in [RL88] for the set of all functions.

Comparison with trial and error method

In view of the analysis in [RL88], $P(\alpha mb,m,b)$ is a good measure of the cost of perfect hashing. Figure 6.2 compares $P(\alpha mb,m,b)$ to the experimental results shown in Table 5.2.

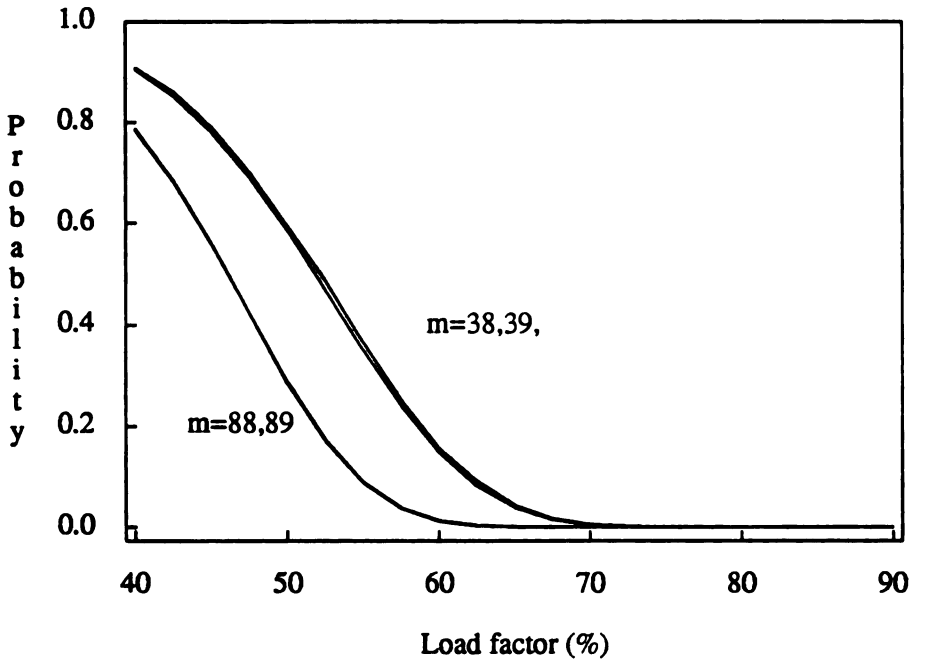


Figure 6.2.1 Probability of a randomly chosen function being perfect ($b = 10$)

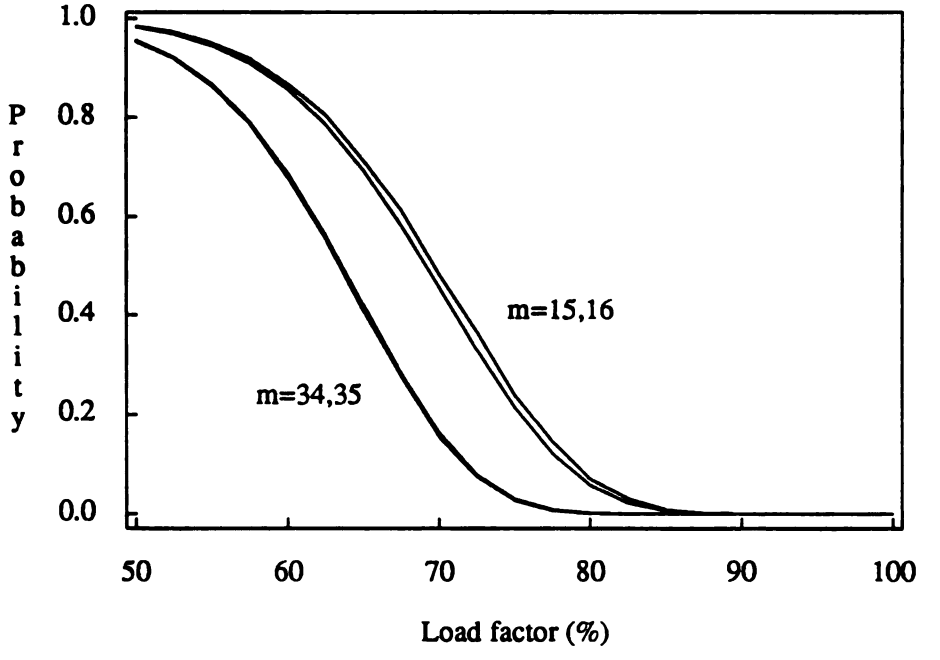


Figure 6.2.2 Probability of a randomly chosen function being perfect ($b = 20$)

We aim to see how many trials would be required if we used trial and error method to obtain the load factor in Table 5.2. Figure 6.2.1 plots the probabilities $P(\alpha mb,m,b)$ as a function of the load factor with b = 10 and m = 38, 39, 88, and 89. In Table 5.2, about 65% load factor with n = 250 was obtained. In other words, on average $m = n/\alpha b = 38.5$ buckets were produced in the experiments. In Figure 6.2.1, the points $P(65,38,10)$ and $P(65,39,10)$ correspond to the experimental result with n = 250 in Table 5.2. When n = 500, about 55% load factor was obtained with b = 10. The points $P(65,88,10)$ and $P(65,89,10)$ then correspond to the experimental result. We observe that these load factors obtained by the experiment can be mapped into the range of the probability from 0.05 to 0.1 in $P(\alpha mb,m,b)$. This means that 10 to 20 trials are required to find a perfect hashing function with load factor 65% for 250 keys and with load factor 55% for 500 keys when the bucket size is 10.

Similarly, Figure 6.2.2 plots $P(\alpha mb,m,b)$ with b = 20 and m = 15, 16, 34, and 35. The point at $\alpha = 79\%$ on the lines with m = 15 and 16 correspond to the experimental result with n = 250, and the points at $\alpha = 72\%$ on the lines with m = 34 and 35 correspond to the result with n = 500 in Table 5.2. We observe that the load factors with n = 250 and 500 in Table 5.2 can be mapped into the range from 0.05 to 0.1 in $P(\alpha mb,m,b)$. The other load factors obtained in Table 5.2 were mapped into the range from 0.05 to 0.4 in $P(\alpha mb,m,b)$.

7. Dynamic Behavior

In previous sections, we discussed about the static model: *i.e.*, we showed how to find perfect hashing functions for a given set of keys. In this section, we will discuss about the dynamic behavior of the proposed schemes such as frequency of rehashing and dynamic load factor.

Insertions and rehashing

In our scheme, an insertion causes rehashing when a new key is hashed into a full bucket or when the address of the new key is beyond the range of the existing hash table. The latter can happen when the new key is extremely large (or small) compared to the keys already stored in the hash table. For the Remainder Reduction perfect hashing function $h(x_i) = \left\lfloor \{(qx_i \bmod M) + s\}/N \right\rfloor$, the probability P_R, of an insertion causing rehashing for the given state where the set of keys has been already hashed, is given by

$$P_R = \frac{R_o + R_f}{M}, \tag{7.1}$$

where R_o is the range of keys not covered by the table, and R_f is the range of keys covered by full buckets. R_o and R_f are given by (7.2) and (7.3) respectively.

$$R_o = max(s, 0) + max((M-1) - (m+1)N, 0), \tag{7.2}$$

where m is the number of buckets.

$$R_f = \sum_{i \in F} R_{fi}, \tag{7.3}$$

where R_{fi} is the range of bucket i, and $F = \{i \mid \text{bucket } i \text{ is full}\}$. R_{fi} is calculated as follows:

$$R_{fi} = \begin{cases} N + min(s, 0) & (\textit{the first bucket}) \\ N + min((M-1) - (m+1)N, 0)) & (\textit{the last bucket}) \\ N & (\textit{otherwise}) \end{cases}$$

If the hash table covers the entire range of M, the term R_o is equal to zero because the range of

keys is equal to M. The probability P_R was computed using (7.1) for each state obtained in Table 5.2. Table 7.1 shows the result.

Table 7.1 Probability of an insertion causing rehashing using RR

n	b=10	b=20	b=30	b=40	b=50
100	0.223	0.211	0.070	-	-
250	0.088	0.142	0.110	0.144	0.208
500	0.049	0.077	0.083	0.107	0.066

For each bucket size, on average, the probability decreases as the number of keys increases. For practical range of the group size, the probability of rehashing is small: with b=50, only one in about 15 insertions causes a rehash when the group size is 500 records. These results are close to those results obtained by a trial and error method in [RL88].

The probability of an insertion causing rehashing is affected by the load factor and the uniformity of the distribution of keys. It is clear that the probability is lower when there are fewer full buckets: *i.e.*, when the load factor is lower on average. For the same load factor, more uniform distribution causes the lower probability. This is determined by the uniformity of the distribution of primary keys in the Quotient Reduction method. Table 7.2 shows the probability of an insertion causing rehashing in the Quotient Reduction and Remainder Reduction methods.

Table 7.2 Probability of an insertion causing rehashing (n=250)

(The load factor in percentage is given within parenthesis.)

Algorithm	b=10	b=20
QR	0.136 (55.1)	0.158 (69.6)
RR	0.088 (65.2)	0.142 (78.9)

In Table 7.2, when b=10, the probability decreases from 13.6% to 8.8% while the load factor increasing 55.1% to 65.2%. This implies that the transformation (5.2) in the Remainder Reduction method yielded more uniform distribution than that of original keys.

In our methods, not only the uniformity of the distribution of keys but also the quotient N affects the probability of rehashing, because smaller N usually yields the fewer full buckets. In STEP 3 of algorithm EQR, "Another set of searches" is made in order to find the smallest value of N among the values that correspond to the same number of buckets. Table 7.3 shows the improvement of the load balance by this set of searches.

Table 7.3 Comparison of the probability of rehashing

n	Algorithm	b=10	b=20	b=30	b=40	b=50
100	EQR'	0.229	0.435	0.522	-	-
	EQR	0.223	0.211	0.070	-	-
250	EQR'	0.095	0.169	0.258	0.316	0.492
	EQR	0.088	0.142	0.110	0.144	0.208
500	EQR'	0.053	0.087	0.138	0.115	0.147
	EQR	0.049	0.077	0.083	0.107	0.066

EQR' denotes the algorithm EQR without "another set of searches".

We observe that the probability is decreased in all cases in Table 7.3. Especially, when the number of buckets is small: *i.e.*, for n=100 b=30 and n=250 b=40,50, the probability of rehashing is reduced drastically. The low probability for small bucket sizes makes the incrementally built up file, which is made by one insertion at a time starting an empty file, more efficient.

Dynamic Load Factor

We can build a file using the proposed external perfect hashing scheme in two ways. The file can be built incrementally by making one insertion at a time starting from an empty file. Whenever a bucket overflows, a new perfect hashing function is found by the proposed methods

for the keys in the group. We refer to the file constructed by this method as an incrementally built file. On the other hand, if all the records are available at a time, the records are divided into groups and then are stored using direct perfect hashing functions. We refer to this as initial loading. We will examine the load factor of an incrementally built file or group (dynamic load factor).

Figure 7.1.1 and Figure 7.1.2 show the load factor of a group, using algorithm RR, plotted as a function of the number of records in the group. The oscillation for the small number of records are due to fragmentation. Peaks and valleys appear when the number of records in the group is around b, $2b$, $3b$,... . In an incrementally built group, the load factor increases when an insertion does not cause a rehash. If an insertion causes a rehash, the resulting load factor is the same as that for a group built by initial loading. The experiment was done for the keys in file(A) using the Remainder Reduction hashing function with parameters $q = 101$ and $M = 8191$. The initial perfect hashing function for x_i $(1 \le i \le b)$ is set to $h(x_i) = \left\lfloor \{(101x_i) \bmod 8191\}/8191 \right\rfloor$. The ordinary hashing function (4.9) was not used due to the small file size. Figure 7.1.1 shows the load factor with $b = 10$, and Figure 7.1.2 shows the load factor with $b = 40$.

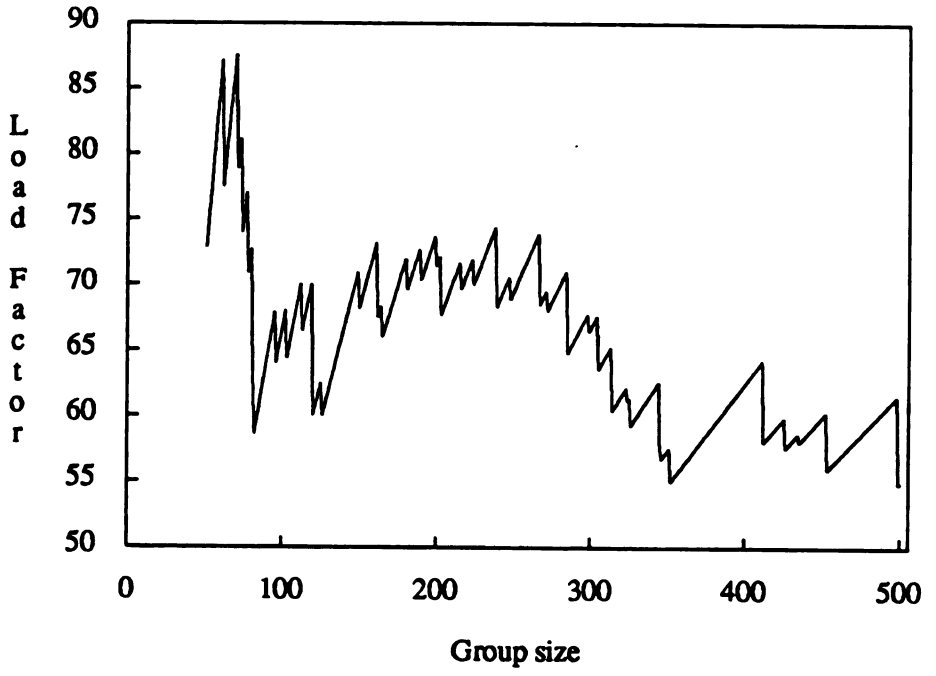


Figure 7.1.1 Load factor of a group ($b = 10$)

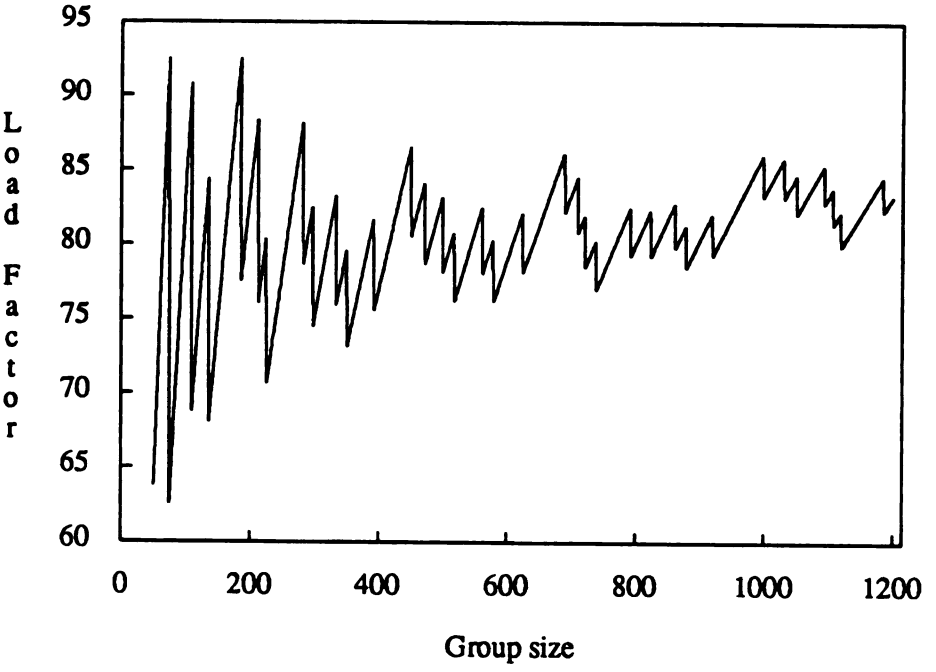


Figure 7.1.2 Load factor of a group ($b = 40$)

The result gives a significant advantage to the external perfect hashing scheme in practice. When b=40, the load factor is over 80% on average. As the group size gradually increases from 500 to 1000, the load factor still stays over 80%. The oscillation becomes smaller as the number of buckets increases.

In order to examine the overall load factor of an incrementally built file, we implemented the proposed composite hashing scheme and simulated with keys in file (A). Figure 6.2 plots the overall load factor of the file as a function of the group size. The ordinary hashing function (4.9) and the Remainder Reduction hashing function with parameters $q = 101$ and $M = 8191$ were used. We observe the behavior similar to above. The effect of the bucket size on the load factor of a file is very significant. Clearly, the small size ($b = 10$) seems impractical because of the resulting low load factor. These results are close to those results obtained in [RL88].

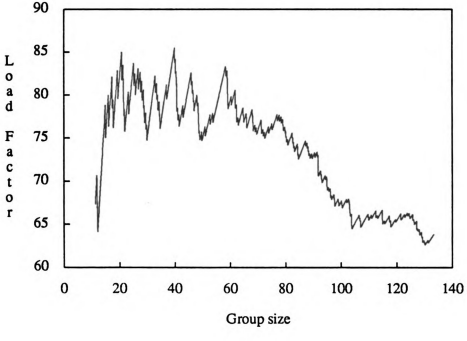


Figure 7.2.1 Overall Load factor (b=10)

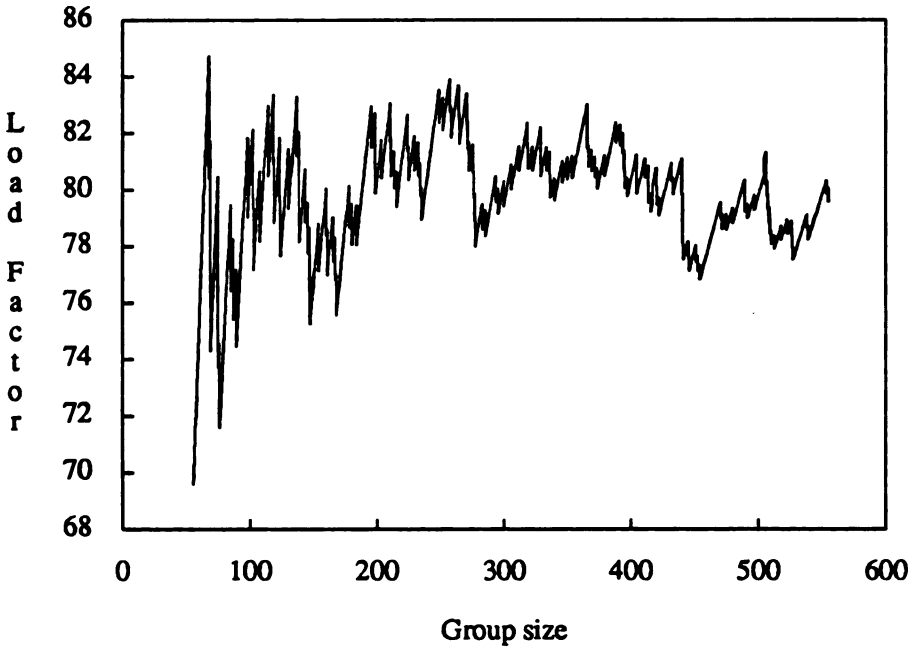


Figure 7.2.2 Overall Load factor ($b = 40$)

8. Conclusions and Future Work

External perfect hashing scheme was proposed in [RL88] and shown to be practical and competitive. One drawback is a trial and error method of finding direct perfect hashing functions. We investigated systematic methods of finding direct perfect hashing functions.

We gave algorithms to find the Quotient Reduction perfect hashing functions. However, the Quotient Reduction method is not practical for a large set of keys due to the computation time. Another disadvantage is that this method is sensitive to the uniformity of the distribution of primary keys. The uniformity of the distribution affects not only the storage utilization but also the probability of rehashing.

The Remainder Reduction method is superior to the Quotient Reduction method. The results of experiments indicate that the proposed method for finding perfect hashing functions is not only practical but also competitive to the other methods such as a trial and error method with reference to the resulting load factor and dynamic behavior.

We also showed experimentally that the relative frequency of perfect hashing functions within $universal_2$ class H_2 and H_3 is the same as that predicted by the analysis for the set of all functions.

In algorithm Efficient QR including procedure *Search*, there is still much room for improvement to reduce the computation time and to obtain better storage utilization.

APPENDIX

APPENDIX

The *INT* Procedure takes intersection of $J_i's$ and finds common elements of each set of admissible increments. An interval is represented by the starting and end values in this procedure. The procedure receives three arguments: the quotient N, the starting value of the set of admissible increments j_s, and the range of the set j_r, and returns the set of intervals Ij (if any), which are common elements, with setting the flag *exist* to *true*.

The calculation of j_s and j_r is as follows .

When $\{r \bmod N + (x_1 + J) \bmod N\} \leq N-1$,

$$0 \leq (x_1 + J) \bmod N \leq N - r \bmod N - 1.$$

Then

$$j_r = (N - r \bmod N - 1) - 0 = (N-1) - r \bmod N .$$

On the other hand, j_s is given by

$$(x_1 + j_s) \bmod N = 0,$$

and hence,

$$j_s = N - x_1 \bmod N.$$

When $\{r \bmod N + (x_1 + J) \bmod N\} \geq N$,

$$N - r \bmod N \leq (x_1 + J) \bmod N \leq N - 1.$$

$$j_r = N - 1 - (N - r \bmod N) = r \bmod N - 1.$$

On the other hand, j_s is given by

$$(x_1 + j_s) \bmod N = N - r \bmod N.$$

Thus,

$$j_s = (N - x_1 \bmod N) + (N - r \bmod N).$$

Procedure *INT* is outlined below.

Procedure *INT* (N, J_s, J_r, Ij, *exist*)
begin
{ Make intervals for the solution space. }
if $j_s + j_r \geq N$ **then**
Create two intervals: $Ij_1 := [0 .. j_s + j_r - N - 1]$ and $Ij_2 := [j_s .. N-1]$.
else

```
        Create the interval: Ij_1 := [j_s .. j_s+j_r-1].
    { check the interval [x_i .. x_{i+b}], 1 ≤ n ≤ n-b. }
    for i := 1 to n - b do
        δ := x_{i+b} - x_i
        if δ < N then
            { Create intervals of admissible increments. }
            Compute the starting value Tj_s and the range Tj_r.
                Tj_s := (N - x_{i+b}) mod N
                Tj_r := δ
            if Tj_s + Tj_r ≥ N then
                Create two intervals:
                    T_1 := [TS_1 .. TE_1] := [0 .. Tj_s+Tj_r-N-1]
                    and
                    T_2 := [TS_1 .. TE_1] := [Tj_s .. N-1].
            else
                Create the interval T_1 := [TS_1 .. TE_1] := [T_s .. T_s+T_r-1].
            { Take intersection of intervals I_j with T's. }
            case of the number of T's being one
                for each interval Ij_i := [s_i .. e_i] do
                    if (TS_1 > e_i) or (TE_1 < s_i) then
                        Delete the interval Ij_i := [s_i .. e_i].
                    else
                        Change the interval Ij_i to [max(TS_1,s_i) .. min(TE_1,e_i)].
                        if the set of intervals Ij_i = Φ then
                            return (exist := false)
                end for
            case of the number of T's being two
                for each interval Ij_i := [s_i .. e_i] do
                    if (TS_1 > e_i) or (TE_1 < s_i) or ((TE_1 < s_i) and (TS_2 > e_i)) then
                        Delete the interval Ij_i := [s_i .. e_i].
                    else if s_i ≤ TE_1 then
                        if e_i ≥ TS_2 then
                            Divide the interval Ij_i into two intervals,
                            [max(TS_1,s_i) .. TE_1] and [TS_2 .. min(TE_2,e_i)].
                        else
                            Change the interval Ij_i to [max(TS_1,s_i) .. min(TE_1,e_i)].
                    else
                        Change the interval Ij_i to [max(TS_2,s_i) .. min(TE_2,e_i)].
                        if the set of intervals Ij_i = Φ then
                            return (exist := false)
                end for
    end for
    return( Ij, exist )
end
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[CC80] Cichelli, R.J. *Minimal perfect hash functions made simple.* Comm. of the ACM, 23, 1, (1980), 17-19

[CH84] Chang, C.C. *The study of an ordered minimal perfect hashing scheme.* Comm. of the ACM, 27, 4, (1984), 384-387

[CR85] Cormak, G.V., Horspool, R.N.S. and Kaiserswerth, M. *Practical Perfect Hashing.* The Computer Journal, 28, 1(1985), 54-58

[CW79] Carter, L.J. and Wegman, M.L. *Universal classes of hash functions.* Journal of Computer and System Sciences, 18, 2(1979), 143-154

[FL68] Feller, W. *An introduction to probability theory and its applications.* Vol.1. New York: John Wiley, 1968

[FR82] Fredman, M.L., Komlos, J. and Szemeredi, E. *Storing a sparse table with O(1) worst case access time.* Proc. 23rd Symposium on Foundations of Computer Science, IEEE Computer Society, 1982, 165-168

[JS81] Jaeschke, G. *Reciprocal Hashing: A method for generating minimal perfect hashing functions.* Comm. of the ACM, 24, 12(1981), 829-833

[MR84] Mairson, H.G. *The program complexity of searching a table.* Ph.D. Thesis, Department of Computer Science, Stanford University, 1984.

[RL88] Ramakrishna, M.V. and Larson, P.A. *File organization using composite perfect hashing.* (to appear) ACM Trans. on Database System. (Earlier version in 1985 ACM-SIGMO Intn'l conference on management of data, 190-200.)

[RM86] Ramakrishna, M.V. *Perfect Hashing for External Files.* Ph.D. Thesis, Department of Computer Science, University of Waterloo, Research Report No. CS-86-25, 1986.

[SP77] Sprugnoli, R. J. *Perfect Hashing Functions: A Single Probe Restructing Method for Static Sets.* Comm. of the ACM, 20, 11(1977), 841-850.