Northeastern University et al v. Google, Inc., Doc. 78 Att. 9

EXHIBIT I

Dockets.Justia.com

Automatic Text Processing

The Transformation, Analysis, and Retrieval of Information by Computer

Gerard Salton

Cornell University

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California
New York • Don Mills, Ontario • Wokingham, England
Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan

This book is in the Addison-Wesley Series in Computer Science

Michael A. Harrison, Consulting Editor

QA76.9
.T48
.S25
1988

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

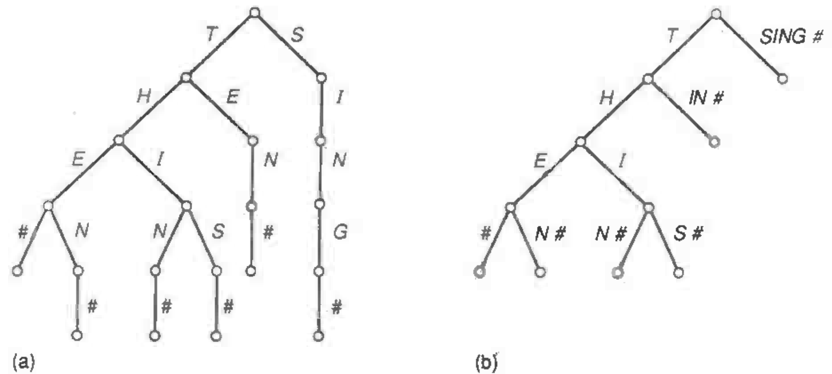
Salton, Gerard.
Automatic text processing: the transformation, analysis, and retrieval of information by computer / by Gerard Salton.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-201-12227-8
1. Text processing (Computer science) I. Title.
QA76.9.T48S25 1989 88-467
005--dc19 CIP

Copyright © 1989 by Addison-Wesley Publishing Company, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, or photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

ABCDEFGHIJ - HA - 898

M.I.T. LIBRARIES
FEB 24 2004
RECEIVED

Figure 7.31 Typical alphabetic search trees. (a) Full digital-search tree. (b) Truncated digital-search tree.



words. For larger dictionaries with tens of thousands of different entries, balanced indexing structures such as B-trees are preferable.

7.7 Hash-Table Access

Balanced or nearly balanced tree structures are advantageous for file access because the resulting search effort is limited to approximately log n key comparisons for files of n records. The keys are available in lexicographic order within the tree structure, and the cost of maintaining files after file additions or deletions is also of order log n. However, faster file-access methods exist that ideally require only a single key transformation to find a stored record, or to add or delete a record. These methods are known as *key-to-address transformations*, or *scatter storage*, or *hash-table accessing*. [29–33]

To obtain file access with a hash-table system, the search key K_i is transformed into a hash-table address using a hashing or hash function h; the table address $h(K_i)$ stores either the record information corresponding to key K_i, or a pointer to an address in the main file where the corresponding record information is located. To be useful in file accessing, hash function h must be able to distinguish similar but nonidentical keys, such as *SAND* and *SANE*, by providing different hash-table addresses. For this reason, hash functions should be chosen that destroy the lexicographic key order and place records with nearly identical keys in different areas of the file. Thus normal file order is lost, and, without additional file sorting, it is impossible to read out, or print, the stored records in key-value order, or to find or delete rec-

ords with the smallest or largest keys. Similarly, range searches designed to retrieve all records within a certain key range are difficult to implement. In principle, hashing is thus more efficient than tree access for finding particular stored records, or for adding or deleting specified records. However, the hash process cannot easily be used for operations that depend on file order.

Ideally, the size of a hash-table is larger than the number of different record keys available for the file under consideration. In these circumstances distinct hash-table addresses may correspond to distinct record keys. When the size of the table is large compared with the number of distinct keys, a *partial hashing* method may be used in which only parts of keys — for example, the first few key characters — determine the corresponding hash-table addresses. Alternatively, when the occupancy rate of the table is low, substantial space may be saved by deleting the actual key information for the corresponding records from the table. In that case, the assumption is made that each hash-table address automatically contains the information for the correct record.

However, the hashing function may not be able to distinguish between certain subsets of distinct keys. In that case $h(K_i) = h(K_j)$ for distinct keys K_i and K_j, and a *collision* is said to occur for the records corresponding to these distinct keys because all these records will be placed at the same hash-table address. A useful hash-table method must therefore include two main components:

1. A hash function $h(K)$ that has a low collision probability, given a particular input key distribution, and a given hash table of size M.
2. A collision-resolution technique that provides storage locations and retrieval strategies for distinct colliding records.

Two main collision-resolution techniques are in widespread use. The first relies on pointers stored with each record in the hash table that give the address in an overflow area of the next colliding record, if any. To retrieve a complete set of colliding records corresponding to a particular hash table address $h(K)$, it is then necessary to follow the pointer chain starting at the base address in the hash table. Figure 7.32 shows such a pointer chain system for a hash table of seven addresses (labeled 0 to 6) using a hash function that places record R_n in address $h(R_n) = n \bmod 7$. The assumed record-insertion order is $R_{11}, R_{16}, R_2, R_{19}, R_5$, and R_9.

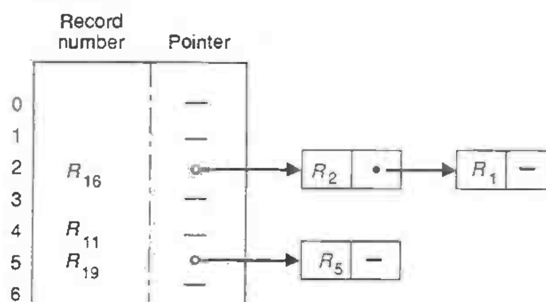
The foregoing collision-resolution technique, *chaining*, provides a conceptually simple hash-table allocation that works well in dynamic situations in which many records are added and deleted. However, the retrieval operations are inefficient when the pointer chains are too long. The chains can be shortened by providing at each position

enough storage locations for a complete *bucket* of records. In that case, the base address $h(K)$ identifies a bucket number, and a pointer chain is followed to an overflow bucket only when the original bucket is full. To locate a particular record in a bucket, either all the records stored in that bucket must be scanned, or each set of colliding records must be stored in increasing or decreasing order in the hash table.

When records are stored in decreasing key order in each pointer chain, the efficiency of unsuccessful searches (for keys that are not in the file) improves — a search carried out with search key K can stop as soon as a smaller key $K(i)$ stored in table position i is found. When a new key K must be inserted into an ordered pointer chain, an interchange process can be used that replaces $K(i)$ with K in the chain, and next inserts $K(i)$ into a new position later in the chain, possibly using additional key interchanges. In Fig. 7.32, an ordered pointer chain starting at table position 2 would give access first to R_{16}, followed by R_9 and R_2 in order.

Another collision-resolution technique consists of starting at the base address $h(K)$ in the hash table and computing the addresses of any overflow storage locations. In the basic *open-addressing* process, a base address b_o is first determined where $b_o = h(K)$. A *probe sequence* is then defined as a permutation of table addresses starting at address b_o and covering the entire hash table of size M. To retrieve a record with key K, the addresses are then scanned in probe sequence order starting with address $b_o = h(K)$, and the probe sequence is followed until the wanted record is found, or an empty slot or "table full" sign is met, in which case the search is unsuccessful. Figure 7.33 shows a program for hash-table access using open addressing with a linear traversal of memory.

Figure 7.32 Hash-table placement with collision resolution by chaining.



The probe sequence is typically constructed by using a hash function $p(K)$ to compute the next table address j from a given address i as $j = (i - p(K))$, and resetting j to $j + M$ when the original j turns negative for a table of size M. In the sample program of Fig. 7.33, the function $p(K)$ is replaced by the constant -1. The use of such linear probe sequences, however, is not recommended because sets of adjacent memory addresses are then filled rapidly, causing a clustering effect in the memory space. In particular, when position i becomes occupied, the probability that position $i + 1$ receives a record is twice as high as before, because now all the records with hash addresses i and $i + 1$ will be placed in table position $i + 1$. The clustering problem is mitigated by using a random probe sequence r_i obtained, for example, with a random number generator. This places records in positions $h(K)+0$, $h(K)+r_1$, $h(K)+r_2,\ldots, h(K)+r_{M-1}$, where r_i represent a permutation of $0, 1,\ldots, M-1$.

Table 7.1 displays typical performance figures for three types of hash-table searches; performance depends on the proportion of filled memory positions (or load factor) α of the table. [34,35] Chaining is the most efficient access method in terms of the number of accessed memory positions. Unfortunately, this technique requires a good deal of extra storage because the pointer chains must be stored and maintained when file items are added and deleted. The open addressing system, which requires no pointers, seems preferable for small memory load factors α. Released memory positions corresponding to deleted records cannot be recovered in an open address system, however, because the probe sequences would be destroyed. Also, when the load

Figure 7.33 Program for hash-table access with open address.

1. [Calculate initial address for search key X]
   $i = d \leftarrow h(X)$

2. [Is location i empty?]
   if K_i is empty
   then $K_i \leftarrow X$ and exit

3. [Increment and test]
   $i \leftarrow i + 1$
   if $i > M$
   then $i \leftarrow 1$
   if $i = d$
   then "overflow" and exit
   go to step 2

factor increases, some probing methods become very expensive, in part because of the previously mentioned clustering problems. The use of random probe sequences based on primary hash functions $h(X)$, followed by secondary hash functions $p(X)$ when needed, provides relatively efficient file access, especially for ordered chains where the probability is high that successful or unsuccessful searches can be terminated early.

No collision-resolution method is fully satisfactory when the hash table is nearly full. In that case, a rehashing process could be used that allocates space in a larger table. Rehashing is conceptually simple when the old and new tables are allocated in disjoint regions of the memory space. When only a limited amount of memory is available to be added to the original table, an "in-place" rehashing method can be used (see Fig. 7.34).

The program of Fig. 7.34 performs a linear sweep of the original table addresses, starting at address 0 and ending at address $M_1 - 1$. When an entry is found that has not yet been rehashed, its key K is used with hash functions $h(K)$ and $p(K)$ to generate a probe sequence defined as $(h(K) - ip(K)) \bmod M_2$, where $M_2 > M_1$ represents the new table size. The probe sequence is followed, and the entry corresponding to key K is put into the first table position in the sequence not yet rehashed. Next the displaced entry, if any, must be rehashed, possibly leading to a cascade of displaced entries. When no displaced entry is found, the original linear sweep of the old table is resumed until

Table 7.1 Typical performance figures for hash-table searches (average number of memory positions accessed for successful (S) and unsuccessful (U) searches given load factor α). [34]

Separate Chaining of Overflow Records			Open Addressing with Linear Probe			Random Probing with Double Hash Function		
$1+\alpha/2$ S			$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ S			$-\frac{1}{\alpha}\ln(1-\alpha)$ S		
$\alpha+e^{-\alpha}$ U			$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ U			$\frac{1}{1-\alpha}$ U		
α	**S**	**U**	α	**S**	**U**	α	**S**	**U**
.10	1.050	1.005	.10	1.056	1.118	.10	1.054	1.111
.30	1.150	1.041	.30	1.214	1.520	.30	1.189	1.429
.50	1.250	1.107	.50	1.500	2.500	.50	1.386	2.000
.70	1.350	1.197	.70	2.167	6.060	.70	1.720	3.333
.90	1.450	1.307	.90	5.500	50.500	.90	2.558	10.000

memory position M_1 is reached, at which point the algorithm terminates.

Obviously, completely reorganizing memory as in Fig. 7.34 is too onerous to undertake frequently. An alternative method to maintain hash tables in dynamic files is provided by the *extensible* or *dynamic* hashing methods, which largely avoid repositioning hash-table entries. [36–39] An index, or directory, is interposed between the key terms and the final record addresses in the main file. The hashing operation then identifies an address in the index rather than in the main

Figure 7.34 In-place rehashing process.

Step	Action
1. [Initialize]	$i \leftarrow 0$ let X represent an empty table entry
2. [Is table entry $T(i)$ rehashable?]	if $T(i)$ is not empty, nor already rehashed then exchange $T(i) \leftrightarrow X$ go to step 4 else continue enumeration at step 3
3. [Advance to next table entry]	$i \leftarrow i + 1$ if $i = M_1$ terminate else go back to step 2
4. [Compute hash function values]	$K \leftarrow$ key(X) $a \leftarrow h(K)$ $c \leftarrow p(K)$
5. [Find unrehashed location]	if $T(a)$ is already marked rehashed then compute new probe address $a \leftarrow (a - c) \bmod M_2$ until finding an address a such that $T(a)$ is not marked rehashed
6. [Insert in table]	exchange $T(a) \leftrightarrow X$ mark $T(a)$ rehashed; entry X now becomes displaced entry
7. [Processed displaced entry]	if displaced entry is empty go to step 3 to resume enumeration else go to step 4 to rehash it and place it in table

file, and each index position in turn points to the main file addresses — normally buckets — where the corresponding records are located. Instead of changing the main file configurations as new records are added, the index is made to grow: Thus when a given bucket overflows, the index entry for that bucket is changed to produce two entries, corresponding to two new buckets replacing the single original bucket. Similarly when the file shrinks, two or more buckets are merged into a single bucket by appropriate index-entry transformations.

Consider, as an example, a situation in which the records are stored in buckets of size b, and an index of size 2^d is used to gain access to the records. Given a key K, the appropriate bucket address is obtained in two steps:

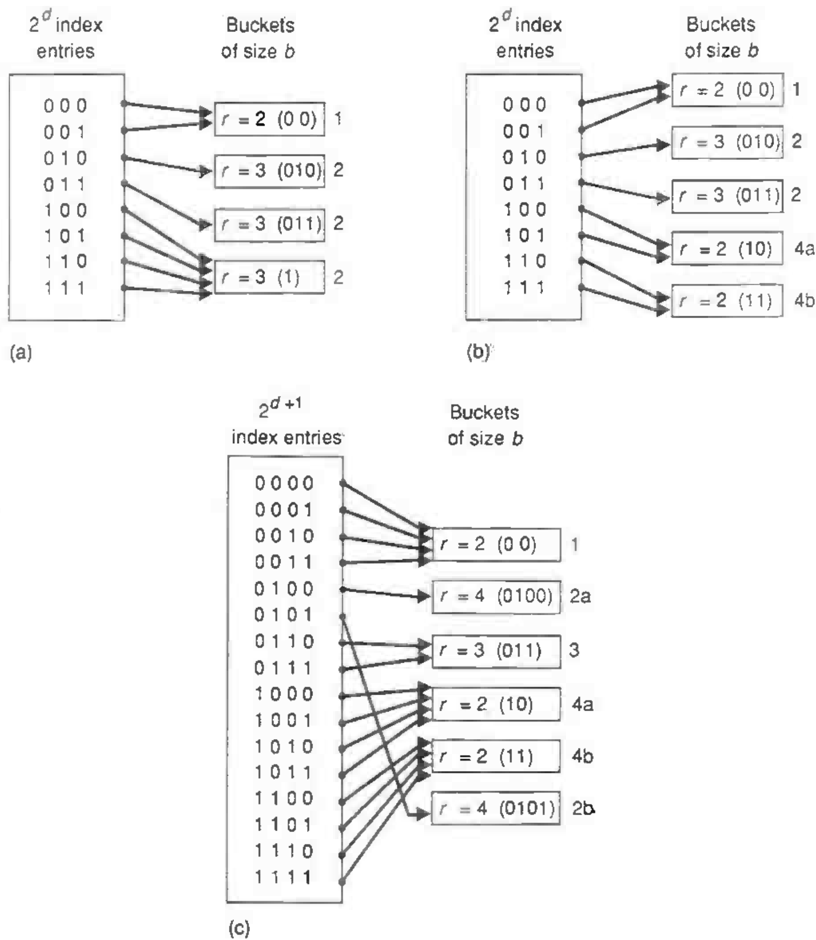
1. A hash function $h_d(K)$ is computed that produces a binary number in the range $0, 1, \ldots, 2^d - 1$.
2. The binary number identifies an index entry that points to the bucket containing the record corresponding to search key K.

An example of this situation is shown in Fig. 7.35(a), where an index of size $2^d = 8$ gives access to one of four storage buckets. Each bucket exhibits a local depth $r \leq d$, where r identifies the key length associated with a given bucket. In the example, enough records were found for hash address $h(K) = 010$ to allocate a separate bucket of local depth 3 to the corresponding records. On the other hand, very few records were associated with index entries 100, 101, 110, and 111. All these records are therefore assembled in a single bucket (number 4) of local depth 1 characterized by the fact that the first, left-most index address digit is a 1.

Three cases must be distinguished when inserting new records into the file. If bucket B corresponding to key value K is not full, a new record is simply inserted into the proper bucket. If bucket B is full, a new bucket B′ must be created to accommodate the new record. If the local depth r of B is smaller than the depth d of the index, it is sufficient to change the pointers of certain index entries to point to the newly created bucket B′, and possibly to reallocate some records from bucket B to B′ in accordance with the augmented index entries of depth $r + 1$ of the corresponding records. This operation is illustrated in Fig. 7.35(b), where bucket number 4 of depth 1 (index entry 1..) is split into two buckets 4a and 4b of depth 2, identified by index entries 10 and 11, respectively. In rare cases, reallocating the records to the split buckets causes bucket overflow when all records wind up in one of the two split parts. In such cases, the bucket splitting must be repeated.

When the bucket to be split has local depth equal to the depth of the index ($r = d$), the index itself must be increased in size. This is done by doubling the index size as shown in Fig. 7.35(c), and creating an index of 2^{d+1} entries of depth $d + 1$. At this point, the bucket of depth $r = d$ can be split into two parts of depth $r + 1 = d + 1$, as previously described. In Fig. 7.35(b), bucket 2 with index entry 010 was split by dou-

Figure 7.35 Memory allocation in dynamic hashing. (a) Initial bucket allocation. (b) Bucket allocation after split of bucket 5. (c) Bucket allocation after split of bucket 2.



bling the index size and making available the new bucket addresses 0100 and 0101, assigned to buckets 2a and 2b, respectively. When the directory size is changed, many of the index pointers must be reallocated, but only the records in the split bucket may have to be moved.

After record deletion, a reverse process can be used to merge bucket pairs that jointly contain fewer than b records. If the merged buckets are the only ones of depth $r = d$, the index size can be halved and the depth reduced to $d - 1$.

The use of extensible hashing prevents large-scale record reallocation in dynamic file environments, but does not resolve the collision problem caused by a poor choice of hash function $h(K)$. Ideally, it should be easy to compute $h(K)$, and few collisions should occur. When collisions are detected, an efficient collision-resolution technique should be available that does not deteriorate with increasing memory-load factors. Many different methods are suggested in the literature to implement hashing functions, two of the simplest being the multiplication and the division methods. [34] In the division method, the key value, or some function of the key value, is divided by a prime number, and the remainder after division is transformed into the needed address. The multiplication method consists of multiplying the key by itself and using the middle digits of the product as record addresses. As shown in Fig. 7.36, this method avoids potential collisions by assigning very different table addresses to keys that differ only by a single low-order digit.

Figure 7.36 Typical hashing method using key multiplication. Assumed hash-table size is 64 positions.

Record *M*: Key 11 01 01 00

Key squared 1 0 1 0 1 [1 1 1 1 0 0] 1 0 0 0 0

Decimal address corresponding to $(1\ 1\ 1\ 1\ 0\ 0)_2$ =

32 + 16 + 8 + 4 + 0 + 0 = 60

Record *N*: Key 11 01 01 01

Key squared 1 0 1 1 0 [0 0 1 0 0 1] 1 1 0 0 1

Decimal address corresponding to $(0\ 0\ 1\ 0\ 0\ 1)_2$ =

0 + 0 + 8 + 0 + 0 + 1 = 9

Many hash functions have been proposed over the years, some specifically applicable to numeric or alphabetic key values. For example, the division method can be used with alphabetic keys by taking a numerical representation of each word, consisting of the concatenated order numbers of the alphabetic characters of the word, dividing this by the table size, and using the remainder of the division as an address for the corresponding record information. [40] Another interesting method is the *dispersed hash* strategy, where r different hash functions are applied to the key information, each producing a 1-bit address in a binary word of length n. The corresponding r bits of the word are then set equal to 1, producing a binary character string of length n of which r bits are set equal to 1. By using appropriate values of n and r as a function of file size, the probability of collision can be made very small. The dispersed hash strategy can construct bit strings that are valid for several distinct search keys simultaneously; it is examined in more detail later in this chapter.

In summary, hash tables afford fast file access when the collision probability is low, and provide for file additions and deletions. However, accurate guesses must be made in advance about required table sizes because restructuring large hash tables is expensive. Further, the hashing process destroys file ordering. When stored records must be processed in order, balanced tree procedures are usually preferable.

7.8 Indexed Searches for Multikey Access

The file-access procedures treated up to now apply primarily to single-key searches, in which the query consists of a single content term attached to a record, or of the value of a single attribute, such as a customer name in a file of telephone subscribers. For single-key searches, the record file can be ordered according to the values of the available search key — for example, the alphabetic order of the names of telephone subscribers. Alternatively, the index that gives access to the main file can be ordered, as in the previously described tree-access methods.

When multikey searches must be performed, a *principal key* can sometimes be identified, and the file can be ordered in accordance with the values of that key. When the principal key is used as part of a search statement, the subsection of the file corresponding to the given principal key value can then be isolated and subjected to a separate search based on the values of any *secondary keys* also included in the search query. For example, to find the records of telephone subscribers named SMITH who also live on STATE STREET, one isolates all the Smiths using the alphabetic file order, and then scans the output looking for those individuals living on State Street. When only second-