1979

Hash-Binary Search: A Fast Technique for Searching an English Spelling Dictionary

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Vincent Y. Shen

Report Number:
79-304

Hash-Binary Search: A Fast Technique for Searching an English Spelling Dictionary

Douglas Comer

Vincent Y. Shen

TR 304

Computer Sciences Department
Purdue University
W. Lafayette, IN 47907

April 1979

Abstract

When a document is prepared using a computer system, it can be checked for spelling errors automatically and efficiently. This paper presents the hash-binary method for searching a static table and applies it to searching an English spelling dictionary. Analysis shows that with only a small amount of space beyond that required to store the keys, the hash-binary search method performs better than either hashing with open-addressing or binary search. Experiments with a sample dictionary verify the results. We also present extensions to account for skewed frequencies of access as well as methods for testing alternative hashing functions.

Keywords and phrases: searching, spelling, hashing, binary search, English dictionary

CR Categories: 3.74, 4.34

I. Introduction

When text is typeset using a computer-based system, it can also be checked for spelling errors automatically. Several different methods of spelling error detection have been proposed. Many of these methods are "frequency-based". For example, a simple technique would be to sort all words in a document and print those which occur infrequently. The list is then checked for misspellings, but habitually misspelled words would escape unnoticed. Morris et al [MORR75] study statistical properties of English words, and describe an algorithm to catch possible typographical errors by examining the relative frequency of trigrams (3-letter combinations). McMahon et al [MCMA78] summarize the statistical methods and describe their use in the UNIX system document preparation aids. One technique applies t hashing functions to each word in an English spelling dictionary to obtain t integers in the range 1 through m. A table of m bits is maintained and the ith bit is set to one if and only if there exists an English word for which one of the hashing functions generates the integer i. To check a candidate word, the system hashes it with the same t hashing functions and checks all t locations in the table to verify that they are all ones. With properly chosen hashing functions and table size, this method can guarantee not to accept a misspelling with a predetermined high probability.

A more conventional method would be to maintain the list of

words in the spelling dictionary in the machine and look up the candidate words. Knuth [KNUT73] comments on the storage of the dictionary, and suggests an organization intended to reduce space requirements. One way to reduce the storage requirement is to remove the suffixes from the words and store only the stems [KNUT73, KERN78]. When a word is to be looked up, its suffixes are removed by the same method, and only the stem is checked. An obvious disadvantage of this method is the possible acceptance of words with illegal stem and suffix combinations. For example, the typo "computions" would pass the spelling test because removal of the suffix "ions" leaves the valid stem "comput".

Basically, we view spelling error detectors as an aid to humans, and as such we do not mind taking the time to look over a short list of possible misspellings as much as the annoyance of finding unreported misspellings in the final document. Thus, we wish the spelling error detector to err only by rejecting some correctly spelled words, but never by accepting misspelled words. This viewpoint requires the use of a dictionary with complete English words and their inflections. However, the dictionary may still be kept small by including only commonly used words. While we can not give the desirable size for such a dictionary, it is clear that the philosophy of "the bigger, the better" does not apply. A complete dictionary of several hundred thousand words has almost no value for catching spelling errors -- misspellings frequently turn out to be obscure or archaic words which appear in the dictionary. Even a dictionary of 40,000 words may be too

large to be useful. For example, the 40,000 word dictionary maintained by the Purdue University Computing Center includes the words "de", "hod", "ila", "pul", and others which would probably be typos in technical prose[1].

Since writers use most words infrequently and a few words very frequently, the best scheme seems to be:

1. Start with a small core of, say, 10,000 commonly used English words.
2. Add new words to the dictionary only as users request them.

This way, an appropriate dictionary evolves without unnecessary or obscure words. A variation of such a scheme was used at Bell Labs to derive a spelling dictionary that is one-third the size of their original [KERN78].

A small spelling dictionary may fit into main memory and can be searched very quickly. The words themselves usually occupy a large portion of the available space, leaving only a little extra space for the program and data structures. This paper explores searching techniques in the context of a small dictionary and presents a method which exploits a small amount of extra space to reduce access time. The method also applies to other searching situations such as checking the static directory of an inverted file organization.

---

[1] Any spelling errors in this paper can be attributed to the Purdue dictionary.

II. The Hash-Binary Search Method

The dictionary to be searched is defined as a static set of n keys, each of which may be stored in one computer word without abbreviation. The memory space available to store this dictionary for searching is M words, M > n. The reader is referred to [SHEI78] for remarks on partitioning the dictionary by word length.

The dictionary, being static, may be preordered in any way and the k extra words of memory space (k = M-n) may be used to assist the search process. For example, one may sort the entire dictionary of n keys and search it using the binary search method, leaving the k words unused. Another method may be to consider the memory space as a hash table of size M, whose loading factor is n/M. Many other methods have been reported in the literature [KNUT73, SHEI78, COME79].

This paper discusses a new search method, called the hash-binary search method, that uses the k extra words as buckets for a hashed file. That is, a hashing function H(x) is defined to map any key x in the key space S to the set of integers (1, 2, ... , k). Word i ($1 \leq i \leq k$) of the memory space is used to store the beginning location (Bi) of the ith bucket. The keys in the ith bucket are stored in the ti consecutive locations from Bi through Bi+1 - 1. These keys may be sorted to allow binary or some other search method inside the buckets.

A search process can be modeled by an access tree as shown in Figure 1 [YA077]. Each step in the search process advances one level from the root toward the leaves along a branch. The search process terminates when the item is found or when a leaf is reached. Thus the performance of a search method may be analyzed in terms of the number of levels in the access tree model, assuming one memory probe is required to advance one level.

The hash-binary search method has an access tree with k branches at the root level (Figure 2). A hashing function H is used to select one of these k branches during the first step of searching by mapping the key to an integer between 1 and k. All keys that are mapped to i ($1 \leq i \leq k$) are considered to be placed in the ith bucket. The keys in each bucket are sorted so that they may be searched using the binary search method. Thus we have balanced binary trees starting at the second level of the access tree model.

The access tree model shows that the best performance is achieved when the tree is balanced at the root level. Hence the performance of the hash-binary method is best when we use a hashing function that distributes an equal number of keys into each bucket. Although such a hashing function may not be easy to find, the following sections of this paper show that a simple remainder hashing function performs extremely well in practice. This is due to the fact that even if the number of keys

distributed to a bucket differs from the optimum by a factor of two, only one additional probe (compared with the optimal case) is needed to search the bucket. The following sections of this paper discuss the performance of the hash-binary method, the selection of the hashing function, and other implementation considerations. We will demonstrate that the hash-binary technique has advantages over other well-known methods for English dictionary searching.

III. Performance Analysis of the Hash-Binary Search Method

The performance of a search method is normally analyzed in terms of the number of comparisons made while locating a key. Since each comparison requires one memory reference, or probe, we will use the number of probes as our measure. Although the number of probes may vary from key to key for a given dictionary, we are interested in the average number of probes which is represented by Pav, assuming that all keys are equally likely to be accessed. Of course, not all words in the dictionary are equally likely to be accessed[2]; we will discuss alterations to account for skewed access frequency distributions later. In addition to analytical comparisons, this section also presents experimental results using an English spelling dictionary.

The general problem of data storage and retrieval for static sets of keys has received wide attention in the literature. Knuth [KNUT73] provides a summary of many basic techniques:

Sequential search - each probe eliminates one key

Binary search - each probe eliminates half of the keys

Interpolation search - each probe eliminates more than half of the keys in a uniform (or near uniform) distribution

Hash table search - each probe eliminates most of the

---

[2] Kucera and Francis [KUCE67] report that the 10 most frequently used words account for 24.3% of all text.

remaining keys (given sufficient memory)

Partial index - a small, auxiliary data structure is searched to eliminate all but a subset of the keys, which is then searched using one of the above methods

Other techniques may be found in [SHEI78, SPRU77, COME79].

The performance of these methods has been analyzed, and closed-form expressions for the average number of probes have been derived assuming that all keys are equally likely to be searched. Since the sequential search method expects to probe half of the keys in the dictionary, we have

$$Pav(sequential) = n/2 \qquad (1)$$

The performance of the binary search method is better for a sorted set of keys and has a logarithmic behavior:

$$Pav(binary) = \lg n - 1 + (\lfloor \lg n \rfloor + 2)/n \qquad (2)$$

Note that the symbol lg represents a base 2 logarithm. It may be possible to improve the performance of the binary search method when the access frequency is non-uniform. One technique, called the median-split search, uses two probes to eliminate half the keys [SHEI78]. The worst-case number of probes is still logarithmic, but keys with a high access frequency are tested early.

When the key values are uniformly distributed over the key space S, the interpolation search method requires [YAO76]

$$Pav(interpolation) = C1\ (\lg\ (\lg\ n)) + C2 \qquad (3)$$

In practice, however, the constants C1 and C2 make it very expensive. This is especially true when the key values have a highly skewed distribution over S like those in a spelling dictionary [KNUT73].

The performance of the hash table search depends on the loading factor a, a = n/M, which represents the fraction of the memory space occupied by the keys. It also depends on the hashing function used, as well as the scheme used to resolve collisions. If the open-addressing scheme is used to resolve collisions, the performance of a "good" hashing function is given by the following formula:

$$Pav(hash) = (1 + 1/(1 - a))/2 \qquad (4)$$

The performance is worse if the item can not be found. That is, for an unsuccessful search,

$$Pav(hash)' = (1 + 1/(1 - a)^2)/2 \qquad (4')$$

As the hash table fills (i.e., a $\rightarrow$ 1), the performance deteriorates. However, it is possible to rearrange the keys in the table so that the performance is never worse than O(lg n) [RIVE78, GONN78]. Such rearrangement introduces computational overhead which must be considered before application.

The average number of probes to access a word in a dictionary using the hash-binary method discussed in the previous section depends on the distribution of words into buckets and the method of searching within a bucket. Assuming a uniform distribution of the keys into buckets, using a binary search

within a bucket leads to an estimate of

$$Pav(\text{hash-binary}) = 1 + \lg(n/k) - 1 + (\lfloor \lg(n/k) \rfloor + 2)/(n/k) \qquad (5)$$

Note that we count the computation of the hashing function and the subsequent bucket selection as one probe. If the item can not be found, one additional probe is needed to account for the worst case using binary search within the buckets.

The performances of these methods may be compared by assuming that every key in the dictionary is equally likely to be accessed. For the case with interpolation search, it is also assumed that the key values are uniformly distributed over the key space. The latter assumption may be relaxed slightly for the hash table method in the sense that a good hashing function will distribute hashed values uniformly, even though the original keys may not be uniformly distributed. Under the same assumption, the keys will be uniformly distributed into buckets by the hash-binary method. For a spelling dictionary of 16,949 entries (approximately 2^{14} words), a sequential search requires about 2^{13} probes from Equation (1), and a binary search requires about 13 probes (Equation (2)). The number of probes needed for the hash table and the hash-binary method for a given loading factor may be found using Equations (4) and (5). Table 1 compares these two methods for several different loading factors.

Table 1

Comparison of the hash table and the hash-binary methods

Loading factor	Hash successful	Hash-binary successful	Hash unsuccessful	Hash-binary unsuccessful
0.50	1.50	2.00	2.50	<3
0.60	1.75	2.25	3.62	<3
0.70	2.17	2.56	6.06	<3
0.80	3.00	3.00	13.00	<4
0.90	5.50	3.73	50.05	<5
0.95	10.50	4.57	200.50	<6
0.99	50.50	6.74	5000.50	<8

Table 1 shows that the hash-binary method performs better than the hash table method when the number of keys occupy over 80% of the memory space. Equations (4) and (5) may be used to determine that the cross-over is actually reached when $k \leq .249n$, or when the memory is at least 80% full ($a \geq .80$). Since there is no accurate formula to represent the average binary search performance within buckets when the bucket sizes are very small, we have not used Equation (2) for some entries in Table 1. When the loading factor is less than 0.80, the average performance for the successful hash-binary search has been computed accurately from the individual bucket sizes. While accurate values for unsuccessful search times are difficult to calculate, the bounds given in the last column of the table clearly demonstrate the superiority of the hash-binary method when the loading factor is greater than 0.80. Equations (2) and (5) show that the hash-binary method is better than the binary search method when $k > 2$. Therefore, the hash-binary method performs better than either binary search or hashing, even if extra effort is expended to rearrange the keys so that the worst-case search time is

limited to O(lg n) [RIVE76, GONN78].

The hash-binary method is certainly slower than the so-called perfect hashing method, which guarantees one probe per lookup. Sprugnoli proposes several perfect hashing functions [SPRU77], but they all require a large amount of space and preprocessing time. In general these methods apply only to static sets of less than 100 keys.

When the access frequency is not uniform over the set of keys, the median-split search method tends to perform better than the binary search [SHEI78]. However, it still exhibits logarithmic performance. In any case it is possible to modify the hash-binary method to use the median-split technique when searching within a bucket. If the hashing function distributes the keys with high access frequencies uniformly into the buckets, the hash-median-split method will perform better than the straight median-split search.

The partial index methods as reported in [COME79] all have more levels and less fan-outs in the access tree than the hash-binary method. Thus, one would expect the average number of probes for these methods to be higher than that of the hash-binary for a well chosen hashing function.

We applied the hash-binary method to our English spelling dictionary of 16,949 entries using a very simple hashing function

as defined below:

$$H(x) = x \bmod p + 1 \qquad (6)$$

where p is the largest prime number less than or equal to k. Table 2 shows the performance results for several different values of p (actual number of buckets). It is surprising to see that such a simple hashing function performs very well.

Table 2
Performance of the hash-binary methods using remainder hashing

		Avg. number of probes		Max. number of probes	
Loading factor	Number of buckets(p)	Actual	Ideal (Table 1)	Actual	Ideal
0.50	16943	2.49	2.00	6	2
0.60	11299	2.62	2.25	6	3
0.70	7253	2.81	2.56	5	3
0.80	4231	3.13	3.00	6	4
0.90	1879	3.80	3.73	6	5
0.95	887	4.61	4.57	7	6
0.99	167	6.75	6.74	9	8

IV. Improving the Performance of the Hash-Binary Search Method

The hashing function used to obtain the excellent results shown in Table 2 computes the bucket number by taking a key x modulo some prime number p, $p \leq k$. While the remainder hashing function performs well on our sample English dictionary, it may not do as well on other static sets of keys. Instead of looking for, and testing, other hashing functions, this section discusses a simple extension to the basic remainder hashing which may be easily tested for performance improvement before actual application.

Let H1(x) be a remainder hashing function with prime number p, and H2(x) be another hashing function. A new hashing function, which is a linear combination of the two functions, is defined as

$$H(x) = \lfloor c1\ H1(x) + c2\ H2(x) + c3 \rfloor \qquad (7)$$

It may be that a good choice of c1 and c2 makes H(x) better than using either H1 or H2 alone. The constant c3 is set after c1 and c2 are chosen to make the minimum value of H(x) zero.

The behavior of H(x) as defined in Equation (7) may be observed by considering the distribution of keys over a two-dimensional space formed by H1 and H2 as shown in Figure 3. A linear combination of H1 and H2 is represented by the projection of these points onto a straight line passing through the origin at an angle θ. The line is further divided into k

equal segments representing the k buckets. Those points whose projections lie within the same segment are placed in the same bucket (Figure 4). We see that using H1 alone implies c2 = 0, which is a vertical projection line. On the other hand, using H2 alone implies a horizontal line with c1 = 0. Other choices of c1 and c2 produce different projections that lead to different distributions into buckets.

The problem of finding a projection line that has the best performance for a given set of keys and two hashing functions has been investigated [COME79a], which reports an $O(k \cdot n^2 \lg(n \cdot k) + n \cdot k^2)$ algorithm to determine which values of c1 and c2 produce the best line. There is no a priori evidence, however, that finding the optimum angle of projection will significantly improve the performance over the simple remainder hashing method. The following strategy should be used before applying the expensive optimization algorithm:

1. Compute the ideal performance for the hash-binary method using Equation (5).
2. Compute the bucket sizes using the remainder hashing function (c2 = 0), and compute the expected performance using the bucket sizes and Equation (2).
3. If the figure obtained by Step 2 is unacceptably higher than that by Step 1, try several different projections whose angles range between 0 and 180 degrees.
4. If all samples of Step 3 yield poor performance, try different hashing functions.

In the unlikely event that Step 3 is needed, a projection is determined by choosing c1 and c2. For any given c1 and c2, the size of the ith bucket ti is determined by counting the number of keys whose hashed value lies between $(i - 1) \cdot \max(H(x))/k$ and $1 \cdot \max(H(x))/k$. Thus r different projections may be evaluated at the same time by making two passes over the dictionary using only space for an array of $r \cdot k$ integers. The performance of the hash-binary method appears to be a continuous function over the angle of the projection line for our sample dictionary. We conjecture that an interval bisection method may be used to find a good angle of projection.

The sampling of different projections or even the optimum algorithm may not work if the functions H1 and H2 place all the keys on a straight line in the two-dimensional space (Figure 5). One would expect, however, that a different choice of H1 and H2 could improve the performance significantly.

Consider a set of keys which has poor bucket distribution under remainder hashing using a prime number p. For the ith bucket to be heavily loaded ($ti >> tj$ for $i \neq j$), there are ti keys of the form $y \cdot p + i + 1$, with y being an integer. These keys must be spread out, probably every p positions, over the key space S. A different simple hashing function that may perform well in such situations divides the entire key space into q equal sized subspaces. This method, called quotient hashing, is defined as

$$H(x) = \lfloor x \;/\; \lfloor S \;/\; q \rfloor \rfloor \qquad (8)$$

Each of the q subspaces will receive an equal number of keys if the keys are uniformly spread out over S. Intuitively, the remainder hashing function performs well when there are clusters of keys in the key space, but performs poorly when the keys are spread out in the key space cyclically.. The quotient hashing function, on the other hand, tends to have a completely opposite behavior. Thus, a linear combination of these two methods, called the remainder-quotient hashing function, may exhibit the nice performance characteristics of both.

The application of the quotient hashing function requires special consideration. Its performance is very sensitive to the internal representation of the keys. For example, suppose the English words are coded as strings of characters in ASCII. Each character occupies only 26 out of the 128 possible values, so treating an ASCII string like an integer produces large gaps in the key space. This, in effect, causes undesirable clustering. Therefore, one should encode the keys carefully before applying quotient hashing or poor performance will result.

V. Conclusions

We have presented the hash-binary search method which hashes a key into a bucket and applies a binary search within the bucket. When only a small amount of space remains after storing the keys, hash-binary search performs better than either binary search or hashing under the assumption that the hashing functions distribute the keys uniformly. Furthermore, hash-binary search is robust in the sense that it is not especially sensitive to the hashing function performance--the search times differ from the ideal situation by at most one comparison even when some bucket is twice as full as the average.

We demonstrated the advantages of the hash-binary method using a sample English dictionary of 16,949 entries. We also suggested several refinements in order to apply the method to other static sets of keys and to sets of keys with different access frequencies.

REFERENCES

[COME79] D. Comer, "English Dictionary Searching With Little Extra Space" (to appear Proc. NCC 79).

[COME79a] D. Comer and M. O'Donnell, "Geometric Problems with Application to Hashing", TR-303, Computer Science Department, Purdue University, W. Lafayette, IN., 1979.

[GONN78] G. Gonnet, "Expected Length of the Longest Probe Sequence in Hash Code Searching." TR CS-RR-78-46, Department of Computer Science, University of Waterloo, Waterloo, Canada, 1978.

[KERN78] B. Kernighan, M. Lesk, and J. Ossanna, "UNIX Time Sharing System: Document Preparation," The Bell System Technical Journal 57:6 (July-August 1978), 2115-2135.

[KNUT73] D. Knuth, The Art of Computer Programming, Vol 3: Sorting and Searching, Addison Wesley, 1973.

[KUCE67] H. Kucera and W. Francis, Computational Analysis of Present-Day American English, Brown University Press, 1967.

[MCMA78] L. E. McMahon, L. L. Cherry, and R. Morris, "Statistical Text Processing," The Bell System Technical Journal 57:6 (July-August 1978), 2137-2154.

[MORR75] R. Morris and L. Cherry, "Computer Detection of Typographical Errors," IEEE Trans. on Professional Communication PC-18 (March 1975), 54-56.

[RIVE78] R. Rivest, "Optimal Arrangement of Keys in a Hash Table", J. ACM 25:2 (April 1978), 200-209.

[SHEI78] B. Sheil, "Median Split Trees: A Fast Lookup Technique for Frequently Occurring Keys," Comm. ACM 21:11 (Nov. 1978), 947-958.

[SPRU77] R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieval Method for Static Sets," Comm. ACM 20:11 (Nov. 1977), 841-850.

[YAO76] A. Yao and F. Yao "The Complexity of Searching an Ordered Random Table," Proc 17th IEEE Symposium FOCS (Oct. 1976), 173-177.

[YAO77] S. Yao, "An Attribute Based Model for Database Access Cost Analysis," ACM TODS 2:1 (March 1977), 45-67.

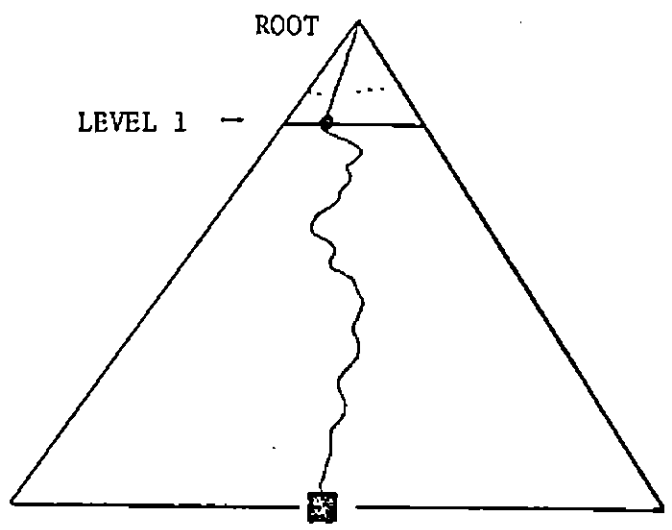
Figure 1. An access tree model of searching with a path from the root to the desired leaf.

Figure 2. An access tree with a k-way decision at the root and k balanced binary subtrees.

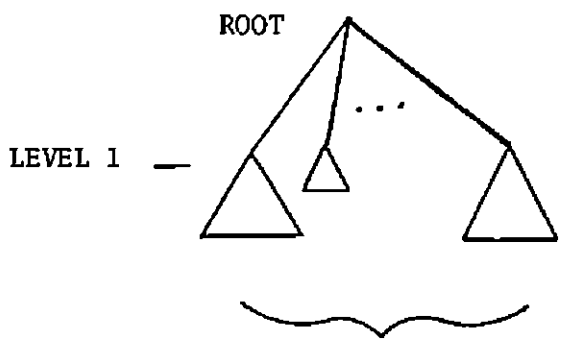
Figure 3. A set of keys plotted in 2-dimensional space using $(H_2(x), H_1(x))$ as the coordinates of key x.

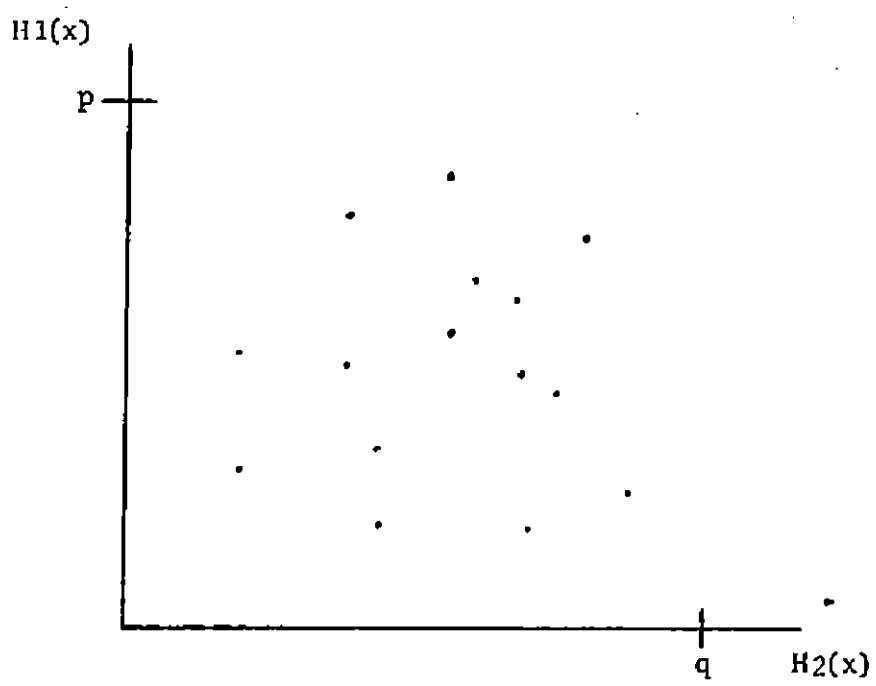
Figure 4. A line of projection at angle θ and the set of keys projected into k buckets which are marked off uniformly along the line of projection.

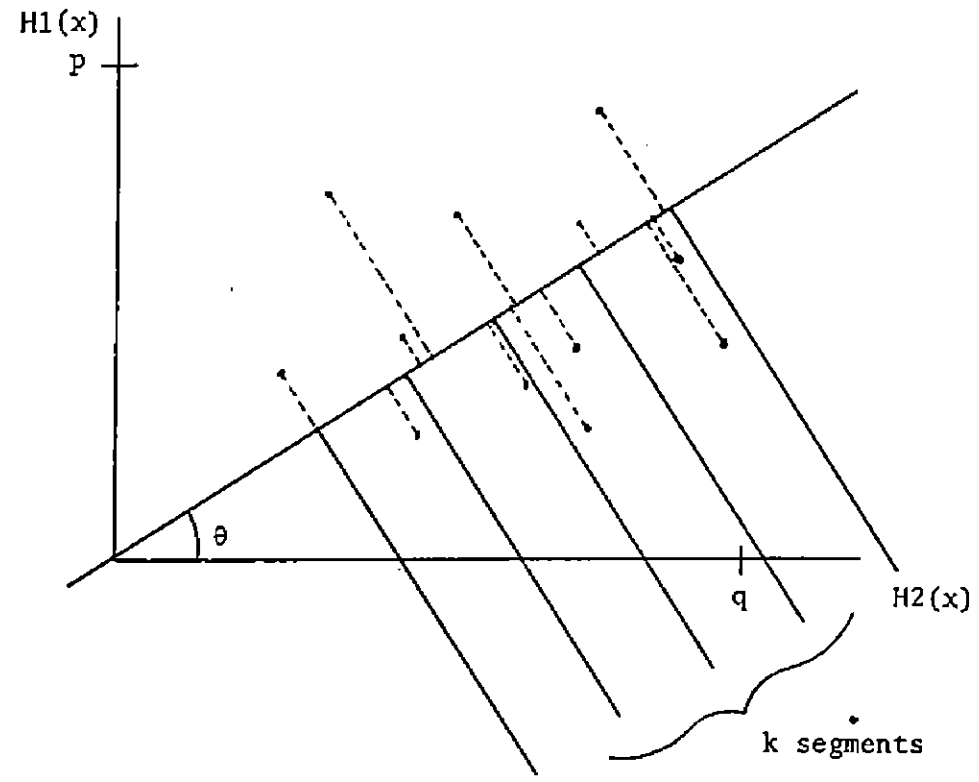
Figure 5. A worst case for projection. The set of keys forms a co-linear set when plotted in 2-dimensional space using hash functions $H_1(x)$ and $H_2(x)$; almost all projections have equal search times.



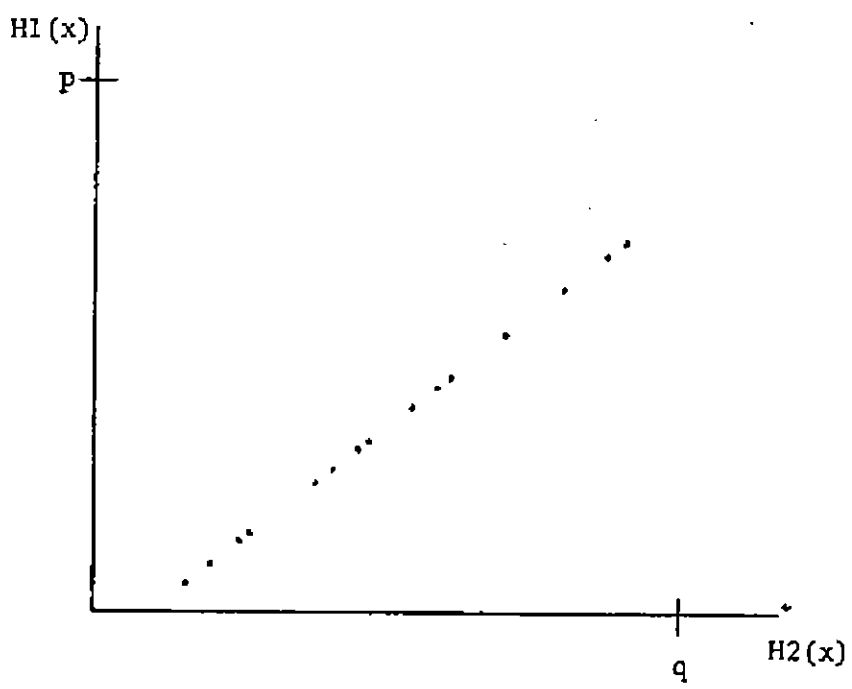
F1


ROOT
LEVEL 1
k balanced binary
trees


H1(x)
p
q
H2(x)



F4


H1 (x)
p
q
H2 (x)