Approximate Dictionary Queries

Gerth Stølting Brodal[1]* and Leszek Gąsieniec[2]**

[1] BRICS***, Computer Science Department, Aarhus University,
Ny Munkegade, DK-8000 Århus C, Denmark.
[2] Max-Planck Institut für Informatik, Im Stadtwald, Saarbrücken D-66123, Germany.

Abstract. Given a set of n binary strings of length m each. We consider the problem of answering d–queries. Given a binary query string α of length m, a d–query is to report if there exists a string in the set within Hamming distance d of α.
We present a data structure of size $O(nm)$ supporting 1–queries in time $O(m)$ and the reporting of all strings within Hamming distance 1 of α in time $O(m)$. The data structure can be constructed in time $O(nm)$. A slightly modified version of the data structure supports the insertion of new strings in amortized time $O(m)$.

1 Introduction

Let $W = \{w_1, \ldots, w_n\}$ be a set of n binary strings of length m each, i.e. $w_i \in \{0,1\}^m$. The set W is called the *dictionary*. We are interested in answering d–queries, i.e. for any query string $\alpha \in \{0,1\}^m$ to decide if there is a string w_i in W with at most Hamming distance d of α.

Minsky and Papert originally raised this problem in [12]. Recently a sequence of papers have considered how to solve this problem efficiently [4, 5, 9, 11, 15]. Manber and Wu [11] considered the application of approximate dictionary queries to password security and spelling correction of bibliographic files. Their method is based on Bloom filters [2] and uses hashing techniques. Dolev *et al.* [4, 5] and Greene, Parnas and Yao [9] considered approximate dictionary queries for the case where d is large.

The initial effort towards a theoretical study of the small d case was given by Yao and Yao in [15]. They present for the case $d = 1$ a data structure supporting queries in time $O(m \log\log n)$ with space requirement $O(nm \log m)$. Their solution was described in the cell-probe model of Yao [14] with word size equal to 1. In this paper we adopt the standard unit cost RAM model [13].

* Supported by the Danish Natural Science Research Council (Grant No. 9400044). This research was done while visiting the Max-Planck Institut für Informatik, Saarbrücken, Germany. Email: `gerth@daimi.aau.dk`.

** On leave from Institute of Informatics, Warsaw University, ul. Banacha 2, 02–097, Warszawa, Poland. WWW: `http://zaa.mimuw.edu.pl/~lechu/lechu.html`, Email: `leszek@mpi-sb.mpg.de`.

*** Basic Research in Computer Science, a Centre of the Danish National Research Foundation

For the general case where $d > 1$, d–queries can be answered in optimal space $O(nm)$ doing $\sum_{i=0}^{d}\binom{m}{i}$ exact queries each requiring time $O(m)$ by using the data structure of Fredman, Komlos and Szemeredi [7]. On the other hand d–queries can be answered in time $O(m)$ when the size of the data structure can be $O(n\sum_{i=0}^{d}\binom{m}{i})$. We present the corresponding data structure of size $O(nm)$ for the 1–query case.

We present a simple data structure based on tries [1, 6] which has optimal size $O(nm)$ and supports 1–queries in time $O(m)$. Unfortunately, we do not know how to construct the data structure in time $O(nm)$ and we leave this as an open problem. However we give a more involved data structure of size $O(nm)$, based on two tries, supporting 1–queries in time $O(m)$ and which can be constructed in time $O(nm)$. Both data structures support the reporting of all strings with Hamming distance at most one of the query string α in time $O(m)$. For general d both data structures support d–queries in time $O(m\sum_{i=0}^{d-1}\binom{m}{i})$. The second data structure can be made semi-dynamic in terms of allowing insertions in amortized time $O(m)$, when starting with an initially empty dictionary. Both data structures work as well for larger alphabets $|\Sigma| > 2$, when the query time is slowed down by a $\log|\Sigma|$ factor.

The paper is organized as follows. In Sect. 2 we give a simple $O(nm)$ size data structure supporting 1–queries in time $O(m)$. In Sect. 3 we present an $O(nm)$ size data structure constructible in time $O(nm)$ which also supports 1–queries in time $O(m)$. In Sect. 4 we present a semi-dynamic version of the second data structure allowing insertions. Finally in Sect. 5 we give concluding remarks and mention open problems.

2 A trie based data structure

We assume that all strings considered are over a binary alphabet $\Sigma = \{0, 1\}$. We let $|w|$ denote the length of w, $w[i]$ denote the i-th symbol of w and w^R denote w reversed. The strings in the dictionary W are called dictionary strings. We let $dist_H(u, v)$ denote the Hamming distance between the two strings u and v.

The basic component of our data structure is a trie [6]. A trie, also called a digital search tree, is a tree representation of a set of strings. In a trie all edges are labeled by symbols such that every string corresponds to a path in the trie. A trie is a prefix tree, i.e. two strings have a common path from the root as long as they have the same prefix. Since we consider strings over a binary alphabet the maximum degree of a trie is at most two.

Assume that all strings $w_i \in W$ are stored in a 2-dimensional array $\mathcal{A}_W$ of size $n \times m$, i.e. of n rows and m columns, such that the i-th string is stored in the i-th row of the array $\mathcal{A}_W$. Notice that $\mathcal{A}_W[i, j]$ is the j-th symbol w_i. For every string $w_i \in W$ we define a set of *associated* strings $A_i = \{v \in \{0, 1\}^m | dist_H(v, w_i) = 1\}$, where $|A_i| = m$, for $i = 1, \ldots, n$. The main data structure is a trie T containing all strings $w_i \in W$ and all strings from A_i, for all $i = 1, \ldots, n$, i.e. every path from the root to a leaf in the trie represents one of the strings. The leaves of T

are labeled by indices of dictionary strings such that a leaf representing a string s and labeled by index i satisfies that $s = w_i$ or $s \in A_i$.

Given a query string α an 1–query can be answered as follows. The 1–query is answered positively if there is an exact match, i.e. $\alpha = w_i \in W$, or $\alpha \in A_j$, for some $1 \leq j \leq n$. Thus the 1–query is answered positively if and only if there is a leaf in the trie T representing the query string α. This can be checked in time $O(m)$ by a top-down traverse in T. If the leaf exists then the index stored at the leaf is an index of a matched dictionary string.

Notice that T has at most $O(nm)$ leaves because it contains at most $O(nm)$ different strings. Thus T has at most $O(nm)$ internal vertices with degree greater than one. If we compress all chains in T into single edges we get a compressed trie T' of size $O(nm)$. Edges which correspond to compressed chains are labeled by proper intervals of rows in the array $\mathcal{A}_W$. If a compressed chain is a substring of a string in the a A_j then the information about the corresponding substring of w_j is extended by the position of the changed bit. Since every entry in $\mathcal{A}_W$ can be accessed in constant time every 1–query can still be answered in time $O(m)$.

A slight modification of the trie T' allows all dictionary strings which match the query string α to be reported. At every leaf s representing a string u in T' instead of one index we store all indices i of dictionary strings satisfying $s = w_i$ or $s \in A_i$. Notice that the total size of the trie is still $O(nm)$ since every index i, for $i = 1, \ldots, n$, is stored at exactly $m + 1$ leaves. The reporting algorithm first finds the leaf representing the query string α and then reports all indices stored at that leaf. There are at most $m + 1$ reported string thus the reporting algorithm works in time $O(m)$. Thus the following theorem holds.

Theorem 1. *There exists a data structure of size $O(nm)$ which supports the reporting of all matched dictionary strings to an 1–query in time $O(m)$.*

The data structure above is quite simple, occupies optimally space $O(nm)$ and allows 1–queries to be answered optimally in time $O(m)$. But we do not know how to construct it in time $O(nm)$. The straight forward approach gives a construction time of $O(nm^2)$ (this is the total size of the strings in W and the associated strings from all A_i sets).

In the next section we give another data structure of size $O(nm)$, supporting 1–queries in time $O(m)$ and constructible in optimal time $O(nm)$.

3 A double-trie data structure

In the following we assume that all strings in W are enumerated according to their lexicographical order. We can satisfy this assumption by sorting the strings in W, for example, by radix sort in time $O(nm)$. Let $I = \{1, \ldots, n\}$ denote the set of the indices of the enumerated strings from W. We denote a set of consecutive indices (consecutive integers) an interval.

The new data structure is composed of two tries. The trie T_W contains the set of stings W whereas the trie $T_{\overline{W}}$ contains all strings from the set $\overline{W}$, where $\overline{W} = \{w_i^R | w_i \in W\}$.

Since T_W is a prefix trie every path from the root to a vertex u represents a prefix p_u of a string $w_i \in W$. Denote by W_u the set $\{w_i \in W | w_i \text{ has prefix } p_u\}$. Since strings in W are enumerated according to their lexicographical order those indices form an interval I_u, i.e. $w_i \in W_u$ if and only if $i \in I_u$. Notice that an interval of a vertex in the trie T_W is the concatenation of the intervals of its children. For each vertex u in T_W we compute the corresponding interval I_u, storing at u the first and last index of I_u.

Similarly every path from the root to a vertex v in $T_{\overline{W}}$ represents a reversed suffix s_v^R of a string $w_j \in W$. Denote by W^v the set $\{w_i \in W | w_i \text{ has suffix } s_v\}$ and by $S_v \subseteq I$ the set of indices of strings in W^v. We organize the indices of every set S_v in sorted lists L_v (in increasing order). At the root r of the trie $T_{\overline{W}}$ the list L_r is supported by a search tree maintaining the indices of all the dictionary strings. For an index in a list L_v the neighbor with the smaller value is called left neighbor and the one with greater value is called right neighbor. If a vertex x is the only child of vertex $v \in T_{\overline{W}}$ then S_x and S_v are identical. If vertex $v \in T_{\overline{W}}$ has two children x and y (there are at most two children since $T_{\overline{W}}$ is a binary trie) the sets S_x and S_y form a partition of the set S_v. Since indices in the set S_v are not consecutive (S_v is usually not an interval) we use additional links to keep fast connection between the set S_v and its partition into S_x and S_y. Each element e in the list L_v has one additional link to the closest element in the list L_x, i.e. to the smallest element e_r in the list L_x such that $e \leq e_r$ or the greatest element e_l in the list L_x such that $e \geq e_l$. Moreover in case vertex v has two children, element e has also one additional link to the analogously defined element $e_l \in L_y$ or $e_r \in L_y$.

Lemma 2. *The tries T_W and $T_{\overline{W}}$ can be stored in $O(nm)$ space and they can be constructed in time $O(nm)$.*

Proof. The trie T_W has at most $O(nm)$ edges and vertices, i.e. the number of symbols in all strings in W. Every vertex $u \in T_W$ keeps only information about the two ends of its interval $I_u = [l..r]$. For all $u \in T_W$ both indices l and r can be easily computed by a postorder traversal of T_W in time $O(nm)$.

The number of vertices in $T_{\overline{W}}$ is similarly bounded by $O(nm)$. Moreover, for any level $i = 1, \ldots, m$ in $T_{\overline{W}}$, the sum $\sum |S_v|$ over all vertices v at this level is exactly n since the sets of indices stored at the children forms a partition of the set kept by their parent. Since $T_{\overline{W}}$ has exactly m levels and every index in an L_v list has at most two additional links the size of $T_{\overline{W}}$ does not exceed $O(nm)$ too. The L_v lists are constructed by a postorder traversal of $T_{\overline{W}}$. A leaf representing the string w_i^R has $L_v = (i)$ and the L_v list of an internal vertex of $T_{\overline{W}}$ can be constructed by merging the corresponding disjoint lists at its children. The additional links are created along with the merging. Thus the trie $T_{\overline{W}}$ can be constructed in time $O(nm)$. □

Answering queries In this section we show how to answer 1–queries in time $O(m)$ assuming that both tries T_W and $T_{\overline{W}}$ are already constructed. We present a sequence of three 1–query algorithms all based on the double-trie structure. The first algorithm Query1 outlines how to use the presented data structure to answer 1–queries. The second algorithm Query2 reports the index of a matched dictionary string. The third algorithm Query3 reports all matched dictionary strings.

Let $pref_\alpha$ be the longest prefix of the string α that is also a prefix of a string in W. The prefix $pref_\alpha$ is represented by a path from the root to a vertex $\overline{u}$ in the trie T_W, i.e. $p_\alpha = p_{\overline{u}}$ but for the only child x of vertex $\overline{u}$ the string p_x is not a prefix of α. We call the vertex $\overline{u}$ the *kernel vertex* for the string α and the path from the root of T_W to the kernel vertex $\overline{u}$ the *leading path* in T_W. The interval $I_\alpha = I_{\overline{u}}$ associated with the kernel vertex $\overline{u}$ is called the *kernel interval* for the string α and the smallest element $\mu_\alpha \in I_\alpha$ is called the *key* for the query string α. Notice that the key $\mu_\alpha \in I_w$, for every vertex w on the leading path in T_W.

Similarly in the trie $T_{\overline{W}}$ we define the *kernel set* $S_{\hat{v}}$ which is associated with the vertex $\hat{v}$, where $\hat{v}$ corresponds to the longest prefix of the string α^R in $T_{\overline{W}}$. The vertex $\hat{v}$ is called a kernel vertex for the string α^R, and the path from the root of $T_{\overline{W}}$ to $\hat{v}$ is called the leading path in $T_{\overline{W}}$.

The general idea of the algorithm is as follows. If the query string α has an exact match in the set W, then there is a leaf in T_W which represents the query string α. The proper leaf can be found in time $O(m)$ by a top-down traverse of T_W, starting from its root.

If the query string α has no exact match in W but it has a match within distance one, we know that there is a string $w_i \in W$ which has a factorization $\pi_\alpha b \tau_\alpha$, satisfying:

(1) π_α is a prefix of α of length l_α,
(2) τ_α is a suffix of α of length r_α,
(3) $b \neq \alpha[l_\alpha + 1]$ and
(4) $l_\alpha + r_\alpha + 1 = m$.

Notice that prefix π_α must be represented by a vertex u in the leading path in T_W and suffix τ_α must be represented by a vertex v in the leading path of $T_{\overline{W}}$. We call such a pair (u, v) a *feasible* pair. To find the string w_i within distance 1 of the query string α we have to search all feasible pairs (u, v). Every feasible pair (u, v) for which $I_u \cap S_v \neq \emptyset$, represents *at least one* string within distance 1 of the query string α. The algorithm Query1 generates consecutive feasible pairs (u, v) starting with $u = \overline{u}$, the kernel vertex in T_W. The algorithm Query1 stops with a positive answer just after the first pair (u, v) with $I_u \cap S_v \neq \emptyset$ is found. It stops with a negative answer if all feasible pairs (u, v) have $I_u \cap S_v = \emptyset$.

Notice that the steps before the while loop in the algorithm Query1 can be performed in time $O(m)$. The algorithm looks for the kernel vertex in T_W going from the root along the leading path (representing the prefix $pref_\alpha$) as long as possible. The last reached vertex u is the kernel vertex $\overline{u}$. Then the corresponding

```
ALGORITHM Query1
begin
u := ū — the kernel vertex in T_W.
Find on the leading path in T_W̄ vertex v such that (u, v) is a feasible pair.
while vertex v exists do
        if I_u ∩ S_v ≠ ∅ then return "There is a match"
        u :=Parent(u)
        v :=Child-on-Leading-Path(v)
od
return "No match"
end.
```

vertex v on the leading path in $T_{\overline{W}}$ is found, if such a vertex exists. Recall that a pair (u, v) must be a feasible pair. At this point the following problem arises. How to perform the test $I_u \cap S_v \neq \emptyset$ efficiently?

Recall that the smallest index μ_α in the kernel interval I_α is called the key for the query string α and recall also that the key $\mu_\alpha \in I_w$, for every vertex w in the leading path in the trie T_W. During the first test $I_u \cap S_v \neq \emptyset$ the position of the key μ_α in S_v is found in time $\log |S_v| \leq \log n \leq m$ (since W only contains binary strings we have $\log n \leq m$). Let $I_u = [l..r]$, a be the left ($a \leq \mu_\alpha$) and b the right ($b > \mu_\alpha$) neighbors of μ_α in the set S_v. Now the test $I_u \cap S_v \neq \emptyset$ can be stated as:

$$I_u \cap S_v \neq \emptyset \equiv l \leq a \ \vee \ b \leq r.$$

If the above test is positive the algorithm Query2 reports the proper index among a and b and stops. Otherwise, in the next round of the while loop the new neighbors a and b of the key μ_α in the new list L_v are computed in constant time by using the additional links between the elements of the old and new list L_v.

Theorem 3. *1–queries to a dictionary W of n strings of length m can be answered in time $O(m)$ and space $O(nm)$.*

Proof. The initial steps of the algorithm (preceding the while loop) are performed in time $O(m + \log n) = O(m)$. The feasible pair (u, v) (if such exists) is simply found in time $O(m)$. Then the algorithm finds in time $O(\log n)$ the neighbors of μ_α in the list L_r which is held at the root of $T_{\overline{W}}$. This is possible since the list L_r is supported by a search tree. Now the algorithm traverses the leading path in $T_{\overline{W}}$ recovering at each level neighbors of μ_α in constant time using the additional links. There are at most m iterations of the while loop since there is exactly m levels in both tries T_W and $T_{\overline{W}}$. Every iteration of the while loop is done in constant time since both neighbors a and b of the key μ_α in the new more sparse set S_v are found in constant time. Thus the total running time of the algorithm is $O(m)$. □

```
ALGORITHM Query2
begin
u := ū — the kernel vertex in T_W.
Find on the leading path in T_{W̄} vertex v such that (u, v) is a feasible pair.
Find the neighbors a and b of the key μ_α in S_v.
while vertex v exists do
        if l ≤ a then return "String a is matched"
        if b ≤ r then return "String b is matched"
        u :=Parent(u); Set l and r according to the new interval I_u.
        v :=Child-on-Leading-Path(v)
        Find new neighbors of μ_α, a and b, in the new list L_v.
od
return "No match"
end.
```

We explain now how to modify the algorithm Query2 to an algorithm reporting all matches to a query string. The main idea of the new algorithm is as follows. At any iteration of the while loop instead of looking only for the left and the right neighbor of the key index μ_α the algorithm Query3 searches one by one all indices to the left and right of μ_α which belong to the list L_v and to the interval I_u. To avoid multiple reporting of the same index the algorithm searches only that part of the new interval I_u which is an extension of the previous one. The variables a and b store the leftmost and the rightmost searched indices in the list L_v.

Theorem 4. *There exists a data structure of size $O(nm)$ and constructible in time $O(nm)$ which supports the reporting of all matched dictionary strings to a 1–query in time $O(m)$.*

Proof. The algorithm Query3 works in time $O(m + \#matched)$, where $\#matched$ is the number of all reported strings. Since there is at most $m+1$ reported strings (one exact matching and at most m matches with one error) the total time of the reporting algorithm is $O(m)$. □

4 A semi-dynamic data structure

In this section we describe how the data structure presented in Sect. 3 can be made semi-dynamic such that new binary strings can be inserted into W in amortized time $O(m)$. In the following w' denotes a string to be inserted into W.

The data structure described in Sect. 3 requires that the strings w_i are lexicographically sorted and that each string has assigned its rank with respect to the lexicographical ordering of the strings. If we want to add w' to W we can use T_W to locate the position of w' in the sorted list of w_is in time $O(m)$. If we continue to maintain the ranks explicitly assigned to the strings we have to

```
ALGORITHM Query3
begin
u := ū — the kernel vertex in T_W.
Find on the leading path in T_{\overline{W}} vertex v such that (u, v) is a feasible pair.
Find the neighbors a and b of the key μ_α in S_v.
while vertex v exists do
        while l ≤ a do
                report "String a is matched"
                a := left neighbor of a in L_v.
        od
        while b ≤ r do
                report "String b is matched"
                b := right neighbor of b in L_v.
        od
        u :=Parent(u); Set l and r according to new I_u.
        v :=Child-on-Leading-Path(v)
        Find a or the left neighbor of a in the new list L_v.
        Find b or the right neighbor of b in the new list L_v.
od
end.
```

reassign new ranks to all strings larger than w'. This would require time $\Omega(n)$. To avoid this problem, observe that the indices are used to store the endpoints of the intervals I_u and to store the sets S_v, and that the only operation performed on the indices is the comparison of two indices to decide if one string is lexicographically less than another string in constant time.

Essentially what we need to know is if given the *handles* of two strings from W, which one of the two strings is the lexicographically smallest. A solution to this problem was given by Dietz and Sleator [3]. They presented a data structure that allows a new element to be inserted into a linked list in constant time if the new element's position is known, and that can answer order queries in constant time.

By applying the data structure of Dietz and Sleator to maintain the ordering between the strings, an insertion can now be implemented as follows. First insert w' into T_W. This requires time $O(m)$. The position of w' in T_W also determines its location in the lexicographically order implying that the data structure of Dietz and Sleator can be updated too. By traversing the path from the new leaf representing w' in T_W to the root of T_W, the endpoints of the intervals I_u can be updated in time $O(m)$.

The insertion of w'^R into $T_{\overline{W}}$ without updating the associated fields can be done in time $O(m)$. Analogously to the query algorithm in Sect. 3, the positions in the S_v sets along the insertion path of w' in $T_{\overline{W}}$ where to insert the handle of w' can be found in time $O(m)$.

The problem remaining is to update the additional links between the elements in the L_v lists. For this purpose we change our representation to the following. Let v be a node with sons x and y. In the following we only consider how to handle the links between L_v and L_x. The links between L_v and L_y are handled analogously. For each element $e \in L_v \cap L_x$ we maintain a pointer from the position of e in L_v to the position of e in L_x. For each element $e \in L_v \setminus L_x$ the pointer is null. Let $e \in L_v$. We can now find the closest element to e in L_x by finding the closest element in L_v that has a non null pointer. We denote such an element to be marked. For this purpose we use the find-split-add data structure of Imai and Asano [10], an extension of a data structure by Gabow and Tarjan [8]. The data structure supports the following operations: Given a pointer to an element in a list, to find the closest marked element (find); to mark an unmarked list element (split); and to insert a new unmarked element into the list adjacent to an element in the list (add). The operations split and add can be performed in amortized constant time and find in worst case constant time on a RAM. Going from e in L_v to e's closest neighbor in L_x can still be performed in worst case constant time, because this only requires one find operation to be performed. When a new element e is added to L_v we just perform add once, and in case e is added to L_x too we also perform split on e. This requires amortized constant time. Totally we can therefore update all the links between the L_v lists in amortized time $O(m)$ when inserting a new string into the dictionary.

Theorem 5. *There exists a data structure which supports the reporting of all matched dictionary strings to a 1–query in worst case time $O(m)$ and that allows new dictionary strings to be inserted in amortized time $O(m)$.*

5 Conclusion

We have presented a data structure for the approximate dictionary query problem that can be constructed in time $O(nm)$, stored in $O(nm)$ space and that can answer 1–queries in time $O(m)$. We have also shown that the data structure can be made semi-dynamic by allowing insertions in amortized time $O(m)$, when we start with an initially empty dictionary. For the general d case the presented data structure allows d–queries to be answered in time $O(m \sum_{i=0}^{d-1} \binom{m}{i})$ by asking 1–queries for all strings within Hamming distance $d-1$ of the query string α. This improves the query time of a naïve algorithm by a factor of m. We leave as an open problem if the above query time for the general d case can be improved when the size of the data structure is $O(nm)$. For example, is there any $o(m^2)$ 2–query algorithm?

Another interesting problem which is related to the approximate query problem and the approximate string matching problem can be stated as follows. Given a binary string t of length n, is it possible to create a data structure for t of size $O(n)$ which allows 1–queries, i.e. queries for occurrences of a query string with at most one mismatch, in time $O(m)$, where m is the size of the query string? By creating a compressed suffix tree of size $O(n)$ for the string, 1–queries can be answered in time $O(m^2)$ by an exhaustive search.

Acknowledgment

The authors thank Dany Breslauer for pointing out the relation to the find-split-add problem.

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
2. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.
3. Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 365–372, 1987.
4. Danny Dolev, Yuval Harari, Nathan Linial, Noam Nisan, and Michael Parnas. Neighborhood preserving hashing and approximate queries. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 251–259, 1994.
5. Danny Dolev, Yuval Harari, and Michael Parnas. Finding the neighborhood of a query in a dictionary. In *Proc. 2nd Israel Symposium on Theory of Computing and Systems*, pages 33–42, 1993.
6. E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1962.
7. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.
8. Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.
9. Dan Greene, Michal Parnas, and Frances Yao. Multi-index hashing for information retrieval. In *Proc. 35th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 722–731, 1994.
10. Hiroshi Imai and Taka Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8:1–18, 1987.
11. Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50:191–197, 1994.
12. M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, Mass., 1969.
13. P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. MIT Press/Elsevier, 1990.
14. Andrew C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.
15. Andrew C. Yao and Frances F. Yao. Dictionary look-up with small errors. In *Proc. 6th Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 388–394. Springer Verlag, Berlin, 1995.