Should tables be sorted?

Lecture #1: September 26, 2005
Scribe: Rajeev Motwani

In the first few lectures we will be talking mainly about hashing and briefly touch upon the area of implicit data structures. Some of the initial discussion may give the impression of being a theoretical game with little relevance to practice, but, as we will show later, this introduces us to some of the most basic issues and ideas in data structuring. Moreover, we will be developing a model (the cell probe model) which is quite standard for proving lower bounds on data structure problems, and be exposed to some ideas in hashing which have had a profound influence in such unrelated fields as complexity theory. Our story begins with a paper written by Andy Yao in 1981 ["Should tables be sorted?", *JACM* (1981)]. Please refer to the handout from the first lecture for an annotated bibliography of relevant articles.

1 The Cell Probe Model

We start by recalling the most basic problem in data structures.

Definition 1 **[The Static Dictionary Problem]**
Let m and n be natural numbers such that $n \leq m$, and define the *key space* as $M = \{1, 2, \ldots, m\}$. Given a data set $S \subseteq M$ such that $|S| = n$, we are required to organize S into a data structure which permits efficient processing of membership[1] queries of the type "Is key q in S?"

A typical solution to this problem involves a preprocessing scheme which organizes the set S into a table T, and a searching scheme which accesses this table to answer queries. You might think that this is a well-solved problem since we all know that S can be sorted in $\mathrm{O}(n \log n)$ time, after which a binary search can asnwer queries in $\lceil \log(n+1) \rceil$ time which is known to be optimal. But the optimality of binary search (following sorting) is based on the assumption of a comparison-based model of computation, and this need not be the correct view of what can be efficiently realized in practice.

Suppose we had m bits of memory available, then we could set the i'th bit to indicate the presence or absence of the key i in S, and using random access a query could then be processed in unit time. The catch with this approach is that the preprocessing and space grow with m, and in typical applications we have that m is many orders of magnitude larger than n. A more practical approach is the use of hash tables which guarantee search time of $\mathrm{O}(1)$ on the average (under some natural probabilistic assumptions about S), although they do have $\mathrm{O}(n)$ worst-case search times. Observe that these schemes do not make any significant use of comparisons and serve to illustrate the need for closely examining our choice of the underlying model of computation.

[1] In most applications, the keys are a part of a large record and so it is important to access the entire record associated with the key. This necessitates the determination of the exact location of the query key in the data structure, rather than just deciding the existence of that key in S. Some of the schemes described below suffer from the drawback that they cannot efficiently locate the query key.

1.1 Implicit Dictionaries

The question we address is the following: using a reasonable amount of space and preprocessing, can we improve upon the performance guaranteed by binary search on sorted tables. In fact, Yao's paper derives its title from this very question. We start by considering a very special type of data structure called an *implicit data structure*. This is a scheme which uses exactly the amount of space required to store S, and has no auxiliary space for storing ordering information which may help the process of answering queries. This would imply a table which could store only the n keys in S, although the exact order in which these keys appear is not pre-determined and this order implicitly can encode information for speeding-up the search.

The following model, called Model A, is a specialization of the Cell Probe Model to implicit dictionaries. The set S has to be stored in a table T consisting of exactly n cells and allowing random access. Each cell can store an arbitrary key from M, but implicitness requires that each cell store a distinct key from S. A *table structure* τ specifies the order in which the keys of S are stored in T. A *search strategy* $\mathcal{S}$ takes as input a query key $q \in M$, and specifies a sequence of probes into the table T which determine the membership of q in S. A probe is the examination of the contents of a particular cell in T, and it should be noted that the probe sequence is completely adaptive in that the choice of the next cell to probe can depend upon the outcome of the earlier probes. All computation is free, and the cost of a search is measured only in terms of the number of probes required by $\mathcal{S}$. (Notice that this is a lower bound on the cost of the search in any reasonable model of computation such as the RAM model.)

We define the following complexity measures:

$$\begin{aligned} c(\tau, \mathcal{S}) &= \max_{S,q}\{\text{number of probes for } q\} \\ f(n, m) &= \min_{\tau,\mathcal{S}} c(\tau, \mathcal{S}) \end{aligned}$$

The function $f(n,m)$ is the inherent (search) complexity of the static dictionary problem in Model A. Choosing τ to be a sorted table and $\mathcal{S}$ to be binary search, we obtain that $f(n,m) \leq \lceil \log(n+1) \rceil$. Our basic goal is to determine whether this upper bound is tight.

We start by considering the value of $f(2,3)$. Let us first assume that τ is a sorted table, and determine whether binary search is the optimal search strategy for this case. (Notice that binary search requires 2 probes in the worst case.) Consider any arbitrary search strategy $\mathcal{S}$ for sorted tables trying to answer the query $q = 2$. The very first probe made by $\mathcal{S}$ can be either to $T(1)$ or $T(2)$. In the first case the result of the probe cannot distinguish between the $S = \{1,2\}$ and $S = \{1,3\}$, while in the second case it cannot distinguish $S = \{1,3\}$ and $S = \{2,3\}$. In either case, a second probe is required to correctly determine the membership of $q = 2$ in S. Thus, binary search is optimal for sorted tables.

However, it is not clear that a sorted table is the best possible choice in the first place. In fact, we claim that $f(2,3) = 1$! To see this consider the τ which stores S in a *cyclic order*. In a cyclic ordering, the very first key in T is the key that this is the immediate successor of the key *not* present in S. Thus, the only difference with respect to the sorted table is that the ordering for $S = \{1,3\}$ is reversed. It is pretty easy to see that this requires only 1 probe.

Exercise 2 Generalize the idea of cyclic orders so as to obtain that, for all n, $f(n, n+1) = 1$.

Having seen that we can beat binary search of sorted tables when $m = n+1$, the next step is to see if the same can be done for larger m. An interesting special case is the value of $f(2,m)$ for $m \geq 4$. Unfortunately, as shown by the following theorem, binary search of sorted tables is indeed optimal for these cases.

Theorem 3 For $m \geq 4$, we have that $f(2, m) = \lceil \log(2+1) \rceil = 2$.

Proof: Basically we need to show only the lower bound and that too only for $m = 4$, and the result follows for all larger m. (Do you see why?)

The idea is to first represent each possible table structure τ by means of a directed graph G. The vertices of G correspond to the four elements of $M = \{1, 2, 3, 4\}$. As for the edges, we start with the complete *undirected* graph (the graph K_4) and orient each edge according to the behavior of τ. In particular, if for $S = \{i, j\}$ the table structure τ places i before j in T then we orient the corresponding edge in G from i to j, otherwise we orient the edge in the reverse direction.

We claim that any orientation of the edges of K_4 has three vertices whose induced subgraph is *not* an oriented cycle (its undirected version is always a 3-cycle). To see this, fix any vertex i and consider its three incident edges – there must be two edges which are either both incoming or both outgoing. The three vertices which form the end-points of these two edges have the desired property.

Now, assume without loss of generality that these three vertices are the ones corresponding to the keys 1, 2 and 3. Clearly, there exists a total ordering on these three keys such that τ stores them in T in that order. But this means that, upto the renaming of the keys, the restriction of τ to these three keys in M is essentially the same as the sorted table structure for the case of $n = 2$ and $m = 3$. But by our preceding analysis, the optimal search strategy for sorted tables in that case is exactly the binary search strategy. This implies that the optimal search strategy for τ restricted to $\{1, 2, 3\}$ is also binary search, and therefore $f(2, 4) = \lceil \log(2+1) \rceil = 2$. ▮

We are now ready to tackle the case of general n. Our approach will be to generalize the ideas used in analyzing the case of $n = 2$, and we will obtain for sufficiently large m we may as well use sorted tables. (The only catch is that we can prove this only for tremendously large values of m, as will become clear shortly.) The proof idea is the following: we show that any τ must behave like a sorted table on the data sets contained in a subspace of M of size $2n - 1$, provided m is sufficiently large; further, for sorted tables in the case of $m = 2n - 1$, binary search is optimal. (Note that for $n = 2$, we have $2n - 1 = 3$.)

Lemma 4 If $m \geq 2n - 1$, and τ stores all data sets in the order specified by a fixed permutation π of M, then $\mathcal{S}$ must use $\lceil \log(n+1) \rceil$ probes in the worst case.

Proof: First, we observe that by renaming the keys it suffices to prove this result for the identity permutation. We now proceed by induction on n. The basis of our induction is the fact that for $n = 2$ and $m = 3$, the query key $q = 2$ requires 2 probes when the table is sorted. Our induction hypothesis is slightly stronger than the statement of the lemma:

> For any $n' < n$ and $m' \geq 2n' - 1$, the query $q = n'$ requires $\lceil \log(n'+1) \rceil$ probes if τ stores the data sets according to the identity permutation.

To establish the induction hypothesis, we consider an arbitrary $\mathcal{S}$ and assume that for the query $q = n$ the first probe is made to the location $T(p)$. By symmetry, we can assume that $p \leq \lceil n/2 \rceil$.

Now we postulate an adversary who adaptively chooses the keys in S depending upon the strategy $\mathcal{S}$. The adversary chooses the keys 1, 2, ..., $\lceil n/2 \rceil$ to be all contained in S, thereby ensuring that for $1 \leq i \leq \lceil n/2 \rceil$, $T(i) = i$ (since τ is using the identity permutation, or sorting.) The adversary also commits to choosing the remaining elements of S from the reduced key space $M' = \{\lceil n/2 \rceil + 1, \ldots, m - \lceil n/2 \rceil\}$.

It is easy to see that the first probe made by $\mathcal{S}$ does not provide any useful information about the contents of the second half of T. In fact, after that probe the search strategy has to solve a reduced search problem involving $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ table entries drawn from a key space of size $m' = m - 2\lceil n/2 \rceil \geq 2n' - 1$. Moreover, the relative rank of the query key $q = n$ in the reduced key space M' is exactly $n - \lceil n/2 \rceil = n'$ and the induction hypothesis is now applicable. Since $n' < n$, we obtain by the induction hypothesis that an additional $\lceil \log(n'+1) \rceil$ probes are needed. Since the total number of probes is $1 + \lceil \log(n'+1) \rceil \geq \lceil \log(n+1) \rceil$, the induction is complete. ∎

To prove the desired result, all we need to show is that for sufficiently large m and any τ there exists a subset of M of size $2n-1$ such that all data sets contained in this subspace are ordered by a fixed permutation. For this purpose we will need a Ramsey theorem.

Fix any table structure τ. Let σ be any permutation of the set $\{1, 2, \ldots, n\}$. Define the set G_σ as the collection of all $S \subseteq M$ of size n such that τ stores S in T according to the permutation σ, i.e. according to τ the cell $T(\sigma(i))$ contains the i'th smallest key in S. For σ ranging over the space of all permutations, the sets G_σ form a partition of all the subsets of M os size n. The following theorem is well-known in Ramsey Theory.

Theorem 5 **[Ramsey Theorem]** Consider any coloring of all the n-subsets of M by c colors. For $k \geq n$ and $m \geq R(n, c, k)$, there exists a set $M' \subseteq M$ of size k all of whose n-subsets are assigned the same color under this coloring.

The function R is a tremendously fast-growing function and we do not give its explicit form. We are now ready to prove the main result.

Theorem 6 For all n and $m \geq R(n, n!, 2n-1)$, $f(n, m) = \lceil \log(n+1) \rceil$.

Proof: Fix any table structure τ. This assigns a permutation σ to each n-subset of M, i.e. the order in which τ stores that subset in T. Viewing the permutations as colors and using $c = n!$, we obtain that there is a subspace $M' \subseteq M$ of size $2n-1$ such that every n-subset of M' is assigned the same permutation. Under reindexing of the entries in the table T, this is equivalent to saying that all n-subsets of M' are stored in a sorted order. The result now follows from Lemma 4. ∎

We have shown that sorting and binary search is optimal for sufficiently large m. Unfortunately, the function R is such that this requires m to be larger than a number which is a tower of successively exponentiated 2's with height of the tower being superlinear in n. In real life, we do not have to deal with such values of m. It remains an outstanding open problem to answer the question raised in the title of Yao's paper for reasonably bounded values of m, e.g. $m \leq 2^{2^n}$. In the sequel we will shed some light on the behavior of $f(n, m)$ for intermediate values of m, but the situation is still quite murky and open problems abound!

1.2 Generalized Cell Probe Models

Another outstanding open problem is to study the behavior of $f(n, m)$ in the non-implicit version of Model A. More precisely, we can allow T to contain an arbitrary subset of the keys in M, allowing keys to repeat as well as the use of keys which are not present in S to encode information which would speed-up the search process. We present some results in a weaker form of this generic model.

The new model, called Model B, assumes the same set-up as in Model A. The crucial new ingredient is a set $P = \{x_1, \ldots, x_p\}$ of special symbols called *pointers*. These are symbols which are not contained in M and can be used freely to encode information. (The choice of the name for this set is intended to highlight the possibility that these symbols could indeed be pointers to other cells.) Further, we relax the implicitness constraint on T by allowing it to contain $r \geq n$ symbols from $S \cup P$, i.e. $T \in (S \cup P)^r$. Note that T can contain only the keys in S, but these could repeat or not appear at all. It is not very hard to see that a wide variety of commonly used data structures (such as linked lists, hash tables, search trees, etc.) can be implemented in Model B. We denote the worst-case search cost in this model by $f(n, m, p, r)$ in the same way as $f(n, m)$.

The following theorem can be proved in much the same way as Theorem 6 and we provide only a sketch of the proof.

Theorem 7 For all n, p, $r \geq n$, and $m \geq R(n, (n+p)^r, 2n-1)$,

$$f(n, m, p, r) = \lceil \log(n+1) \rceil .$$

Proof: **[Sketch]** As before, fix a table structure τ. The behavior of τ on a given $S \subseteq M$ is encoded as an r-tuple $(z_1, \ldots, z_r)$ whose entries are determined as follows. Consider the contents of T when τ organizes S into it: if $T(j)$ contains the k'th smallest key in S, then set $z_j = k$; otherwise, $T(j)$ contains a pointer x_l and so set $z_j = n + l$. Note that in the case where there are no pointers and T is implicit, the r-tuple reduces to an encoding of the permutation used by τ for S.

By the Ramsey Theorem, there is a subspace $M' \subseteq M$ of size $2n-1$ such that all the n-subsets of M' are assigned the same r-tuple. In the tables for such $S \subseteq M'$, the same pointers appear at the same locations and are therefore irrelevant, and moreover the location of the keys in T depend only on their rank in S. It is now clear that these tables can be viewed as being sorted and so Lemma 4 applies, giving the desired bound. Note that the tables may actually have some keys of S completely omitted, but this can only hurt the search strategy. ∎

Once again, this theorem applies to tremendously large values of m and it remains an interesting open problem to determine the behavior of $f(n, m, p, r)$ in general.

1.3 Perfect Schemes

In this section we only consider the Model A. We briefly consider an alternate version of the question studied above. Define a *perfect* strategy as one which permits query processing in unit time, i.e. any query can be answered by making only one probe. In general, when do k probes suffice? It is clear from the preceding discussion that as m increases, the number of probes required also increases. Therefore, we define the function $b(n, k)$ as the largest m such that $f(n, m) = k$. In other words, we need more than k probes if and only if $m > b(n, k)$.

We have already seen that $f(2, 3) = 1$ and $f(2, 4) = 2$, implying that $b(2, 1) = 3$. We now characterize $b(n, 1)$ for $n > 2$, showing that it equals $2n - 2$. Note the curious coincidence of this being one less than the value $2n - 1$ which played an important role in sorted tables!

Theorem 8 For $n > 2$, $b(n, 1) = 2n - 2$.

Proof: We only prove a lower bound of $2n - 2$ on $b(n, 1)$ by presenting a data structure for the case $m = 2n-2$ which uses only one probe. The upper bound is omitted as it involves a tedious case analysis.

Our idea will be to view the table T as a hotel with n rooms where each key in M is a tenant assigned one of these rooms. Since there are $2n-2$ keys in all, some rooms are assigned two tenants. For $1 \leq j \leq n-2$, room j is assigned key $n+j$ as an upper tenant and key j as the lower tenant. The rooms $n-1$ and n are assigned only the lower tenants, viz. the keys $n-1$ and n, respectively.

Consider now the task facing τ when a set S of size n is specified. The problem is that some rooms may have both their tenants present in S, and the following rules are then used to reshuffle the tenancy assignments.

Rule I: In the rooms j, for $1 \leq j \leq n-2$, if only one of the two tenants shows up then place it in its correct room.

Rule II: If both tenants show up for a particular room, then assign the upper tenant to any room which has none of its tenants in S. (There always exist empty rooms for this purpose!)

Rule III: All unassigned tenants, including the lower tenants of Rule II and the keys $n-1$ and n, undergo a cycle shift in their room assignments. (Actually, it suffices to use any permutation which maps each of these tenants to a room other than its own room.)

Following this, it is easy to devise a search strategy which always uses only one probe. Given a query q, probe the room R_q assigned to q in the original tenancy assignments. If q is in R_q (when Rule I applied to it), then we are done. On the other hand, if R_q contains the lower tenant of another room we know that Rule III applied to q and it must be somewhere in the table. Finally, if R_1 has an upper tenant of some other room, it follows by Rule II that $q \notin S$. $\blacksquare$

Notice that the scheme falls apart if we have $2n-1$ keys in all. For example, if we choose to assign two tenants to room $n-1$ also then a bad case is when $S = \{1, 2, \ldots, n-1, n+1\}$. In this scenario, the application of Rule III is a problem since at that stage there is only one key which needs to undergo a cyclic shift.

This strategy has the drawback that it does not determine the precise location of a query key in T, it just decides the membership problem. It remains an interesting open problem to fix this shortcoming. Another problem is to provide some characterization of $b(n,k)$ for $k > 1$. In the next section, we show that two probes are always sufficient for large enough m provided we add one extra cell to T which can store any key from M as a guide for the search strategy. Notice that the addition of this extra cell circumvents the lower bounds of $\lceil \log(n+1) \rceil$ in both Models A and B.

1.4 Model C

The new model, called Model C, is basically an extension of Model A where we allow the table T to be larger than n in size, and we also permit τ to store any key from M in the table, even those keys which do not belong to S. We will show that using only one extra cell (beyond those needed to store S) is enough to permit query processing in 2 probes, provided $m > h(n)$. Unlike in the case of the Ramsey Theorem, the function h is of a very reasonable size. The original result by Yao was that $h(n) = 2^{16n^2}$ suffices, but Tarjan later improved this to $2^{\mathrm{O}(n \log n)}$. (A result by Graham shows that there exists such a τ for $h(n) = 2^{\mathrm{O}(n)}$ but this result is non-constructive.) Finally, Tarjan and Yao have shown that using $\mathrm{O}(n)$ extra cells, it is possible to process queries using $\mathrm{O}(1)$ probes provided m is no more than a polynomial in n. The last result is based on the use of tries, and we omit its discussion. An interesting open problem is to explore the power of this new model between these two extreme values of m.

The result that we prove here is a non-constructive one and achieves $h(n) = 2^{O(n \log n)}$. The basic idea in all these results is to use a combinatorial structure called a separating system.

Definition 9 An *ordered n-partition* P is an ordered sequence of sets $(A_1, \ldots, A_n)$ which form a partition of M.

Definition 10 Let $S \subseteq M$ be a data set of size n such that s_i is the i'th smallest key in S, for $1 \leq i \leq n$. Then an ordered n-partition $P = (A_1, \ldots, A_n)$ is a *separator* for S if for $1 \leq i \leq n$, $s_i \in A_i$.

Definition 11 An (n, m)-*separating system* is a collection of ordered n-partitions $\mathcal{P} = \{P^1, \ldots, P^r\}$ such that every data set $S \subseteq M$ of size n has a separator in $\mathcal{P}$.

We first show that given a separating system $\mathcal{P}$, we can devise a 2 probe strategy in Model C.

Theorem 12 Suppose there exists an (n, m)-separating system $\mathcal{P}$ of size $r = h(n)$. Them for $m \geq h(n)$, we devise a table structure τ for a table with one extra cell so as to be able to answer all possible queries in exactly two probes.

Proof: The table structure takes any given data set S and finds a separator P^D for it in the separating system $\mathcal{P}$. Suppose that the table entries are indexed by 0 through n. The index D (called the directory) is stored in $T(0)$, and the i'th smallest key in S is stored in $T(i)$. Notice that since $m \geq |\mathcal{P}|$, the keys in M can be used to encode indices for the ordered partitions.

The search strategy is to first examine the cell $T(0)$ to determine the separator $P^D = (A_1, \ldots, A_n)$ for S. Now, if the query key q lies in A_i, then either $s_i = q$ or q does not belong to S. (Do you see why this follows from the above definitions?) Thus, the second probe is to the location $T(i)$, and this suffices to answer the query. ∎

We now sketch the proof of the existence of small separating systems. The basic idea is to independently choose r equipartitions $\mathcal{P} = \{P^1, \ldots, P^r\}$ uniformly at random. This means that each partition $P = (A_1, \ldots, A_n)$ such that, for all i, $|A_i| = m/n$ is equally likely to be chosen in each of the r samples. We now show that the probability that $\mathcal{P}$ is not a separating system is strictly less than 1, thereby implying the existence of such a system. (This is an example of the use of the *probabilistic method.*)

Fix any set $S \subseteq M$ of size n, and observe that a random equipartition is a separator for S with probability equal to the ratio of the number of equipartitions separating S to the total number of equipartitions, call this probability p. We have that:

$$p = \frac{(m-n)!}{m!} \left(\frac{m}{n}\right)^n \approx 1/n^n.$$

Now the probability that none of the partitions in $\mathcal{P}$ is a separator for S is $(1-p)^r$. Finally, observe that the probability that $\mathcal{P}$ is not a separating system is the probability that there exists a set S for which no separator was chosen, and the latter is bounded by

$$\binom{m}{n} (1-p)^r.$$

It is now a routine calculation to verify that this probability is strictly less than 1, provided we choose r to be $2^{cn \log n}$ for a suitable constant c.

1.5 Some Remarks

Observe that the schemes which can be constructed in Models A and B can only use the "addressing" power of the RAM model, and this is not sufficient to improve upon sorting and binary search at least for large values of m. However, in Model C making only a very mild use of the "encoding" power of the RAM model we are able to achieve an astonishingly strong improvement over the sorting paradigm and that too for reasonable values of m. In fact, the only reason for requiring that m be larger than a single exponential in n is to permit the encoding of large sets into the keys of M! It remains an interesting open problem to explore the power of Model C for smaller value of m. In the next two topics, we present partial (positive) answers to this last question with some use of randomness.

2 Universal Hash Functions

In the static dictionary problem, we are provided with a data set S and are required to organize it into a data structure which supports efficient processing of membership queries. In the *dynamic dictionary* problem, the set S is not provided in advance, instead it is constructed via a series of insert and delete operations intermingled with the queries. Since the size of S is not fixed a priori, we will use the notation that $|S| = s$ and reserve the use of the symbol n for other purposes.

Standard solutions to this problem involve balanced search trees for the dynamic case. (Observe that these are a generalization of the binary search technique used for the static case.) These methods take $O(\log s)$ time to process any query or update operation, and are fairly general in that they only assume the existence of a total ordering on the key space M.

It would be desirable to obtain query and update time which is bounded independently of the size of S. Of course, given enough memory, say m bits, we can perform all operations in unit time by storing an m-bit vector specifying the elements of M present in S, using a random access mechanism to examine any bit. As before, this is not a good solution as in practice m is usually much larger than s. We shall restrict ourselves to space-efficient data structures which use only $O(s)$ memory. Now, we have seen that even under the very general RAM model of computation permitting the use of pointers, for sufficiently large m there is a lower bound of $\Omega(\log(s+1))$ on the time required to process a query in a static dictionary. This model is quite general in that it allows for all kinds of linked data structures including sorted tables and search trees. (A crucial assumption for this lower bound is that only the keys from S can be stored in the memory. We have already seen that permitting the storage of arbitrary keys from M allows us to circumvent this lower bound, and the next lecture will provide a more practical version of that scheme.)

A *hash table* is a data structure which allows us to obtain $O(1)$ query time on the *average*, using only a linear amount of storage. A table T consists of n cells indexed by $N = \{0, 1, \ldots, n-1\}$. A *hash function* h is a mapping from M into N. To insert an item into this data structure we compute the hash value $h(x)$ for each $x \in S$, and then store the key x in the cell $T(h(x))$. This would permit $O(1)$ time processing of query and update operations provided no two keys in S are hashed to the same location in the table.

A *collision* occurs between two keys x and y if $h(x) = h(y)$. A *perfect hash function* for a set S is a function $h : M \to N$ such that h does not cause any collisions among the keys of the set S. It is easy to see that there exists a perfect hash function for any set S of cardinality at most n. While perfect hash functions can be constructed for arbitrary sets, this does not solve the basic problem for two reasons. First, there is the overhead of computing and storing the description of the perfect hash function. Furthermore, this will only work for the static dictionary problem since a hash function cannot be perfect for more than a very small number of sets, and we don't know the set S in advance. Consequently, there has been a lot of interest in the design of hash functions which perform well on a randomly chosen set S. Under probabilistic assumptions about S, we can construct hash functions which have on the average only $O(1)$ keys colliding at any given table location. The colliding keys can then be organized into a secondary data structure accessible from that location, or they can be rehashed into a new location using a new hash function. Several such methods guarantee an expected time of $O(1)$ per operation, but suffer from the disadvantage that assumptions must be made about the distribution of the input sets.

We describe a randomized solution to this problem which achieves the desired time bound. The solution requires the construction of a class of hash functions which have found a surprisingly large number of applications in areas far removed from the original problem, e.g. routing in networks, or complexity theory. The basic idea is to choose a *family* of hash functions $H = \{h : M \rightarrow N\}$, where each $h \in H$ is easily represented and evaluated. While any one function $h \in H$ may not be perfect for very many choices of the set S, it can be ensured that for every set S of small cardinality, a large fraction of the hash functions in H are near-perfect for S. Thus, for any particular set S, a random choice of $h \in H$ will give the desired performance. Observe that a hash function is just another type of a fingerprint function for the keys in M. It is not very hard to see that the hash functions described here can also be used to solve some of the problems discussed in earlier sections.

Definition 13 Let $M = \{0, 1, \ldots, m-1\}$ and $N = \{0, 1, \ldots, n-1\}$, with $m \geq n$. A family H of functions from M into N is said to be 2-*universal* if for all x, $y \in M$, such that $x \neq y$, and h chosen uniformly at random from H,

$$\mathbf{P}[h(x) = h(y)] \leq \frac{1}{n}.$$

A totally random mapping from M to N has a collision probability of exactly $1/n$; thus, a random choice from a 2-universal family of hash functions gives a seemingly random function. Obviously, the collection of all possible functions from M to N is a 2-universal family. The family of all possible functions has several disadvantages. Firstly, picking a random function then requires $\Omega(m \log n)$ random bits. Moreover, this is also the number of bits of storage required to represent that function. Our goal, therefore, is to obtain much smaller 2-universal families of functions which require a small amount of space and are easy to evaluate. The reason we call this a 2-universal family is that a random $h \in H$ is required to behave like a random function only with respect to pairs of elements. It is not surprising, therefore, that we are able to construct 2-universal families containing only a small subset of all possible functions. There is a natural generalization of this definition to that of k-universal families. Observe the similarity to the notions of pairwise and k-wise independence.

Before we provide a construction for such families, let us see why this gives a good solution to the dynamic dictionary problem. From here on fix the sets M, N and H as in Definition 13. For any x, $y \in M$ and $h \in H$, define the following indicator function for a collision between the keys x and y under the hash function h.

$$\delta(x, y, h) = \begin{cases} 1 & \text{for } h(x) = h(y) \text{ and } x \neq y \\ 0 & \text{otherwise} \end{cases}$$

Let $S \subseteq M$, and define the following extensions of the indicator function δ.

$$\begin{aligned} \delta(x, y, H) &= \sum_{h \in H} \delta(x, y, h) \\ \delta(x, S, h) &= \sum_{y \in S} \delta(x, y, h) \end{aligned}$$

$$\delta(x,S,H) = \sum_{y\in S}\delta(x,y,H).$$

For a 2-universal family H and any $x \neq y$, we have $\delta(x,y,H) \leq |H|/n$.

Lemma 14 For all $x \in M$, $S \subseteq M$, and random $h \in H$,

$$\mathbf{E}[\delta(x,S,h)] \leq \frac{|S|}{n}.$$

Proof: The following simple calculation constitute the proof.

$$\begin{aligned}
\mathbf{E}[\delta(x,S,h)] &= \sum_{h\in H}\frac{\delta(x,S,h)}{|H|} \\
&= \frac{1}{|H|}\sum_{h\in H}\sum_{y\in S}\delta(x,y,h) \\
&= \frac{1}{|H|}\sum_{y\in S}\sum_{h\in H}\delta(x,y,h) \\
&= \frac{1}{|H|}\sum_{y\in S}\delta(x,y,H) \\
&\leq \frac{1}{|H|}\sum_{y\in S}\frac{|H|}{n} \\
&= \frac{|S|}{n}
\end{aligned}$$

∎

We now apply this lemma to the analysis of a dynamic dictionary scheme based on a 2-universal family H. The algorithm first chooses a hash function $h \in H$ uniformly at random, and then processes updates and queries using this h. Each key x is stored at the location $h(x)$, and there could be many keys stored in that location due to possible collisions. The keys colliding at a given location are stored in a linked list and a pointer to the head of the list is maintained in that cell. The amount of time required to perform an insert, delete or query operation involving a key x is at most one more than the length of the linked list stored at the location $h(x)$. Assuming that the set of items currently stored in the table is $S \subseteq M$, the length of the linked list is given by $\delta(x,S,h)$ which has expected value $|S|/n$. Of course, we could actually use a balanced binary search tree instead of a linked list to reduce the cost of each operation to $\mathrm{O}(\log \delta(x,S,h))$, but this hardly seems worth the effort given that we expect the number of collisions at each table location to be fairly small.

Consider a request sequence $R = R_1R_2\ldots R_r$ of update and query operations starting with an empty hash table. Suppose that this sequence contains s insert operations; then, the table will never contain more than s keys. Let $\rho(h,R)$ denote the total cost of processing these requests using the hash function $h \in H$ and the linked list scheme for collision resolution. The following theorem is easy to prove.

Theorem 15 For any sequence R of length r with s inserts, and h chosen uniformly at random from a 2-universal family H,

$$\mathbf{E}[\rho(h,R)] \leq r\left(1+\frac{s}{n}\right).$$

If we pick the table size n to be larger than the maximum number of elements ever present in the table, this gives an expected time per operation of at most 2. By Markov's inequality, the probability that the total cost of the request sequence will exceed $2rt$ is at most $1/t$. We emphasize that this analysis does not assume anything about the request sequence R except a bound on its length, and therefore it gives a more desirable solution than the standard hashing strategies which assume some probability distribution on the input.

Let us now turn to the question of constructing 2-universal families. We start by showing that our definition of 2-universality is essentially the best possible since significantly smaller collision probability cannot be obtained for $m \gg n$.

Theorem 16 For any family H of functions from M to N, there exist $x,\ y \in M$ such that

$$\delta(x, y, H) > \frac{|H|}{n} - \frac{|H|}{m}.$$

Proof: Fix some function $h \in H$, and for each $z \in N$ define the set of elements of M mapped to z as

$$A_z = \{x \in M \mid h(x) = z\}$$

Note that the sets A_z, for $z \in N$, form a partition of M. Define $\delta(X, Y, h) = \sum_{x \in X} \delta(x, Y, h)$. It is easy to verify that

$$\delta(A_w, A_z, h) = \begin{cases} 0 & w \neq z \\ |A_z|(|A_z| - 1) & w = z \end{cases}$$

This is because any two elements which collide must belong to the same set A_z, and the number of collisions within the elements of A_z is exactly $|A_z|(|A_z| - 1)$. It is then clear that the total number of collisions between all possible pairs of elements is minimized when these sets A_z are all of the same size. We obtain that

$$\begin{aligned} \delta(M, M, h) &= \sum_{z \in N} |A_z|(|A_z| - 1) \\ &\geq n\left(\frac{m}{n}\left(\frac{m}{n} - 1\right)\right) = m^2\left(\frac{1}{n} - \frac{1}{m}\right). \end{aligned}$$

This calculation was for any fixed choice of $h \in H$, and so $\delta(M, M, H) = \sum_{h \in H} \delta(M, M, h) \geq |H| m^2 (1/n - 1/m)$. By the pigeon-hole principle there must exist a pair of elements $x,\ y \in M$ such that

$$\begin{aligned} \delta(x, y, H) &\geq \frac{\delta(M, M, H)}{m^2} \\ &\geq \frac{|H| m^2 \left(\frac{1}{n} - \frac{1}{m}\right)}{m^2} \\ &\geq |H|\left(\frac{1}{n} - \frac{1}{m}\right) \end{aligned}$$

∎

Our construction of a 2-universal family is based on algebraic ideas. Fix m and n, and choose a prime $p \geq m$. We will work over the field $\mathcal{Z}_p = \{0, 1, \ldots, p-1\}$. Let $g : \mathcal{Z}_p \to N$ be the function given by $g(x) = x \pmod{n}$. For all $a,\ b \in \mathcal{Z}_p$ define the linear function $f_{a,b} : \mathcal{Z}_p \to \mathcal{Z}_p$ and the hash function $h_{a,b} : \mathcal{Z}_p \to N$ as follows.

$$\begin{aligned} f_{a,b}(x) &= ax + b \pmod{p} \\ h_{a,b}(x) &= g(f_{a,b}(x)) \end{aligned}$$

We define the family of hash functions $H = \{h_{a,b} \mid a, b \in \mathcal{Z}_p \text{ with } a \neq 0\}$ and claim that it is 2-universal. Note that although H uses $\mathcal{Z}_p$ as its domain, the claim applies to the restriction of H to any subset of $\mathcal{Z}_p$, in particular to the domain M.

Lemma 17 For all x, $y \in \mathcal{Z}_p$ such that $x \neq y$,

$$\delta(x, y, H) = \delta(\mathcal{Z}_p, \mathcal{Z}_p, g).$$

Proof: We would like to show that the number of hash functions in H which cause x and y to collide is exactly the size of the residue classes of $\mathcal{Z}_p$ modulo n. Suppose that the elements x and y collide under a specific function $h_{a,b}$. Let $f_{a,b}(x) = r$ and $f_{a,b}(y) = s$. A collision takes place if and only if $g(r) = g(s)$, or $r \equiv s \pmod{n}$. Now, having fixed x and y, for each such choice of r and s the values of a and b are uniquely determined as the solution to the following system of linear equations over the field $\mathcal{Z}_p$.

$$\begin{aligned} ax + b &\equiv r \pmod{p} \\ ay + b &\equiv s \pmod{p} \end{aligned}$$

Thus, the number of hash functions $h_{a,b}$ which cause a collision between x and y is exactly the number of choices of r and s such that $r \equiv s \pmod{n}$. The latter is given by $\delta(\mathcal{Z}_p, \mathcal{Z}_p, g)$. ∎

Given the similarity of the definition of 2-universality to pairwise independence, it is not surprising that the constructions and their proofs are also very similar. (You might wish to learn more about the notion of pairwise independence if it is new to you.)

Theorem 18 The family $H = \{h_{a,b} \mid a, b \in \mathcal{Z}_p \text{ with } a \neq 0\}$ is a 2-universal family.

Proof: For each $z \in N$, let $A_z = \{x \in \mathcal{Z}_p \mid g(x) = z\}$; it is clear that $|A_z| \leq \lceil p/n \rceil$. In other words, for every $r \in \mathcal{Z}_p$ there are at most $\lceil p/n \rceil$ different choices of $s \in \mathcal{Z}_p$ such that $g(r) = g(s)$. Since there are a total of p different choices of $r \in \mathcal{Z}_p$ to start with, we obtain that

$$\delta(\mathcal{Z}_p, \mathcal{Z}_p, g) \leq p(\left\lceil \frac{p}{n} \right\rceil - 1) \leq \frac{p(p-1)}{n}.$$

Lemma 17 now implies that for any distinct x and y in $\mathcal{Z}_p$, $\delta(x, y, H) \leq p(p-1)/n$. Since the size of $|H|$ is exactly $p(p-1)$, this gives the desired result that $\delta(x, y, H) \leq |H|/n$. ∎

Since we can always choose $p = \mathrm{O}(m)$, the number of random bits needed to sample a hash function from H is no more than $2 \log p = \mathrm{O}(\log m)$. Observe that choosing, storing and evaluating hash functions from H is remarkably simple and efficient. Just pick a and b independently and uniformly at random from $\mathcal{Z}_p$, these are stored using very little memory, and then computing $h_{a,b}$ is a trivial task. Contrast this with the situation where a totally random function is used as a hash function.

Exercise 19 In this exercise we will consider an alternate construction of 2-universal families of hash functions. For each $a \in \mathcal{Z}_p$, define the function $f_a(x) = ax \pmod{p}$, and $h_a(x) = g(f_a(x))$, and let $H = \{h_a \mid a \in \mathcal{Z}_p,\ a \neq 0\}$. Show that H is nearly-2-universal in that, for all $x \neq y$,

$$\delta(x, y, H) \leq \frac{2|H|}{n}.$$

Also, show that the bound on the collision probability is close to the best possible for this family of hash functions.