Alcom-FT Technical Report Series
ALCOMFT-TR-02-66

To appear in proceedings of ICALP 2002. psy227 Springer Verlag.
Available on-line at http://www.brics.dk/ pagh/papers/

One-Probe Search[1]

Anna Östlin and Rasmus Pagh

BRICS[2], Department of Computer Science
University of Aarhus, Denmark
{annao,pagh}@brics.dk

Abstract. We consider dictionaries that perform lookups by probing a *single word* of memory, knowing only the size of the data structure. We describe a randomized dictionary where a lookup returns the correct answer with probability $1-\epsilon$, and otherwise returns "don't know". The lookup procedure uses an expander graph to select the memory location to probe. Recent explicit expander constructions are shown to yield space usage far smaller than what would be required using a deterministic lookup procedure. Our data structure supports efficient *deterministic* updates, exhibiting new probabilistic guarantees on dictionary running time.

1 Introduction

The *dictionary* is one of the most well-studied data structures. A dictionary represents a set S of elements (called *keys*) from some universe U, along with information associated with each key in the set. Any $x \in U$ can be looked up, i.e., it can be reported whether $x \in S$, and if so, what information is associated with x. We consider this problem on a unit cost word RAM in the case where keys and associated information have fixed size and are not too big (see below). The most straightforward implementation, an array indexed by the keys, has the disadvantage that the space usage is proportional to the size of U rather than to the size of S. On the other hand, arrays are extremely time efficient: A single memory probe suffices to retrieve or update an entry.

It is easy to see that there exists no better deterministic one-probe dictionary than an array. In this paper we investigate *randomized* one-probe search strategies, and show that it is possible, using much less space than an array implementation, to look up a given key with probability arbitrarily close to 1. The probability is over coin tosses performed by the lookup procedure. In case the memory probe did not supply enough information to answer the query, this is realized by the lookup procedure, and it produces the answer "don't know".

[1] Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
[2] Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

In particular, by iterating until an answer is found, we get a Las Vegas lookup procedure that can have an expected number of probes arbitrarily close to 1.

It should be noted that one-probe search is impossible if one has no idea how much data is stored. We assume that the query algorithm knows the size of the data structure – a number that only changes when the size of the key set changes by a constant factor. The fact that the size, which may rarely or never change, is the only kind of global information needed to query the data structure means that it is well suited to support concurrent lookups (in parallel or distributed settings). In contrast, all known hash function based lookup schemes have some kind of global hash function that must be changed regularly. Even concurrent lookup of the *same* key, without accessing the same memory location, is supported to some extent by our dictionary. This is due to the fact that two lookups of the same key are not very likely to probe the same memory location.

A curious feature of our lookup procedure is that it makes its decision based on a constant number of *equality* tests – in this sense it is comparison-based. However, the data structure is not *implicit* in the sense of Munro and Suwanda [11], as it stores keys not in S.

Our studies were inspired by recent work of Buhrman et al. [4] on randomized analogs of bit vectors. They presented a Monte Carlo data structure where one bit probe suffices to retrieve a given bit with probability arbitrarily close to 1. When storing a sparse bit vector (few 1s) the space usage is much smaller than that of a bit vector. When storing no associated information, a dictionary solves the *membership* problem, which can also be seen as the problem of storing a bit vector. Our Las Vegas lookup procedure is stronger than the Monte Carlo lookup procedure in [4], as a wrong answer is never returned. The price paid for this is an *expected* bound on the number of probes, a slightly higher space usage, and, of course, that we look up one word rather than one bit. The connection to [4] is also found in the underlying technique: We employ the same kind of unbalanced bipartite expander graph as is used there. Recently, explicit constructions[3] of such graphs with near-optimal parameters have been found [14, 15].

Let $u = |U|$ and $n = |S|$. We assume that one word is large enough to hold one of $2u + 1$ different symbols plus the information associated with a key. (Note that if this is not the case, it can be simulated by accessing a number of consecutive words rather than one word – an efficient operation in many memory models.) Our main theorem is the following:

Theorem 1. *For any constant $\epsilon > 0$ there exists a nonexplicit one-probe dictionary with success probability $1 - \epsilon$, using $O(n \log \frac{2u}{n})$ words of memory. Also, there is an explicit construction using $n \cdot 2^{O((\log \log u)^3)}$ words of memory.*

Note that the space overhead for the nonexplicit scheme, a factor of $\log \frac{2u}{n}$, is exponentially smaller than that of an array implementation.

In the second part of the paper we consider dynamic updates to the dictionary (insertions and deletions of keys). The fastest known dynamic dictionaries use

[3] Where a given neighbor of a vertex can be computed in time polylogarithmic in the number of vertices.

hashing, i.e., they select at random a number of functions from suitable families, which are stored and subsequently used deterministically to direct searches.

A main point in this paper is that a fixed structure with random properties (the expander graph) can be used to move random choices from the data structure itself to the lookup procedure. The absence of hash functions in our data structure has the consequence that updates can be performed in a very local manner. We show how to deterministically perform updates by probing and changing a number of words that is nearly linear in the degree of the expander graph (which, for optimal expanders, is at most logarithmic in the size of the universe). Current explicit expanders are not fast enough for our dynamic data structure to improve known results in a standard RAM model. However, if we augment the RAM with an instruction for computing neighbors in an optimal expander graph with given numbers of vertices, an efficient dynamic dictionary can be implemented.

Theorem 2. *In the expander-augmented RAM model, there is a dictionary where a sequence of a insertions/deletions and b lookups in a key set of size at most n takes time $O(a(\log \frac{2u}{n})^{1+o(1)} + b + t)$ with probability $1 - 2^{-\Omega(a+t/(\log \frac{2u}{n})^{1+o(1)})}$. The space usage is $O(n \log \frac{2u}{n})$ words.*

When the ratio between the number of updates and lookups is small, the expected average time per dictionary operation is constant. Indeed, if the fraction of updates is between $(\log \frac{2u}{n})^{-1-\Omega(1)}$ and $n^{-\omega(1)}$, and if $u = 2^{n^{1-\Omega(1)}}$, the above yields the best known probability, using space polynomial in n, that a sequence of dictionary operations take average constant time. The intuitive reason why the probability bound is so good, is that time consuming behavior requires bad random choices in many invocations of the lookup procedure, and that the random bits used in different invocations are *independent*.

1.1 Related Work

As described above, this paper is related to [4], in scope as well as in tools. The use of expander graphs in connection with the membership problem was earlier suggested by Fiat and Naor [6], as a tool for constructing an efficient implicit dictionary.

Yao [16] showed an $\Omega(\log n)$ worst case lower bound on the time for dictionary lookups on a restricted RAM model allowing words to contain only keys of S or special symbols from a fixed set whose size is a function of n (e.g., pointers). The lower bound holds when space is bounded by a function of n, and u is sufficiently large. It extends to give an $\Omega(\log n)$ lower bound for the expected time of randomized Las Vegas lookups.

Our data structure violates Yao's lower bound model in two ways: 1. We allow words to contain certain keys not in S (accessed only through equality tests); 2. We allow space depending on u. The second violation is the important one, as Yao's lower bound can be extended to allow 1. Yao also considered deterministic one-probe schemes in his model, showing that, for $n \leq u/2$, a space usage of $u/2 + O(1)$ words is necessary and sufficient for them to exist.

The worst case optimal number of word probes for membership was studied by Pagh in [13] in the case where U equals the set of machine words. It was shown that *three* word probes are necessary when using m words of space, unless $u = 2^{\Omega(n^2/m)}$ or $u \leq n^{2+o(1)}$. Sufficiency of three probes was shown for all parameters (in most cases it followed by the classic dictionary of Fredman et al. [7]). In the expected sense, most hashing based dictionaries can be made to use arbitrarily close to 2 probes per lookup by expanding the size of the hash table by a constant factor.

Dictionaries with sublogarithmic lookup time that also allow efficient deterministic updates have been developed in a number of papers [1, 2, 8, 9, 12]. Let n denote an upper bound on the number of keys in a dynamic dictionary. For lookup time $t = o(\log\log n)$, the best known update time is $n^{O(1/t)}$, achieved by Hagerup et al. [9]. The currently best *probabilistic* guarantee on dynamic dictionary performance, first achieved by Dietzfelbinger and Meyer auf der Heide in [5], is that each operation takes constant time with probability $1 - O(m^{-c})$, where c is any constant and m is the space usage in words (which must be some constant factor larger than n). This implies that a sequence of a insertions/deletions and b lookups takes time $O(a + b + t)$ with probability $1 - O(m^{-t/n})$.

2 Preliminaries

In this section we define (n, d, ϵ)-expander graphs and state some results concerning these graphs. For the rest of this paper we will assume ϵ to be a multiple of $1/d$, as this makes statements and proofs simpler. This will be without loss of generality, as the statements we show do not change when rounding ϵ down to the nearest multiple of $1/d$.

Let $G = (U, V, E)$ be a bipartite graph with left vertex set U, right vertex set V, and edge set E. We denote the set of neighbors of a set $S \subseteq U$ by $\Gamma(S) = \bigcup_{s \in S}\{v \mid (s, v) \in E\}$. We use $\Gamma(x)$ as a shorthand for $\Gamma(\{x\})$, $x \in U$.

Definition 3. *A bipartite graph $G = (U, V, E)$ is d-regular if the degree of all nodes in U is d. A bipartite d-regular graph $G = (U, V, E)$ is an (n, d, ϵ)-expander if for each $S \subseteq U$ with $|S| \leq n$ it holds that $|\Gamma(S)| \geq (1 - \epsilon)d|S|$.*

Lemma 4. *For $0 < \epsilon < 1$ and $d \geq 1$, if $|V| \geq (1-\epsilon)dn(2u/n)^{1/\epsilon d}e^{1/\epsilon}$ then there exists an (n, d, ϵ)-expander graph $G = (U, V, E)$, where $|U| = u$.*

Proof. Our proof is a standard application of the probabilistic method. Let $G = (U, V, E)$ be a randomly generated graph created by the following procedure. For each $u \in U$ choose d neighbors with replacement, i.e., an edge can be chosen more than once, but then the double edges are removed. We will argue that the probability that this graph fails to be a (n, d, ϵ)-expander graph is less than 1 for the choices of $|V|$ and d as stated in the lemma. The degrees of the nodes in U in this graph may be less than d, but if there exists a graph that is expanding with degree at most d for all nodes, then there clearly exists a graph that is expanding with exactly degree d as well.

We must bound the probability that some subset of $i \leq n$ vertices from U has fewer than $(1-\epsilon)di$ neighbors. A subset $S \subseteq U$ of size i can be chosen in $\binom{u}{i}$ ways and a set $V' \subseteq V$ of size $(1-\epsilon)di$ can be chosen in $\binom{|V|}{(1-\epsilon)di}$ ways. (Note that $|V| \geq (1-\epsilon)di$.) The probability that such a set V' contains all of the neighbors for S is $(\frac{(1-\epsilon)di}{|V|})^{di}$. Thus, the probability that some subset of U of size $i \leq n$ has fewer than $(1-\epsilon)di$ neighbors is at most

$$\begin{aligned}
&\sum_{i=1}^{n} \binom{u}{i} \binom{|V|}{(1-\epsilon)di} \left(\frac{(1-\epsilon)di}{|V|}\right)^{di} \\
&< \sum_{i=1}^{n} \left(\frac{ue}{i}\right)^{i} \left(\frac{|V|e}{(1-\epsilon)di}\right)^{(1-\epsilon)di} \left(\frac{(1-\epsilon)di}{|V|}\right)^{di} \\
&\leq \sum_{i=1}^{n} \left(\left(\frac{(1-\epsilon)di}{|V|}\right)^{\epsilon d} e^{d} u/i\right)^{i} .
\end{aligned}$$

If the term in the outermost parentheses is bounded by 1/2, the sum is less than 1. This is the case when $|V|$ fulfills the requirement stated in the lemma.□

Corollary 5. *For any constants $\alpha, \epsilon > 0$ there exist an (n, d, ϵ)-expander $G = (U, V, E)$ for the following parameters:*

- $|U| = u$, $d = O(\log(2u/n))$ *and* $|V| = O(n \log(2u/n))$.
- $|U| = u$, $d = O(1)$ *and* $|V| = O(n\,(2u/n)^{\alpha})$.

Theorem 6. *(Ta-Shma [14]) For any constant $\epsilon > 0$ and for $d = 2^{O((\log\log u)^3)}$, there exists an* explicit *(n, d, ϵ)-expander $G = (U, V, E)$ with $|U| = u$ and $|V| = n \cdot 2^{O((\log\log u)^3)}$.*

3 Static Data Structure

Let $S \subseteq U$ denote the key set we wish to store. Our data structure is an array denoted by T. Its entries may contain the symbol x for keys $x \in S$, the symbol $\neg x$ for keys $x \in U \backslash S$, or the special symbol $\perp \notin U$. (Recall our assumption that one of these symbols plus associated information fits into one word.) For simplicity we will consider the case where there is no information associated with keys, i.e., we solve just the membership problem. Extending this to allow associated information is straightforward. We make use of a $(2n+1, d, \epsilon/2)$-expander with neighbor function Γ. Given that a random element in the set $\Gamma(x)$ can be computed quickly for $x \in U$, the one-probe lookup procedure is very efficient.

procedure lookup$_\epsilon(x)$
  choose $v \in \Gamma(x)$ at random;
  if $T[v] = x$ **then return** *'yes'*
  else if $T[v] \in \{\neg x, \perp\}$ **then return** *'no'*
  else return *'don't know'*;
end;

The corresponding Las Vegas lookup algorithm is the following:

> **procedure** lookup(x)
>   **repeat**
>   choose $v \in \Gamma(x)$ at random;
>   **until** $T[v] \in \{x, \neg x, \perp\}$;
>   **if** $T[v] = x$ **then return** *'yes'* **else return** *'no'*;
> **end**;

3.1 Requirements to the Data Structure

The success probability of lookup$_\epsilon(x)$ and the expected time of lookup(x) depends on the content of the entries indexed by $\Gamma(x)$ in T. To guarantee correctness and success probability $1-\epsilon$ in each probe for x, the following conditions should hold:

1. If $x \in S$, at least a fraction $1-\epsilon$ of the entries $T[v]$, $v \in \Gamma(x)$, contain x, and none contain $\neg x$ or $\perp$.
2. If $x \notin S$, at least a fraction $1-\epsilon$ of the entries $T[v]$, $v \in \Gamma(x)$, contain either $\neg x$ or $\perp$, and none contain x.

By inserting $\perp$ in all entries of T except the entries in $\Gamma(S)$, condition 2 will be satisfied for all $x \notin S$ with $|\Gamma(x) \cap \Gamma(S)| \leq \epsilon d$. A key notion in this paper is the set of ϵ-*ghosts* for a set S, which are the keys of U that have many neighbors in common with S. For each ϵ-ghost x we will need some entries in T with content $\neg x$.

Definition 7. *Given a bipartite graph $G = (U, V, E)$, a key $x \in U$ is an* ϵ-ghost *for the set $S \subseteq U$ if $|\Gamma(x) \cap \Gamma(S)| > \epsilon|\Gamma(x)|$ and $x \notin S$.*

Lemma 8. (Buhrman et al. [4]) *There are at most n ϵ-ghosts for a set S of size n in a $(2n+1, d, \epsilon/2)$-expander graph.*

In order to fulfill conditions 1 and 2, we need to assign entries in T to the keys in S and to the ϵ-ghosts for S.

Definition 9. *Let $G = (U, V, E)$ be a bipartite d-regular graph and let $0 < \epsilon < 1$. An* assignment *for a set $S \subseteq U$, is a subset $A \subseteq E \cap (S \times \Gamma(S))$ such that for $v \in \Gamma(S)$, $|A \cap (S \times \{v\})| = 1$. A* $(1-\epsilon)$-balanced assignment *for S is an assignment A, where for each $s \in S$ it holds that $|A \cap (\{s\} \times \Gamma(s))| \geq (1-\epsilon)d$.*

Lemma 10. *If a graph $G = (U, V, E)$ is an (n, d, ϵ)-expander then there exists a $(1-\epsilon)$-balanced assignment for every set $S \subseteq U$ of size at most n.*

To show the lemma we will use Hall's theorem [10]. A *perfect matching* in a bipartite graph (U, V, E) is a set of $|U|$ edges such that for each $x \in U$ there is an edge $(x, v) \in E$, and for each $v \in V$ there is at most one edge $(x, v) \in E$.

Theorem 11. (Hall's theorem) *In any bipartite graph $G = (U, V, E)$, where for each subset $U' \subseteq U$ it holds that $|U'| \leq |\Gamma(U')|$, there exists a perfect matching.*

Proof of Lemma 10. Let S be an arbitrary subset of U of size n. Let $G' = (S, \Gamma(S), E')$ be the subgraph of G induced by the nodes S and $\Gamma(S)$, i.e., $E' = \{(s, v) \in E \mid s \in S\}$. To prove the lemma we want to show that there exists an assignment A such that for each $s \in S$, $|A \cap (\{s\} \times \Gamma(s))| \geq (1 - \epsilon)d$. The idea is to use Hall's theorem $(1 - \epsilon)d$ times by repeatedly finding a perfect matching and removing the nodes from $\Gamma(S)$ in the matching.

Since G is an (n, d, ϵ)-expander we know that for each subset $S' \subseteq S$ it holds that $|\Gamma(S')| \geq (1 - \epsilon)d|S'|$. Assume that we have i perfect matchings from S to non-overlapping subsets of $\Gamma(S)$ and denote by M the nodes from $\Gamma(S)$ in the matchings. For each subset $S' \subseteq S$ it holds that $|\Gamma(S') \backslash M| \geq ((1 - \epsilon)d - i)|S'|$. If $(1 - \epsilon)d - i \geq 1$ then the condition in Hall's theorem holds for the graph $G_i = (S, (\Gamma(S) \backslash M), E' \backslash E_i)$, where E_i is the set of edges incident to nodes in M, and there exists a perfect matching in G_i. From this it follows that at least $(1 - \epsilon)d$ non-overlapping (in $\Gamma(S)$) perfect matchings can be found in G'. The edges in the matchings define a $(1 - \epsilon)$-balanced assignment. □

3.2 Construction

We store the set S as follows:

1. Write $\perp$ in all entries not in $\Gamma(S)$.
2. Find the set $\bar{S}$ of ϵ-ghosts for S.
3. Find a $(1 - \epsilon)$-balanced assignment for the set $S \cup \bar{S}$.
4. For $x \in S$ write x in entries assigned to x.
   For $x \in \bar{S}$ write $\neg x$ in entries assigned to x in $\Gamma(S)$.

By Lemma 8 the set $\bar{S}$ found in step 2 contains at most n keys, and by Lemma 10 it is possible to carry out step 3. Together with the results on expanders in Section 2, this concludes the proof of Theorem 1.

We note that step 2 takes time $\Omega(|U \backslash S|)$ if we have only oracle access to Γ. When the graph has some structure it is sometimes possible to do much better. Ta-Shma shows in [14] that this step can be performed for his class of graphs in time polynomial in the size of the right vertex set, i.e., polynomial in the space usage. All other steps are clearly also polynomial time in the size of the array.

In the dynamic setting, covered in Section 4, we will take an entirely different approach to ghosts, namely, we care about them only if we see them. We then argue that the time spent looking for a particular ghost before it is detected is not too large, and that there will not be too many different ghosts.

4 Dynamic Updates

In this section we show how to implement efficient dynamic insertions and deletions of keys in our dictionary. We will use a slightly stronger expander graph than in the static case, namely a $(4n', d, \epsilon/3)$-expander where n' is an upper bound on the size of the set that can be handled. The parameter n' is assumed to be known to the query algorithm. Note that n' can be kept in the range, say, n to $2n$ at no asymptotic cost, using standard global rebuilding techniques. Our dynamic dictionary essentially maintains the static data structure described in the previous section. Additionally, we maintain the following auxiliary data structures:

- A priority queue with all keys in S plus some set $\bar{S}$ of keys that appear negated in T. Each key has as priority the size of its assignment, which is always at least $(1-\epsilon)d$.
- Each entry $T[v]$ in T is augmented with
  - A pointer $T_p[v]$ which, if entry v is assigned to a key, points to that key in the priority queue.
  - A counter $T_c[v]$ that at any time stores the number of keys in S that have v as a neighbor.

Since all keys in the priority queue are assigned $(1-\epsilon)d$ entries in T, the performance of the lookup procedure is the desired one, except when searching for ϵ-ghosts not in $\bar{S}$. We will discuss this in more detail later.

4.1 Performing Updates

We first note that it is easy to maintain the data structure during deletions. All that is needed when deleting $x \in S$ is decreasing the counters $T_c[v]$, $v \in \Gamma(x)$, and replacing x with $\neg x$ or $\perp$ (the latter if the counter reaches 0). Finally, x should be removed from the priority queue. We use a simple priority queue that requires space $O(d+n)$, supports `insert` in $O(d)$ time, and `increasekey`, `decreasekey`, `findmin` and `delete` in $O(1)$ time. The total time for a deletion in our dictionary is $O(d)$.

When doing insertions we have to worry about maintaining a $(1-\epsilon)$-balanced assignment. The idea of our insertion algorithm is to assign *all* neighbors to the key being inserted. In case this makes the assignment of other keys too small (easily seen using the priority queue), we repeat assigning all neighbors to them, and so forth. Every time an entry in T is reassigned to a new key, the priority of the old and new key are adjusted in the priority queue. The time for an insertion is $O(d)$, if one does not count the associated cost of maintaining assignments of other keys. The analysis in Section 4.2 will bound this cost. Note that a priori it is not even clear whether the insertion procedure terminates.

A final aspect that we have to deal with is ghosts. Ideally we would like $\bar{S}$ to be at all times the current set of ϵ-ghosts for S, such that a $(1-\epsilon)$-balanced assignment was maintained for all ghosts. However, this leaves us with the hard problem of finding new ghosts as they appear. We circumvent this problem by only including keys in $\bar{S}$ if they are selected for examination and found to be ϵ-ghosts. A key is selected for examination if a lookup of that key takes more than $\log_{1/\epsilon} d$ iterations. The time spent on examinations and on lookups of a ghost before it is found, is bounded in the next section.

The sequence of operations is divided up into stages, where each stage (except possibly the last) contains n' insert operations. After the last insertion in a stage, all keys in $\bar{S}$ that are no longer ϵ-ghosts are deleted. This is done by going through all keys in the priority queue. Keys of $\bar{S}$ with at least $(1-\epsilon)d$ neighbors containing $\perp$ are removed from the priority queue. Hence, when a new stage starts, $\bar{S}$ will only contain ϵ-ghosts.

4.2 Analysis

We now sketch the analysis of our dynamic dictionary. First, the total work spent doing assignments and reassignments is analyzed. Recall that the algorithm maintains a $(1-\epsilon)$-balanced assignment for the set $S \cup \bar{S}$ of keys in the priority queue. Keys enter the priority queue when they are inserted in S, and they may enter it when they are ϵ-ghosts for the current set. It clearly suffices to bound the total work in connection with insertions in the priority queue, as the total work for deletions cannot be larger than this. We will first show a bound on the number of keys in $\bar{S}$.

Lemma 12. *The number of keys in the set $\bar{S}$ never exceeds $2n'$.*

Proof. Let S be the set stored at the beginning of a stage. $\bar{S}$ only contains ϵ-ghosts for S at this point. Let S' denote the keys inserted during the stage. New keys inserted into $\bar{S}$ have to be ϵ-ghosts for $S \cup S'$. According to Lemma 8, the fact that $|S \cup S'| \leq 2n'$ implies that there are at most $2n'$ ϵ-ghosts for $S \cup S'$ (including the ϵ-ghosts for S). Thus, the number of keys in $\bar{S}$ during any stage is at most $2n'$. □

It follows from the lemma that the number of insertions in the priority queue is bounded by 3 times the number of insertions performed in the dictionary. The remainder of our analysis of the number of reassignments has two parts: We first show that our algorithm performs a number of reassignments (in connection with insertions) that is within a constant factor of *any* scheme maintaining a $(1-\epsilon/3)$-balanced assignment. The scheme we compare ourselves to may be *off-line*, i.e., know the sequence of operations in advance. Secondly, we give an off-line strategy for maintaining a $(1-\epsilon/3)$-balanced assignment using $O(d)$ reassignments per update. This proof strategy was previously used for an assignment problem by Brodal and Fagerberg [3].

In the following lemmas, the set M is the set for which a balanced assignment is maintained, and the insert and delete operations are insertions and deletions in this set. In our data structure M corresponds to $S \cup \bar{S}$.

Lemma 13. *Let $G = (U, V, E)$ be a d-regular bipartite graph. Suppose O is a sequence of insert and delete operations on a dynamic set $M \subseteq U$. Let B be an algorithm that maintains a $(1-\frac{\epsilon}{3})$-balanced assignment for M, and let C be our "assign all" scheme for maintaining a $(1-\epsilon)$-balanced assignment for M. If B makes at most k reassignments during O, then C assigns all neighbors to a key at most $\frac{3}{\epsilon}(k/d + |M|_{\text{start}})$ times, where $|M|_{\text{start}}$ is the initial size of M.*

Proof. To show the lemma we will argue that the assignment of C, denoted A_C, will become significantly "less different" from the assignment of B, denoted A_B, each time C assigns all neighbors of a key to that key. At the beginning $|A_B \backslash A_C| \leq d|M|_{\text{start}}$, since $|A_B| \leq d|M|_{\text{start}}$. Each of the k reassignments B performs causes $|A_B \backslash A_C|$ to increase by at most one. This means that the reassignments made by C during O can decrease $|A_B \backslash A_C|$ by at most $k + d|M|_{\text{start}}$ in total.

Each time C assigns all entries in $\Gamma(x)$ to a key x, at least ϵd reassignments are done, since the assignment for x had size less than $(1-\epsilon)d$ before the reassignment. At this point at least $(1 - \frac{\epsilon}{3})d$ pairs (x, e) are included in A_B, i.e., at most $\frac{\epsilon}{3}d$ of the neighbors of x are not assigned to x in A_B. This means that at least $\frac{2\epsilon}{3}d$ of the reassignments made by C decrease $|A_B \backslash A_C|$, while at most $\frac{\epsilon}{3}d$ reassignments may increase $|A_B \backslash A_C|$. In total, $|A_B \backslash A_C|$ is decreased by at least $\frac{\epsilon}{3}d$ when C assigns all neighbors to a key. The lemma now follows, as $|A_B \backslash A_C|$ can decrease by $\frac{\epsilon}{3}d$ at most $(k + d|M|_{\text{start}})/(\frac{\epsilon}{3}d)$ times. □

Lemma 14. *Let $G = (U, V, E)$ be a $(4n', d, \epsilon/3)$-expander. There exists an off-line algorithm maintaining a $(1-\frac{\epsilon}{3})$-balanced assignment for a dynamic set $M \subseteq U$, during a stage of $3n'$ insertions, by performing at most $4dn'$ reassignments, where $|M| \leq n'$ at the beginning of the stage.*

Proof. Let M' be the set of $3n'$ keys to insert. Define $\tilde{M} = M \cup M'$; we have $|\tilde{M}| \leq 4n'$. Let $A_{\tilde{M}}$ be a $(1 - \frac{\epsilon}{3})$-balanced assignment for $\tilde{M}$ (shown to exist in Lemma 10).

The off-line algorithm knows the set M' of keys to insert from the start, and does the following. First, it assigns neighbors to the keys in M according to the assignment $A_{\tilde{M}}$, which requires at most dn' reassignments. Secondly, for each insertion of a key $x \in M'$, it assigns neighbors to x according to $A_{\tilde{M}}$, which requires at most d reassignments. This will not cause any key already in the set to lose an assigned neighbor, hence no further reassignments are needed to keep the assignment $(1-\frac{\epsilon}{3})$-balanced. It follows that he total number of reassignments during the $3n'$ insertions is at most $4dn'$, proving the lemma. □

The above two lemmas show that in a sequence of a updates to the dictionary there are $O(a)$ insertions in the priority queue, each of which gives rise to $O(d)$ reassignments in a certain off-line algorithm, meaning that our algorithm uses $O(ad)$ time for maintaining a $(1-\epsilon)$-balanced assignment for the set $S \cup \bar{S}$ in the priority queue.

We now turn to analyzing the work done in the lookup procedure. First we will bound the number of iterations in all searches for keys that are *not* undetected ϵ-ghosts. Each iteration has probability at least $1 - \epsilon$ of succeeding, independently of all other events, so we can bound the probability of many iterations using Chernoff bounds. In particular, the probability that the total number of iterations used in the b searches exceeds $\frac{2}{1-\epsilon}b + t$ is less than $e^{-\frac{1-\epsilon}{4}t}$.

When searching for a key that is not an undetected ϵ-ghost, the probability of selecting it for examination is bounded from above by $1/d$. In particular, by Chernoff bounds we get that, for $k > 0$, the total number of examinations during all b lookups is at most $b/d + k$ with probability $1 - (\frac{e}{1+kd/b})^{b/d+k}$. For $k = (2e-1)b/d + t/d$ we get that the probability of more than $2eb/d + t/d$ examinations is bounded by $2^{-t/d}$. Each examination costs time $O(d)$, so the probability of spending $O(b+t)$ time on such examinations is at least $1 - 2^{-t/d}$.

We now bound the work spent on finding ϵ-ghosts. Recall that an ϵ-ghost is detected if it is looked up, and the number of iterations used by the lookup procedure exceeds $\log_{1/\epsilon} d$. Since we have an ϵ-ghost, the probability that a single lookup selects the ghost for examination is at least $\epsilon^{\log_{1/\epsilon} d - 1} = \Omega(1/d)$. We define $d' = O(d)$ by $1/d' = \epsilon^{\log_{1/\epsilon} d - 1}$. Recall that there are at most $2n'$ ϵ-ghosts in a stage, and hence at most $2a$ in total. We bound the probability that more than $4ad' + k$ lookups are made on undetected ϵ-ghosts, for $k > 0$. By Chernoff bounds the probability is at most $e^{-k/4d'}$. Each lookup costs $O(\log d)$ time, so the probability of using time $O(ad \log d + t)$ is at least $1 - e^{-t/4d' \log d}$.

In summary, we have bounded the time spent on four different tasks in our dictionary:

- The time spent looking up keys that are not undetected ϵ-ghosts is $O(b+t)$ with probability $1 - 2^{-\Omega(t)}$.
- The time spent examining keys that are not undetected ϵ-ghosts is $O(b+t)$ with probability $1 - 2^{-\Omega(t/d)}$.
- The time spent looking up ϵ-ghosts before they are detected is $O(ad \log d + t)$ with probability $1 - 2^{-\Omega(t/d \log d)}$.
- The time spent assigning, reassigning and doing bookkeeping is $O(ad)$.

Using the above with the first expander of Corollary 5, having degree $d = O(\log \frac{2u}{n'})$, we get the performance bound stated in Theorem 2. Using the constant degree expander of Corollary 5 we get a data structure with constant time updates. This can also be achieved in this space with a trie, but a trie would use around $1/\alpha$ word probes for lookups of keys in the set, rather than close to 1 word probe, expected.

5 Conclusion and Open Problems

In this paper we studied dictionaries for which a single word probe with good probability suffices to retrieve any given key with associated information. The main open problem we leave is whether the space usage of our dictionary is the best possible for one-probe search.

It is known that three word probes are necessary and sufficient in the worst case for lookups in dictionaries, even when using superlinear space. An obvious open question is how well one can do using two word probes and a randomized lookup procedure. Can the space utilization be substantially improved? Another point is that we bypass Yao's lower bound by using space dependent on u. An interesting question is: How large a dependence on u is necessary to get around Yao's lower bound. Will space $n \log^* u$ do, for example?

Acknowledgment. We thank Thore Husfeldt for helpful comments.

References

[1] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 335–342. ACM Press, 2000.

[2] Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 295–304. ACM Press, 1999.

[3] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representations of sparse graphs. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures (WADS '99)*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer-Verlag, 1999.

[4] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 449–458. ACM Press, 2000.

[5] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 1990.

[6] Amos Fiat and Moni Naor. Implicit $O(1)$ probe search. *SIAM J. Comput.*, 22(1):1–10, 1993.

[7] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.

[8] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.

[9] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

[10] Philip Hall. On representatives of subsets. *J. London Math. Soc.*, 10:26–30, 1935.

[11] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *J. Comput. System Sci.*, 21(2):236–250, 1980.

[12] Rasmus Pagh. A trade-off for worst-case efficient dictionaries. *Nordic J. Comput.*, 7(3):151–163, 2000.

[13] Rasmus Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 425–432. ACM Press, 2001.

[14] Amnon Ta-Shma. Storing information with extractors. To appear in Information Processing Letters.

[15] Amnon Ta-Shma, Christopher Umans, and David Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 143–152. ACM Press, 2001.

[16] Andrew C.-C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 28(3):615–628, 1981.