Fast Local Searches and Updates in Bounded Universes

Prosenjit Bose* Karim Douïeb* Vida Dujmović* John Howat* Pat Morin*

Abstract

Given a bounded universe $\{0, 1, \ldots, U-1\}$, we show how to perform (successor) searches in $O(\log\log\Delta)$ expected time and updates in $O(\log\log\Delta)$ expected amortized time, where Δ is the rank difference between the element being searched for and its successor in the structure. This unifies the results of traditional bounded universe structures (which support successor searches in $O(\log\log U)$ time) and hashing (which supports exact searches in $O(1)$ time). We also show how these results can be extended to answer approximate nearest neighbour queries in low dimensions.

1 Introduction

Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$. We address the problem of maintaining a dictionary subject to searches, insertions and deletions on a subset of $\mathcal{U}$ of size n. Hash tables [6] can support these operations in $O(1)$ expected time per operation. Unfortunately, hash tables only work for exact searches; they do not provide a useful result for elements that are not stored in the dictionary. Data structures for *successor searching*, however, return the element being searched for if it is present, or the successor of that element otherwise. Such searches can be supported in $O(\log\log U)$[1] time using van Emde Boas trees [11], x-fast tries or y-fast tries [12].

We introduce the idea of *local searching* in a bounded universe in order to unify these two types of structures. Our goal is to execute successor searches in time $O(\log\log\Delta)$, where Δ is the rank difference between the element being searched for and its successor in the structure. If it happens that the element being searched for is in the structure, we have $\Delta = 0$ and so the bound is equivalent to that offered by hashing for exact searches. Furthermore, we have $\Delta < U$, and so the bound is never worse than the $O(\log\log U)$ bound offered by the usual bounded universe structures.

Our results. We show how to perform such searches in $O(\log\log\Delta)$ expected time, as well as insertions and deletions in $O(\log\log\Delta)$ expected amortized time using $O(n\log\log\log U)$ expected space. For the static version of the problem, the bound holds even in the worst case. For a constant $d \geq 2$ dimensions on the universe $\mathcal{U}^d$, we show how to answer $(1+\epsilon)$-approximate nearest neighbour queries in $O\big((1/\epsilon^d)\log\log\Delta\big)$ expected time and $O(n\log\log\log U)$ expected space, where Δ is the Euclidean distance to the returned point. This structure still supports insertions and deletions in $O(\log\log\Delta)$ expected amortized time.

1.1 Related Work

In one dimension, such searches can be performed in time $O(\log\log U)$ using, for example, van Emde Boas trees [11] or y-fast tries [12]. Several results consider the distance between a query and an element in the structure. Johnson [8] describes a priority queue where insertion and deletion take time $O(\log\log D)$, where D is the difference between the successor and predecessor (in terms of priority) of the query. The idea of *finger searching* involves starting a search from somewhere inside a structure by supplying the search operation with a pointer. Skip lists [10] and finger search trees [3], for example, support finger searches in $O(\log\delta)$ time, where δ is the rank distance between the given pointer and the query. On the RAM, Kaporis et al. [9] describe a data structure with $O(1)$ worst-case update time and $O(\log\log\delta)$ expected search time for a large class of input distributions, where δ is the distance between the query and some finger. For the *temporal precedence problem*, a list is maintained under insertions. Given two pointers into the list, the data structure must decide which element was inserted first. An optimal solution is given by Brodal et al. [2] that allows the latter operation to be completed in $O(\log\log\delta)$ time, where δ is *temporal distance* between the two given elements.

In two or more dimensions, one concentrates on finding the (approximate) *nearest neighbour* of a query point. In bounded universes, Amir et al. [1] show how to answer queries and perform updates in expected time $O(\log\log U)$. In unbounded universes, Derryberry et al. [5] describe a data structure that finds an approximate nearest neighbour p of the point q in some constant dimension in time $O(\log\delta(p,q))$, where $\delta(p,q)$ is the number of points in a certain box containing p and q.

*School of Computer Science, Carleton University. {jit,karim,vida,jhowat,morin}@cg.scs.carleton.ca. This research was partially supported by NSERC and MRI.

[1] In this paper, we define $\log x = \log_2(x+2)$.

2 One Dimension

In this section, we describe the one-dimensional version of the data structure. We begin with a review of x- and y-fast tries.

2.1 A Review of x- and y-fast Tries

x- and y-fast tries were presented by Willard [12] as a space-efficient alternative to van Emde Boas trees [11]. An x-fast trie on $\mathcal{U}$ is a binary tree whose leaves are elements of $\mathcal{U}$ and whose internal nodes represent prefixes of these leaves. The height of the tree is $\Theta(\log U)$. At any internal node, moving to the left child appends a 0 to the prefix and moving to the right child appends a 1. The prefix at the root node is empty. Therefore, a node at depth i represents a block of $2^{\log_2 U - i}$ elements of U having the same i highest order bits, and any root-to-leaf path in the trie yields the binary representation of the element at the leaf of that path. At every level of the tree (where the level of a node is its height), a hash table is maintained on all elements at that level. Each internal node with no left (respectively right) child is augmented with an additional pointer to the smallest (respectively largest) leaf in its subtree, and all leaves maintain pointers to the next leaf and the previous leaf. The structure uses $O(n \log U)$ space. Searches are executed by performing a binary search on the hash tables to find the deepest node whose prefix is a prefix of the query. If the query has binary representation $q_{\log_2 U} \cdots q_0$, at depth i we search for $q_{\log_2 U} \cdots q_{\log_2 U - i}$. If this is node happens to be a leaf, then the search is complete. Otherwise, this node must have exactly one child and thus has a pointer to the largest leaf in its subtree. Following this pointer will lead to the leaf storing either the successor or predecessor the query, and since the leaves form a doubly-linked list, the successor is then returned. The binary search takes $O(\log \log U)$ time and the subsequent operations take $O(1)$ time for a total of $O(\log \log U)$ time.

The y-fast trie overcomes the fact that the space used by an x-fast trie is a function of both n and U. Indirection is used to reduce this space to $O(n)$. The leaves are divided into groups of size $O(\log U)$ and each group is placed into a balanced binary search tree. Each binary search tree has a representative and these representatives are stored in the trie. A search in the trie for an element will eventually lead to a binary search tree which can then be searched in time $O(\log \log U)$. In order to facilitate insertions and deletions, the binary trees are rebuilt when their sizes are doubled or quartered. This allows insertions and deletions in $O(\log \log U)$ amortized time. The space required is now $O(n)$, since the x-fast trie stores all $O(n/\log U)$ representatives, each appearing in $O(\log U)$ hash tables.

2.2 The Static Case

In order to make local searches fast, the key observation is that instead of performing a binary search on the levels, we can instead perform a doubly exponential search beginning from the leaves. In this manner, if a query is stored inside the structure, it can be found in $O(1)$ time. If a query is far away from its successor, however, the exponential search can move higher up the structure. For now, we consider the static case.

Suppose we have an x-fast trie. During a search, we perform hash table queries for the appropriate prefix of the query at levels 2^{2^i} for $i = 0, 1, \ldots, O(\log \log \log U)$, starting at the leaves. If the prefix is found, the query can be answered by performing a search in the trie starting at that node. If the prefix is not found, search for the successor of that prefix (in the usual binary ordering) at the same level. If there is one, the query can be answered by following a pointer from that successor to the smallest leaf in its subtree. If the successor of the prefix was not found, then the exponential search is continued.

Lemma 1 *The exponential search algorithm for an x-fast trie described is correct and can be performed in worst-case $O(\log \log \Delta)$ time.*

Proof. If the query is contained in the trie, it must be stored in the hash table containing all leaves and thus can be found in $O(1) = O(\log \log \Delta)$ time. Otherwise, assume the query is not contained in the trie. We must show that its successor is found in $O(\log \log \Delta)$ time. Assume that a hash table query is first successful at level 2^{2^i}. Then both the prefix of the query and the successor of that prefix were not found at level $2^{2^{i-1}}$. If the prefix of the query and the successor *were* in the trie, then the subtree rooted there would have $O\left(2^{2^{2^{i-1}}}\right)$ elements. However, since they are not present, these elements are not present and thus at least $\Delta \geq 2^{2^{2^{i-1}}}$ elements of $\mathcal{U}$ are between the query and its successor. The exponential search up to level 2^{2^i} requires $O(i)$ time and searching the trie from that level will take $O\left(\log \log 2^{2^{2^{i-1}}}\right) = O\left(2^i\right)$ time. Since $\Delta \geq 2^{2^{2^{i-1}}}$ implies that i is $O(\log \log \log \Delta)$, the search is performed in $O\left(2^i\right) = O(\log \log \Delta)$ time. □

Note that although x-fast tries use $O(n \log U)$ space, we need only store the hash tables for the levels visited by the exponential search. The hash tables thus use $O(n \log \log \log U)$ space. We still need to traverse the rest of the trie in the second stage of the search, however. To reduce the space required, we simply store an additional y-fast trie with the same elements where each internal node of the x-fast trie maintains a pointer to the corresponding node in the y-fast trie. Instead of traversing the x-fast trie, we follow the pointer and traverse the

y-fast trie of size $O(n)$ in the same time bound. This modified trie thus uses only $O(n \log\log\log U)$ space.

Theorem 2 *Let $\mathcal{U} = \{0, 1, \ldots, U-1\}$. There exists a data structure that supports searches over a subset of $\mathcal{U}$ in worst-case $O(\log\log\Delta)$ time in $O(n\log\log\log U)$ space.*

By replacing "successor" with "predecessor," the same time bound can be achieved where Δ is defined as the rank difference between the element being searched for and its *predecessor* in the structure. If both structures are searched simultaneously until one returns an answer, it is possible to answer such queries in $O(\log\log\Delta)$ where Δ is the minimum of the rank differences between the query and its successor and predecessor in the structure.

2.3 The Dynamic Case

We now turn our attention to the dynamic version of the problem, where we would like to support insertion and deletion of elements of $\mathcal{U}$ into the trie. Our goal is to support these operations in $O(\log\log\Delta)$ time as well. One straightforward solution is to modify the data structure described in Theorem 2. To insert an element, search for it and add the appropriate prefixes to the hash tables. To delete an element, remove it from all hash tables. These operations can be performed in $O(\log\log\Delta + \log\log\log U)$ time. Our goal is to remove the $O(\log\log\log U)$ term from the update time. The primary reason this is difficult is that each hash table must maintain the appropriate prefixes of the elements in the subtrees rooted at that level. In order to overcome this, we will use an indirection trick similar to that used by y-fast tries, but instead maintain a single skip list [10].

This skip list will store all of the elements that are currently in the trie. Pointers are maintained between the elements of the trie and their corresponding elements in the skip list. Each element of the hash tables will further maintain a y-fast trie whose universe is the maximal subset of $\mathcal{U}$ that could be stored as leaves of that node. Therefore, a y-fast trie pointed to by an element of a hash table at height 2^{2^i} is over a universe of size $2^{2^{2^i}}$. Recall that a y-fast trie on a universe of size U' stores the bottom portion of the trie as a collection of balanced binary search trees each of size $O(\log U')$, which we will refer to as *buckets*. In our setting, this means that at height 2^{2^i}, the y-fast tries have buckets of size 2^{2^i}. We will not maintain these buckets explicitly, however. Instead, any pointer in the top portion of a y-fast trie that points into a bucket will instead point to a *representative* of that bucket that is stored in the skip list.

Representatives for buckets will be selected based on levels in the skip list. For reasons that will become clear during the analysis, we select the probability of promotion for the skip list to be $1/3$. Consider a y-fast trie pointed to by a hash table element at level 2^{2^i}. The representatives of the buckets for that y-fast trie will be the elements having that prefix who have height at least 2^i in the skip list. The expected size of a bucket is thus the expected number of elements between two elements of height 2^i. Since the probability of promotion is $1/3$, the expected size of a bucket is $1/(1/3)^{2^i} = 3^{2^i}$.

Lemma 3 *The expected space required for the data structure described is $O(n\log\log\log U)$.*

To execute a search, we proceed as before and perform hash table queries for the appropriate prefix at levels 2^{2^i} for $i = 0, 1, \ldots, O(\log\log\log U)$. If the prefix is not found in a level, search for the successor of that prefix (in the usual binary ordering) at the same level. If such a successor is found, the query can be answered by following a pointer to the minimum representative in the associated y-fast trie of the successor and then performing a finger search in the skip list using this representative. If the successor of the prefix was not found, then the exponential search is continued. If the prefix is found at a particular level, the search is continued in the associated y-fast trie to find the representative of the successor. At this point, a pointer can be followed into the skip list tree where a finger search can be performed to find the actual successor.

Lemma 4 *The search algorithm described takes $O(\log\log\Delta)$ expected time.*

Proof. If the query is contained in the trie, it must be stored in the hash table containing all leaves and thus can be found in $O(1) = O(\log\log\Delta)$ time. Otherwise, assume the query is not contained in the trie and a hash table query is first successful at level 2^{2^i}; then $\Delta \geq 2^{2^{2^{i-1}}}$, so i is $O(\log\log\log\Delta)$.

If the successful hash table query was for the prefix, then the query is answered by querying the associated y-fast trie, which can be done in $O\big(2^i\big)$ time. This yields the correct representative, and so a finger search is then performed in the skip list. Since the representative and the successor are in the same bucket, the expected distance between them is at most 3^{2^i}. The search in the skip list thus takes $O\Big(\log 3^{2^i}\Big) = O\big(2^i\big) = O(\log\log\Delta)$ expected time. If the successful hash table query was for the successor of the prefix, then the query is answered by following a pointer from the successor to the minimum representative in the associated y-fast trie. The same argument as the previous case yields the same time bound. □

Insertion can be accomplished by first searching for the element to be inserted in order to produce a pointer

to that element's successor. This pointer is then used to insert the element into the skip list. We then insert the appropriate prefixes of the element into the hash tables and the corresponding y-fast tries. The y-fast tries may need to be restructured as the buckets increase in size. The key observation is that we need only alter a hash table of a y-fast trie if its representative changes.

Lemma 5 *The insertion algorithm described takes $O(\log\log\Delta)$ expected amortized time.*

Proof. During an insertion, we insert the element into the skip list given a pointer to its successor. The probability that the new element reaches level 2^i (and thus becomes a representative) is $(1/3)^{2^i}$. If this happens, the prefixes of the new element must be added to every hash table of the corresponding y-fast trie. Since the corresponding y-fast trie has height 2^{2^i} and each hash table insertion takes $O(1)$ amortized time, the time to modify the y-fast trie is $O\left(2^{2^i}\right)$. This could happen for every level 2^{2^i}, however, since the y-fast tries are nested. The total expected amortized time to restructure the y-fast tries is thus

$$\sum_{i=0}^{\log\log\log U}\left(\frac{1}{3}\right)^{2^i}2^{2^i}\leq\sum_{i=0}^{\infty}\left(\frac{2}{3}\right)^{2^i}\leq\sum_{i=0}^{\infty}\left(\frac{2}{3}\right)^{i}=O(1)$$

Since the initial search takes expected time $O(\log\log\Delta)$ by Lemma 4, the insertion takes $O(\log\log\Delta)$ expected amortized time in total. □

Deletion works similarly. The element is searched for in order to produce a pointer to it. The pointer is then used to delete the element from the skip list. Finally, the appropriate prefixes of the element must be deleted from the hash tables (if they are not a prefix of another element in the structure) and the element must be removed from the corresponding y-fast tries. The analysis used for insertion applies to deletion as well.

Lemma 6 *The deletion algorithm described takes $O(\log\log\Delta)$ expected amortized time.*

Combining Lemma 4, Lemma 5 and Lemma 6 yields

Theorem 7 *Let $\mathcal{U}=\{0,1,\ldots,U-1\}$. There exists a data structure that supports searches over a subset of $\mathcal{U}$ in $O(\log\log\Delta)$ expected time and insertions and deletions of elements of $\mathcal{U}$ in $O(\log\log\Delta)$ expected amortized time, where Δ is the distance between the query and its successor. The expected space required for this structure is $O(n\log\log\log U)$.*

3 Two or More Dimensions

For some constant $d\geq 2$ dimensions, we consider the universe $\mathcal{U}^d$ and wish to return a $(1+\epsilon)$-approximation of the nearest neighbour of the query point (or the query point, if it happens to be in the structure) in time $O(\log\log\Delta)$, where Δ is the Euclidean distance to the point returned.

Chan [4] points out that by placing $d+1$ shifted versions of the stored points onto a space-filling curve[2], queries can be answered by answering the query on each of these curves (lists) and taking the closest point to be the desired approximation. Chan [4] notes that this yields a query time of $O\big((1/\epsilon^d)\log n\big)$ and $O(\log n)$ update time for sets of size n.

Observe that searching within the shifted lists is a one-dimensional problem and thus solvable by Theorem 7. This almost completely solves the problem, except that we cannot bound the search time in *every* one-dimensional structure as a function of Δ, rather only the structure which finds the element that is ultimately returned. To fix this, we randomly shift the entire point set using the technique described by Har-Peled [7, Chapter 11] before building any of the data structures; this results in the *expected* time spent searching in any one-dimensional structure to be $O(\log\log\Delta)$ for a total of $O\big((1/\epsilon^d)\log\log\Delta\big)$ expected time. Insertions and deletions are performed by performing the insertion or deletion in each one-dimensional structure.

Theorem 8 *Let $\mathcal{U}=\{0,1,\ldots,U-1\}$ and d be a constant. There exists a data structure that supports $(1+\epsilon)$-approximate nearest neighbour searches over a subset of $\mathcal{U}^d$ in $O\big((1/\epsilon^d)\log\log\Delta\big)$ expected time and insertions and deletions of elements of $\mathcal{U}^d$ in $O(\log\log\Delta)$ expected amortized time. The expected space required for this structure is $O(n\log\log\log U)$.*

4 Conclusion and Future Work

We have introduced the idea of fast local searches in bounded universes. In one dimension, we showed how to perform successor searches in $O(\log\log\Delta)$ expected time and updates in $O(\log\log\Delta)$ expected amortized time and $O(n\log\log\log U)$ expected space. For some constant $d\geq 2$ dimensions, searches for $(1+\epsilon)$-approximate nearest neighbours can be performed in $O\big((1/\epsilon^d)\log\log\Delta\big)$ expected time and insertions and deletions in $O(\log\log\Delta)$ expected amortized time. The expected space required for this structure is $O(n\log\log\log U)$.

[2] Chan [4] uses the z-order, which is obtained by computing the *shuffle* of a point. If the i-th coordinate of a point p is p_i with binary representation $p_{i,\log U}\cdots p_{i,0}$, then the shuffle of p is defined to be the binary number $p_{1,\log U}\cdots p_{d,\log U}\cdots p_{1,0}\cdots p_{d,0}$.

Future Work. The following are possible directions for future research. It would be interesting to see if it is possible to keep the stated query times (or something close to them) using space that is linear in n. We also rely on randomization and amortization for everything except the static version of the one-dimensional problem; can the other versions be solved without randomization or amortization? One might consider replacing the skip list with the optimal finger search structure of Brodal et al. [3], but this seems to require handling restructuring the y-fast tries in a more complicated way.

References

[1] Arnon Amir, Alon Efrat, Piotr Indyk, and Hanan Samet. Efficient regular data structures and algorithms for dilation, location, and proximity problems. *Algorithmica*, 30(2):164–187, 2001.

[2] Gerth Stølting Brodal, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, and Kostas Tsichlas. Optimal solutions for the temporal precedence problem. *Algorithmica*, 33(4):494–510, 2002.

[3] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003.

[4] Timothy M. Chan. Closest-point problems simplified on the RAM. In *SODA '02: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 472–473, 2002.

[5] John Derryberry, Daniel D. Sleator, Donald Sheehy, and Maverick Woo. Achieving spatial adaptivity while finding approximate nearest neighbors. In *CCCG '08: Proceedings of the 20th Canadian Conference on Computational Geometry*, pages 163–166, 2008.

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

[7] Sariel Har-Peled. Geometric approximation algorithms. Online at `http://valis.cs.uiuc.edu/~sariel/teach/notes/aprx/`, 2009.

[8] Donald B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Theory of Computing Systems*, 15(1):295–309, 1981.

[9] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. Improved bounds for finger search on a RAM. In *ESA '03: Proceedings of the 11th Annual Eureopean Symposium on Algorithms, LNCS 2832*, pages 325–336, 2003.

[10] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[11] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[12] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.