A Distribution-Sensitive Dictionary with Low Space Overhead*

Prosenjit Bose, John Howat, and Pat Morin

School of Computer Science, Carleton University
1125 Colonel By Dr., Ottawa, Ontario, CANADA, K1S 5B6
{jit,jhowat,morin}@scs.carleton.ca

Abstract. The time required for a sequence of operations on a data structure is usually measured in terms of the worst possible such sequence. This, however, is often an overestimate of the actual time required. *Distribution-sensitive* data structures attempt to take advantage of underlying patterns in a sequence of operations in order to reduce time complexity, since access patterns are non-random in many applications. Unfortunately, many of the distribution-sensitive structures in the literature require a great deal of space overhead in the form of pointers. We present a dictionary data structure that makes use of both randomization and existing space-efficient data structures to yield very low space overhead while maintaining distribution sensitivity in the expected sense.

1 Introduction

For the *dictionary problem*, we would like to efficiently support the operations of Insert, Delete and Search over some totally ordered universe. There exist many such data structures: AVL trees [1], red-black trees [5] and splay trees [9], for instance. Splay trees are of particular interest because they are *distribution-sensitive*, that is, the time required for certain operations can be measured in terms of the distribution of those operations. In particular, splay trees have the *working set property*, which means that the time required to search for an element is logarithmic in the number of distinct accesses since that element was last searched for. Splay trees are not the only dictionary to provide the working set property; the working set structure [6] and the unified structure [6] also have it.

Unfortunately, such dictionaries often require a significant amount of space overhead. Indeed, this is a problem with data structures in general. Space overhead often takes the form of pointers: a binary search tree, for instance, might have two pointers per node in the tree: one to each child. If this is the case and we assume that pointers and keys have the same size, then it is easy to see that 2/3 of the storage used by the binary search tree consists of pointers. This seems to be wasteful, since we are really interested in the data itself and would rather not invest such a large fraction of space in overhead. To remedy this situation,

* This research was partially supported by NSERC and MRI.

there has been a great deal of research in the area of *implicit* data structures. An implicit data structure uses only the space required to hold the data itself (in addition to only a constant number of words, each of size $O(\log n)$ bits). Implicit dictionaries are a particularly well-studied problem [8, 2, 4].

Our goal is to combine notions of distribution sensitivity with ideas from implicit dictionaries to yield a distribution-sensitive dictionary with low space overhead.

1.1 Our Results

We will present a dictionary data structure with worst-case insertion and deletion times $O(\log n)$ and expected search time $O(\log t(x))$, where x is the key being searched for and $t(x)$ is the number of distinct queries made since x was last searched for, or n if x has not yet been searched for. The space overhead required for this data structure is $O(\log \log n)$, *i.e.*, $O(\log \log n)$ additional words of memory (each of size $O(\log n)$ bits) are required aside from the data itself. Current data structures that can match this query time include the splay tree [9] (in the amortized sense) and the working set structure [6] (in the worst case), although these require $3n$ and $5n$ pointers respectively, assuming three pointers per node (one parent pointer and two child pointers). We also show how to modify this structure (and by extension the working set structure [6]) to support predecessor queries in time logarithmic in the working set number of the predecessor.

The rest of the paper is organized in the following way. Section 2 briefly summarizes the working set structure [6] and shows how to modify it to reduce its space overhead. Section 3 shows how to modify the new dictionary to support more useful queries with additional–but sublinear–overhead. These modifications are also applicable to the working set structure [6] and make both data structures considerably more useful. Section 4 concludes with possible directions for future research.

2 Modifying the Working Set Structure

In this section, we describe the data structure. We will begin by briefly summarizing the working set structure [6]. We then show how to use randomization to remove the queues from the working set structure, and finally how to shrink the size of the trees in the working set structure.

2.1 The Working Set Structure

The working set structure [6] consists of k balanced binary search trees $T_1, \ldots, T_k$ and k queues $Q_1, \ldots, Q_k$. Each queue has precisely the same elements as its corresponding tree, and the size of T_i and Q_i is 2^{2^i}, except for T_k and Q_k which simply contain the remaining elements. Therefore, since there are n elements, $k = O(\log \log n)$. The structure is manipulated with a *shift* operation in the

following manner. A shift from i to j is performed by dequeuing an element from Q_i and removing the corresponding element from T_i. The removed element is then inserted into the next tree and queue (where "next" refers to the tree closer to j), and the process is repeated until we reach T_j and Q_j. In this manner, the oldest elements are removed from the trees every time. The result of a shift is that the size of T_i and Q_i has decreased by one and the size of T_j and Q_j has increased by one.

Insertions are made by performing a usual dictionary insertion into T_1 and Q_1, and then shifting from the first index to the last index. Such a shift makes room for the newly inserted element in the first tree and queue by moving the oldest element in each tree and queue down one index. Deletions are accomplished by searching for the element and deleting it from the tree (and queue) it was found in, and then shifting from the last index to the index the element was found at. Such a shift fills in the gap created by the removed element by bringing elements up from further down the data structure. Finally, a search is performed by searching successively in $T_1, T_2, \ldots, T_k$ until the element is found. This element is then removed from the corresponding tree and queue and inserted into T_1 and Q_1 in the manner described previously. By performing this shift, we ensure that elements searched for recently are towards the front of the data structure and will therefore by found quickly on subsequent searches.

The working set structure was shown by Iacono [6] to have insertion and deletion costs of $O(\log n)$ and a search cost of $O(\log t(x))$, where x is the key being searched for and $t(x)$ is the number of distinct queries made since x was last searched for, or n if x has not yet been searched for. To see that the search cost is $O(\log t(x))$, consider that if x is found in T_i, it must have been dequeued from Q_{i-1} at some point. If this is the case, then $2^{2^{i-1}}$ accesses to elements other than x have taken place since the last access to x, and therefore $t(x) \geq 2^{2^{i-1}}$. Since the search time for x is dominated by the search time in the tree it was found in, the cost is $O\left(\log 2^{2^i}\right) = O(\log t(x))$.

2.2 Removing the Queues

Here we present a simple use of randomization to remove the queues from the working set structure. Rather than relying on the queue to inform the shifting procedure of the oldest element in the tree, we simply pick a random element in the tree and treat it exactly as we would the dequeued element. Lemma 1 shows that we still maintain the working set property in the expected sense.

Lemma 1. *The expected search cost in the randomized working set structure is* $O(\log t(x))$.

Proof. Fix an element x and let $t = t(x)$ denote the number of distinct accesses since x was last accessed. Suppose that x is in T_i and a sequence of accesses occurs during which t distinct accesses occur. Since T_i has size 2^{2^i}, the probability that x is not removed from T_i during these accesses is at least

$$\begin{aligned}\Pr\{x \text{ not removed from } T_i \text{ after } t \text{ accesses}\} &\geq \left(1 - \frac{1}{2^{2^i}}\right)^t \\ &= \left(1 - \frac{1}{2^{2^i}}\right)^{\frac{t2^{2^i}}{2^{2^i}}} \\ &\geq \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i}}}\end{aligned}$$

Now, if $x \in T_i$, it must have been selected for removal from T_{i-1} at some point. The probability that x is in at least the i-th tree is therefore at most

$$\Pr\{x \in T_j \text{ for } j \geq i\} \leq 1 - \frac{1}{4}^{\frac{t}{2^{2^{i-1}}}}$$

An upper bound on the expectation $E[S]$ of the search cost S is therefore

$$\begin{aligned}E[S] &= \sum_{i=1}^{k} O(\log |T_i|) \times \Pr\{x \in T_i\} \\ &\leq \sum_{i=1}^{k} O(\log |T_i|) \times \Pr\{x \in T_j \text{ for } j \geq i\} \\ &\leq \sum_{i=1}^{k} O\Big(\log\left(2^{2^i}\right)\Big)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^{i-1}}}}\right) \\ &= \sum_{i=1}^{k} O\big(2^i\big)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^{i-1}}}}\right) \\ &= \sum_{i=1}^{\lfloor \log\log t \rfloor} O\big(2^i\big)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^{i-1}}}}\right) + \sum_{i=1+\lfloor \log\log t \rfloor}^{k} O\big(2^i\big)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^{i-1}}}}\right) \\ &= O(\log t) + \sum_{i=1}^{k-\lfloor \log\log t \rfloor} O\Big(2^{i+\lfloor \log\log t \rfloor}\Big)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2(i+\lfloor \log\log t \rfloor)-1}}}\right) \\ &= O(\log t) + O(\log t) \sum_{i=1}^{k-\lfloor \log\log t \rfloor} O\big(2^i\big)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2(i+\lfloor \log\log t \rfloor)-1}}}\right) \\ &\leq O(\log t) + O(\log t) \sum_{i=1}^{k-\lfloor \log\log t \rfloor} O\big(2^i\big)\left(1 - \left(\frac{1}{4}\right)^{\frac{t}{2^{2^i \log t}}}\right) \\ &= O(\log t) + O(\log t) \sum_{i=1}^{k-\lfloor \log\log t \rfloor} O\big(2^i\big)\left(1 - \left(\frac{1}{4}\right)^{\frac{1}{t^{2^i-1}}}\right)\end{aligned}$$

It thus suffices to show that the remaining sum is $O(1)$. We will assume that $t \geq 2$, since otherwise x can be in at most the second tree and can therefore be found in $O(1)$ time. Considering only the remaining sum, we have

$$\begin{aligned}\sum_{i=1}^{k-\lfloor \log\log t \rfloor} O(2^i)\left(1-\left(\frac{1}{4}\right)^{\frac{1}{t^{2^i-1}}}\right) &\leq \sum_{i=1}^{k-\lfloor \log\log t \rfloor} O(2^i)\left(1-\left(\frac{1}{4}\right)^{\frac{1}{2^{2^i-1}}}\right)\\ &\leq \sum_{i=1}^{k-\lfloor \log\log t \rfloor} O(2^i)\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)\\ &\leq \sum_{i=1}^{\infty} O(2^i)\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)\end{aligned}$$

All that remains to show is that this infinite sum is bounded by a decreasing geometric series (and is therefore constant.) The ratio of consecutive terms is

$$\lim_{i\to\infty} \frac{2^{i+1}\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^{i+1}}}}\right)}{2^i\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)} = 2\lim_{i\to\infty} \frac{\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^{i+1}}}}\right)}{\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)}$$

If we substitute $u = \frac{1}{2^{2^i}}$, we find that $\frac{1}{2^{2^{i+1}}} = u^2$. Observe that as $i \to \infty$, we have $u \to 0$. Thus

$$2\lim_{i\to\infty} \frac{\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^{i+1}}}}\right)}{\left(1-\left(\frac{1}{16}\right)^{\frac{1}{2^{2^i}}}\right)} = 2\lim_{u\to 0} \frac{\left(1-\left(\frac{1}{16}\right)^{u^2}\right)}{\left(1-\left(\frac{1}{16}\right)^{u}\right)} = 2\lim_{u\to 0} \frac{8\left(\frac{1}{16}\right)^{u^2} u\ln 2}{4\left(\frac{1}{16}\right)^{u}\ln 2} = 0$$

Therefore, the ratio of consecutive terms is $o(1)$ and the series is therefore bounded by a decreasing geometric series. An expected search time of $O(\log t(x))$ follows.

At this point, we have seen how to eliminate the queues from the structure at a cost of an *expected* search cost. In the next section, we will show how to further reduce space overhead by shrinking the size of the trees.

2.3 Shrinking the Trees

Another source of space overhead in the working set structure is that of the trees. As mentioned before, many pointers are required to support a binary search tree. Instead, we will borrow some ideas from the study of implicit data structures. Observe that there is nothing special about the trees used in the working set structure: they are simply dictionary data structures that support logarithmic time queries and update operations. In particular, we do not rely on the fact that they are trees. Therefore, we can replace these trees with one of the many implicit dictionary data structures in the literature (see, *e.g.*, [8, 2, 4, 3].) The

dictionary of Franceschini and Grossi [3] provides a worst-case optimal implicit dictionary with access costs $O(\log n)$, and so we will employ these results. It is useful to note that any dictionary that offers polylogarithmic access times will yield the same results: the access cost for each operation in our data structure will be the maximum of the access costs for the substructure, since the shifting operation consists of searches, insertions and deletions in the substructures.

Unfortunately, the resulting data structure is not implicit in the strict sense. Since each substructure can use $O(1)$ words of size $O(\log n)$ bits and we have $O(\log \log n)$ such substructures, the data structure as a whole could use as much as $O(\log \log n)$ words of size $O(\log n)$ bits each. Nevertheless, this is a significant improvement over the $O(n)$ additional words used by the traditional data structures. We have

Theorem 1. *There exists a dictionary data structure that stores only the data required for its elements in addition to $O(\log \log n)$ words of size $O(\log n)$ bits each. This dictionary supports insertions and deletions in worst-case $O(\log n)$ time and searches in expected $O(\log t(x))$ time, where $t(x)$ is the working set number of the query x.*

3 Further Modifications

In this section, we describe a simple modification to the data structure outlined in Section 2 that makes searches more useful. This improvement comes at the cost of additional space overhead.

Until now, we have implicitly assumed that searches in our data structure are successful. If they are not, then we will end up searching in each substructure at a total cost of $O(\log n)$ and returning nothing. Unfortunately, this is not very useful.[1] Typically, a dictionary will return the largest element in the dictionary that is smaller than the element searched for or the smallest element larger than the element searched for. Such predecessor and successor queries are a very important feature of comparison-based data structures: without them, one could simply use hashing to achieve $O(1)$ time operations. Predecessor queries are simple to implement in binary search trees since we can simply examine where we "fell off" the tree. This trick will not work in our data structure, however, since we have many such substructures and we will have to know when to stop.

Our goal is thus the following. Given a search key x, we would (as before) like to return x in time $O(\log t(x))$ if x is in the data structure. If x is not in the data structure, we would like to like to return $pred(x)$ in time $O(\log t(pred(x)))$, where $pred(x)$ denotes the predecessor of x.

To accomplish this, we will augment our data structure with some pointers. In particular, every item in the data structure will have a pointer to its successor. During an insertion, each substructure will be searched for the inserted element

[1] Note that this is also true of the original working set structure [6]. The modifications described here are also applicable to it.

for a total cost of $O(\log n)$ and the smallest successor in each substructure will be recorded. The smallest such successor is clearly the new element's successor in the whole structure. Therefore, the cost of insertion remains $O(\log n)$. Similarly, during a deletion only the predecessor of the deleted element will need to have its successor pointer updated and thus the total cost of deletion remains $O(\log n)$.

During a search for x, we proceed as before. Consider searching in any particular substructure i. If the result of the search in substructure i is in fact x, then the analysis is exactly the same as before and we can return x in time $O(\log t(x))$. Otherwise, we won't find x in substructure i. In this case, we search substructure i for the predecessor (in that substructure) of x.[2] Denote this $pred_i(x)$. Since every element knows its successor in the structure as a whole, we can determine $succ(pred_i(x))$. If $succ(pred_i(x)) = x$, then we know that x is indeed in the structure and thus the query can be completed as before. If $succ(pred_i(x)) < x$, then we know that there is still an element smaller than x but larger than what we have seen, and so we continue searching for x. Finally, if $succ(pred_i(x)) > x$, then we have reached the largest element less than or equal to x, and so our search stops.

In any case, after we find x or $pred(x)$, we shift the element we returned to the first substructure as usual. The analysis of the time complexity of this search algorithm is exactly the same as before; we are essentially changing the element we are searching for during the search. The search time for the substructure we stop in dominates the cost of the search and since we return x if we found it or $pred(x)$ otherwise, the search time is $O(\log t(x))$ if x is in the dictionary and $O(\log t(pred(x)))$ otherwise.

Of course, this augmentation incurs some additional space overhead. In particular, we now require n pointers, resulting in a space overhead of $O(n)$. While the queues are now gone, we still have one pointer for each element in the dictionary. To fix this, observe that we can leave out the pointers for the last few trees and simply do a brute-force search at the cost of slightly higher time complexity. Suppose we leave the pointers out of the last j trees: T_{k-j+1} to T_k, where k represents the index of the last tree, as before. Therefore, each element x in the trees $T_1, \ldots, T_{k-j}$ has a pointer to $succ(x)$. Now, suppose we are searching in the data structure and get to T_{k-j+1}. At this point, we may need to search all remaining trees, since if we do not find the key we are looking for, we have no way of knowing when we have found its predecessor. Consider the following lemma.

Lemma 2. *Let $0 \leq j \leq k$. A predecessor search for x takes expected time $O\big(2^j \log t(x)\big)$ if x is in the dictionary and $O\big(2^j \log t(pred(x))\big)$ otherwise.*

Proof. Assume the search reaches T_{k-j+1}, since otherwise our previous analyses apply. We therefore have $t(x) \geq 2^{2^{k-j}}$, since at least $2^{2^{k-j}}$ operations have taken

[2] Here we are assuming that substructures support predecessor queries in the same time required for searching. This is not a strong assumption, since any comparison-based dictionary must compare x to $succ(x)$ and $pred(x)$ during an unsuccessful search for x.

place.[3] As before, the search time is bounded by the search time in T_k. Since T_k has size at most 2^{2^k}, we have an expected search time of

$$O\Big(\log 2^{2^k}\Big) = O\Big(\log 2^{2^{k-j}2^j}\Big) = O\Big(2^j \log 2^{2^{k-j}}\Big) \leq O\big(2^j \log t(x)\big)$$

This analysis applies as well when x is not in the dictionary. In this case, the search can be accomplished in time $O\big(2^j \log t(pred(x))\big)$.

One further consideration is that once an element is found in the last j trees, we need to find its successor so that it knows where it is once it is shifted to the front. However, this is straightforward because we have already examined all substructures in the data structure and so we can make a second pass. It remains to consider how much space we have saved using this scheme. Since each tree has size the square of the previous, by leaving out the last j trees, the total number of extra pointers used is $O\Big(n^{1/2^j}\Big)$. We therefore have

Theorem 2. *Let $0 \leq j \leq k$. There exists a dictionary that stores only the data required for its elements in addition to $O\Big(n^{1/2^j}\Big)$ words of size $O(\log n)$ bits each. This dictionary supports insertions and deletions in worst-case $O(\log n)$ time and searches in expected $O\big(2^j \log t(x)\big)$ time if x is found in the dictionary and expected $O\big(2^j \log t(pred(x))\big)$ time otherwise (in which case $pred(x)$ is returned).*

Observe that $j \leq k = O(\log \log n)$, and so while the dependence on j is exponential, it is still quite small relative to n. In particular, take $j = 1$ to get

Corollary 1. *There exists a dictionary that stores only the data required for its elements in addition to $O(\sqrt{n})$ words of size $O(\log n)$ bits each. This dictionary supports insertions and deletions in worst-case $O(\log n)$ time and searches in expected $O(\log t(x))$ time if x is found in the dictionary. If x is not in the dictionary, $pred(x)$ is returned in expected $O(\log t(pred(x)))$ time.*

4 Conclusion

We have seen how to modify the Iacono's working set structure [6] in several ways. To become more space efficient, we can remove the queues and use randomization to shift elements, while replacing the underlying binary search trees with implicit dictionaries. To support more useful search queries, we can sacrifice some space overhead to maintain information about some portion of the elements in the dictionary in order to support returning the predecessor of any otherwise unsuccessful search queries. All such modifications maintain the working set property in an expected sense.

[3] This follows from Lemma 1, which is why the results here hold in the expected sense.

4.1 Future Work

The modifications described in this paper leave open a few directions for research.

The idea of relying on the properties of the substructures (in this case, implicitness) proved fruitful. A natural question to ask, then, is what other substructure properties can carry over to the dictionary as a whole in a useful way? Other substructures could result in a combination of the working set property and some other useful properties.

In this paper, we concerned ourselves with the working set property. There are other types of distribution sensitivity, such as the *dynamic finger property*, which means that query time is logarithmic in the rank difference between successive queries. One could also consider a notion complementary to the idea of the working set property, namely the *queueish property* [7], wherein query time is logarithmic in the number of items *not* accessed since the query item was last accessed. Are there implicit dictionaries that provide either of these properties? Could we provide any of these properties (or some analogue of them) for other types of data structures?

Finally, it would be of interest to see if a data structure that does not rely on randomization is possible, in order to guarantee a *worst case* time complexity of $O(\log t(x))$ instead of an expected one.

References

1. G.M. Adelson-Velskii and E.M. Landis. An algorithm for the organization of information. *Soviet Math. Doklady*, 3:1259–1263, 1962.
2. Gianni Franceschini and Roberto Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log\log n)$ time. In *SODA '03: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 670–678, 2003.
3. Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *WADS '03: Proceedings of the 30th International Workshop on Algorithms and Data Structures*, pages 114–126, 2003.
4. Gianni Franceschini and J. Ian Munro. Implicit dictionaries with $O(1)$ modifications per update and fast search. In *SODA '06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 404–413, 2006.
5. Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS '78: Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.
6. John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *SODA '01: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–522, 2001.
7. John Iacono and Stefan Langerman. Queaps. *Algorithmica*, 42(1):49–56, 2005.
8. J Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O\left(\log^2 n\right)$ time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986.
9. Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.