PUTTING YOUR DICTIONARY ON A DIET

Pat Morin
School of Computer Science, Carleton University
morin@cs.carleton.ca

ABSTRACT. We show that any comparison-based dictionary data structure that requires cn memory in addition to the storage required for the data elements can be transformed into a dictionary that requires only ϵn additional memory, for any $\epsilon > 0$. This transformation does not increase the running times of algorithms for searching, inserting and deleting, except by a small constant factor that is independent of ϵ and an additive term of $O(1/\epsilon)$ or $O(1/\epsilon^2)$ depending on the implementation used.

Keywords: Data structures, algorithms, space efficiency

1 Introduction

A *dictionary* is an abstract data type that stores a set of elements from some total order $(X, <)$. A dictionary supports insertions and deletions of elements in X and searches of the form: Given an element $x \in X$, report the smallest value y currently stored in the dictionary that is larger than X, or **nil** if no such value exists.

Dictionaries are a fundamental data type and many data structures that implement dictionary operations have been proposed, each having its own advantages and disadvantages. Issues include complexity of implementation, memory overhead, constants involved in running times, worst-case versus expected versus amortized performance guarantees, and so on. Many dictionary implementations are also special because they have certain *properties* such as the working-set property [8, 12], the queueish property [9], the unified property [8, 12], the ability to insert and delete in constant time [7], and the (static or dynamic) finger property [1, 3, 4, 6, 7].

These special properties are a large part of the reason that so many dictionary implementations exist. Many algorithmic applications require dictionaries that have some of these properties to ensure the bounds on their running times or storage requirements. Thus, it is not always possible to substitute one dictionary implementation for another.

One particular aspect of dictionary implementations that has received much attention is the issue of memory overhead. *Memory overhead* is any storage used by the data structure beyond what is actually required to store the data elements. Several implementations of dictionaries, including splay trees [12], skiplists [10], scapegoat trees [5], and variants of treaps [1] have been proposed that reduce the extra storage to 2 pointers per data item. In fact, with an increase in search cost, skiplists can reduce the overhead to $1 + \epsilon$ pointers per item.

Much research effort has gone into these attempts to reduce the memory overhead of dictionary data structures. For example, in a recent paper, Blelloch *et al* [2] describe a fairly intricate data structure of size $O(\log n)$ called a *hand* that sits on top of a degree-balanced search tree (e.g., an (a, b)-tree [7]) so that it supports efficient finger searches. However, it has long been known that degree-balanced search trees easily support efficient finger searches if we augment each node with three pointers [3]. Thus, the main contribution of the hand is to eliminate these three pointers per node.

In this paper, we consider the problem of implementing dictionaries that use only ϵn additional storage, for any $\epsilon > 0$. One way to achieve this is with a $(b, 2b)$-tree [7]. If implemented carefully, such a tree uses only $O(1/b)$ pointers per item. However, this solution is somewhat unsatisfactory since, although a $(b, 2b)$-tree can be augmented with a hand so that it supports finger searches, it nevertheless lacks some of the other properties mentioned above.

In this paper we point out a trick that can be used to make *any* pointer-based dictionary data structure that uses cn additional storage, for any constant c, into an equivalent data structure that uses $\epsilon n + O(1)$ additional storage, for any $\epsilon > 0$. By applying this simple trick, it is possible to use any dictionary data structure (and its properties) that is required by the underlying application without worrying about the memory overhead.

The trick we use is quite simple, and is similar to the idea used in degree-balanced search trees [7]. Each node contains at least a elements and at most b elements for appropriately chosen a and b. However, unlike degree-balanced search trees, we are able to maintain this property under insertions and deletions without causing changes that cascade throughout the data structure. This trick does not increase the running times of algorithms for searching, inserting and deleting, except by a small constant factor that is independent of ϵ and an additive term of $O(1/\epsilon)$ or $O(1/\epsilon^2)$ depending on which of our two implementations are used.

The remainder of the paper is organized as follows. In Section 2, we introduce fat items and show how they reduce the memory overhead of any comparison-based dictionary data structure. In Section 3 we give an alternative implementation of fat items that increases the additive constant while decreasing the multiplicative constant. Finally, in Section 4 we summarize and conclude with some final remarks.

2 Slimming Down with Fat Items

For simplicity, in the following, we assume that our dictionary never contains fewer than b elements (b^2 in the subsequent section). This is not a restrictive assumption, since we can achieve any memory overhead $\epsilon > 0$ with a value of $b = O(1/\epsilon)$. Our approach is to group dictionary items together into fat items, and use any dictionary data structure (skiplist, 2-3 tree, red-black tree, etc) to store these fat items. To help reduce confusion, we will refer the data structure we are describing as "the dictionary" and the data structure we are using as "the data structure." Similarly, we will use the term fat items and elements to refer to groups of elements of X and individual elements of X, respectively.

A *fat item* f is a data structure that contains two pointers $\mathrm{pred}(f)$ and $\mathrm{succ}(f)$ that point to other fat items, a pointer to an array $\mathrm{elem}(f)$ that contains anywhere between b and $2b$ elements of X, and an integer $\mathrm{size}(f)$ that tells the number of elements in $\mathrm{elem}(f)$. The pred and succ fields are used to link all fat items together into a doubly-linked list. The elements of $\mathrm{elem}(f)$ are always sorted in increasing order, and the elements of $\mathrm{elem}(f)$ are all less than the elements of $\mathrm{elem}(g)$ if f appears before g in the doubly-linked list. In this way, the list of fat items gives the elements of the dictionary in sorted order.

Note that comparing two fat items is a constant time operation, since to test if $f < g$ we compare the last element of $\mathrm{elem}(f)$ to the first element of $\mathrm{elem}(g)$. Similarly, given an element x of our total order we can test if $x < f$, respectively $f < x$, by comparing x with the first, respectively last, element in $\mathrm{elem}(f)$. Therefore, to reduce the storage requirements of a dictionary, we make a data structure (skiplist, splay tree, binary search tree, etc.) on a set of fat items whose elem arrays contain all the elements in our dictionary.

Searching. If we search for some element x in the data structure, we find the fat item f such that $\mathrm{elem}(f)$ contains the smallest key greater than or equal to x. An additional search in $\mathrm{elem}(f)$ finds the actual element we are looking for at a cost of $O(b)$, or $O(\log b)$ if we use binary search. If the data structure takes $S(n)$ time to perform a search on a set of n keys, then the dictionary takes $S'(n) = O(S(n/b) + \log b)$ time to perform the search.

Insertion. Assume we are given a pointer to the dictionary node containing the fat item f we would find if we searched for x. To insert x, we proceed as follows. If $\mathrm{elem}(f)$ contains fewer than $2b$ elements then we simply reallocate $\mathrm{elem}(f)$ to increase its size by 1 and add x to $\mathrm{elem}(f)$. Otherwise ($\mathrm{elem}(f)$ contains $2b$ items), we split $\mathrm{elem}(f)$ into two fat items, one that contains b elements and one that contains $b + 1$ elements and insert the newly created fat item into our data structure. Therefore, if the data structure takes $I(n)$ time to perform an insertion on a set of n elements then insertion in the dictionary takes $I'(n) = O(I(n/b) + b)$ time.

Deletion. To delete an element x from the dictionary, we assume we are given a pointer to the dictionary node containing the fat item f such that $\mathrm{elem}(f)$ contains x. If $\mathrm{elem}(f)$ has size greater than b, then we simply reallocate $\mathrm{elem}(f)$ to decrease its size by 1 and exclude x. Otherwise ($\mathrm{elem}(f)$ has size b), we examine a fat item g that is a neighbour of f in the linked list. If $\mathrm{elem}(g)$ contains more than b elements then we take the first or last element in $\mathrm{elem}(g)$ (depending on whether $g = \mathrm{succ}(f)$ or $g = \mathrm{pred}(f)$) and use it to replace x in g. Otherwise ($\mathrm{elem}(f)$ and $\mathrm{elem}(g)$ both have size b), we merge f and g into a single fat node and delete one of them from the data structure. If $D(n)$ is the time it takes to delete an element x from a data structure of size n, then deletion in the modified data structure takes $D'(n) = O(D(n/b) + b)$ time.

Memory Overhead. A data structure containing m fat items contains at least $n = bm$ elements of X. Each fat item has a constant amount of overhead, and the data structure has an overhead of cm for some constant c, so the overhead is $O(n/b)$.

Theorem 1. *Given a comparison-based dictionary data structure that requires $S(n)$, $I(n)$, and $D(n)$ time to search, insert, and delete (respectively) an item from a set of n items and that has a memory overhead of cn for some constant c, we can construct a dictionary data structure that requires $S'(n) = O(S(\frac{n}{b}) + \log b)$, $I'(n) = O(I(\frac{n}{b}) + b)$, and $D'(n) = O(D(\frac{n}{b}) + b)$ time to search, insert, and delete (respectively) and that has a memory overhead of $O(\frac{n}{b})$.*

3 Less Memory Management and Less Searches

In the implementation described in the previous section, each insertion and deletion results in one or two small arrays being resized. In some systems, all this allocating and freeing of memory can become expensive (this is one of the main drawbacks of pointer-based data structures). To avoid this, we can use an alternative design in which the elem arrays of fat items are all of size $b + 1$ and contain somewhere between $b-1$ and $b+1$ items, so that at most two array spaces may be wasted in each fat item. It follows immediately that this scheme, like the previous one, creates a dictionary with a memory overhead of $O(n/b)$.

Insertion. To insert the item x into the fat item f, we consider a set of b consecutive fat items, one of which is f. If one of these fat items has an elem array containing fewer than $b + 1$ elements of X then we can insert x by redistributing the elements across the b fat items. Otherwise (all b fat items hold $b+1$ elements of X), we can create a new fat item and redistribute the $b^2 + b$ elements of X across the $b + 1$ fat items so that each fat item contains b elements. Once this is done, one of the fat items has room to insert x.

Deletion. Deletion of the item x from the fat item f is similar. If one of the b consecutive fat items contains more than $b-1$ elements then we can delete x by redistributing elements across these b fat items. Otherwise (all b fat items contain $b-1$ elements of X), we can delete a fat item and redistribute the b^2-b elements of X across the remaining $b-1$ fat items so that each fat item receives b elements of X. Once this is done, we can delete x from the fat item containing x.

Analysis. One aspect of the cost of insertion and deletion is the cost of redistributing the elements among the b fat nodes. This cost is clearly $O(b^2)$, which is a factor b worse than the cost of the method outlined in the previous section. However, as we will see below, this method does have its advantages.

To analyze the above scheme, we use the potential function method of amortized analysis [11]. We say that the potential of a fat item f is $\Phi(f) = \frac{1}{b}|b - \mathrm{size}(f)|$ and the potential Φ of the entire data structure is the sum of the potentials of all fat items in the data structure. Clearly this potential function is always non-negative. We will use the convention that one unit of potential is equivalent to K units of work, which is the cost of one memory allocation or deallocation and one insertion or deletion in the data structure.

We analyze the amortized cost of the insertion procedure. The analysis of the deletion procedure is exactly the same. The insertion procedure has two cases. In the first case (no new node is allocated), the redistribution of the b elements across b nodes is easily done so that the increase in potential is at most $1/b$ (one array goes from having b elements to having $b+1$ elements), so the amortized cost in this case is $O(K/b + b^2)$. In the second case, one memory allocation is done and one insertion in the data structure is done, so the actual cost is $K + O(b^2)$. However, in this case, the potential decreases by $(b-1)/b$, since $b-1$ fat items go from having $b+1$ elements in their elem arrays to having b elements in their elem arrays, so the amortized cost in this case is also $O(K/b + b^2)$.

Theorem 2. *Given a comparison-based dictionary data structure that requires $S(n)$, $I(n)$, and $D(n)$ time to search, insert, and delete (respectively) an item from a set of n items and that has a memory overhead of cn for some constant c, we can construct a dictionary data structure that requires $S'(n) = O(S(\frac{n}{b}) + \log b)$, $I'(n) = O(\frac{1}{b}I(\frac{n}{b}) + b^2)$, and $D'(n) = O(\frac{1}{b}D(\frac{n}{b}) + b^2)$ amortized time to search, insert, and delete (respectively) and that has a memory overhead of $O(\frac{n}{b})$.*

4 Summary and Remarks

We have shown that any comparison-based dictionary data structure of linear size can be put on a diet, so that its memory overhead is reduced to ϵn, for any $\epsilon > 0$. This diet does not change the running times of the operations on the dictionary except by a small constant factor independent of ϵ and an additive term of $O(1/\epsilon)$ or $O(1/\epsilon^2)$ depending on the scheme used.

The practical value of this is that users of dictionary data structures need not worry about the storage overhead associated with the dictionary data structure they choose. If it becomes a problem then they can simply apply this technique and the problem is solved. The theoretical value of this is that designers of dictionary data structures need no longer make unnecessarily complicated algorithms and representations for the sake of reducing the memory overhead. They can get on with the problem of designing dictionaries that have useful running time properties without worrying about memory overhead.

References

[1] C. R. Aragon and R. Seidel. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[2] G. E. Blelloch, B. Maggs, and M. Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the 14th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA 2003)*, 2003. To appear.

[3] M. R. Brown and R. E. Tarjan. The design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[4] R. Cole. On the dynamic finger conjecture for splay trees part II: The proof. Technical Report TR1995-701, Courant Institute, New York University, 1995.

[5] I. Galperin and R. Rivest. Scapegoat trees. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 165–174, 1993.

[6] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing (STOCS'77)*, pages 49–60, 1977.

[7] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[8] J. Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 516–522, 2001.

[9] J. Iacono and S. Langerman. Queaps. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation (ISAAC 2002)*, 2002. To appear.

[10] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[11] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(2):652–686, 1985.

[12] Daniel D. Sleator and Robert E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28:202–208, 1985.