PREPRINT. SIAM Journal of Computing, 24(5):1091-1103, 1995.

NEW TIGHT BOUNDS ON UNIQUELY REPRESENTED DICTIONARIES*

ARNE ANDERSSON† AND THOMAS OTTMANN‡

KEYWORDS: analysis of algorithms, data structures, dictionary problem, uniquely represented dictionaries, binary search trees

Abstract. We present a solution to the dictionary problem where each subset of size n of an ordered universe is represented by a unique structure, containing a (unique) binary search tree. The structure permits the execution of *search*, *insert*, and *delete* operations in $O(n^{1/3})$ time in the worst case. We also give a general lower bound, stating that for any unique representation of a set in a graph of bounded outdegree, one of the operations *search* or *update* must require a cost of $\Omega(n^{1/3})$. Therefore, our result sheds new light on previously claimed lower bounds for unique representation of dictionaries.

1. Introduction. A *dictionary* is a set of items on which *search*, *insert* or *delete* operations can be performed. The *dictionary problem* asks for a family of data structures to store the sets of items and for algorithms to carry out the dictionary operations efficiently. We consider a data structure as a graph consisting of nodes linked together by pointers, one item is stored in each node. The nodes represent the storage locations. Pointer paths correspond to access paths for the stored items.

In general, there can be many different structures which store the same set of items. We may perform a sequence of insert and delete operations starting with an initially empty structure in order to obtain a structure storing a given set of items. The shape of the obtained structure may depend on the size of the stored set, on the set (but not only on its size) or on the generation history.

We call a dictionary *set-uniquely* represented, if each *set* of items is represented by a unique data structure. In other words: For each set of items there is only one possible graph that represents the set. We call a dictionary *size-uniquely* represented if each set of the same *size* is represented by the same structure. Note that each size-unique representation is also set-unique. We also require that the values stored in the nodes are constrained by a fixed (for any graph) total order, that is, the representation is *order-unique*. The *unique representation problem* for dictionaries asks for efficient algorithms for maintaining a set- or size-unique representation of dictionaries.

A simple example of a size-unique representation of dictionaries is a sorted linked list. The randomized search trees of Aragon and Seidel [1] (with the random generator replaced by a universal hash function) is an example of a set-unique representation of dictionaries which is not size-unique.

It seems that deterministic solutions to the unique representation problem require more time for at least one of the three dictionary operations. Thus, a large variety of different structures to store the same set or to store sets of the same size seems to be the price to be paid for fast search and update. This observation was made already quite early by Snyder [5]. He shows that $\Omega(\sqrt{n})$ time is necessary for at least one of the three dictionary operations, if the dictionary is size-uniquely represented by a

* A preliminary version of this article appeared in Proceedings of Foundations of Computer Science, FOCS '91. This work was supported by grants from the National Swedish Board for Technical Development and from the Deutsche Forschungsgemeinschaft (DFG Ot 64/8–1).

† University of Lund, Department of Computer Science, Box 118, S-22100 Lund, Sweden, email: arne@dna.lth.se

‡ Universität Freiburg, Institut für Informatik, Rheinstr. 10–12, 79104 Freiburg, Germany, email: ottmann@informatik.uni-freiburg.de

tree-like search structure. He also presents a structure which he calls the "jelly-fish" for which each of the three dictionary operations search, insert, and delete can be performed in time $O(\sqrt{n})$. However, the computational model underlying Snyder's lower bound proof and the operations used in the update algorithms for the jelly-fish structure are not the same. Hence, as we will show in this paper, there is room for improvement.

Recently, Sundar and Tarjan [6] have studied the unique representation problem in a different context. They use a nondestructive CONS operation as the only primitive for creating and changing trees and they show that $\Theta(\sqrt{n})$ CONS operations are necessary and sufficient to maintain unique binary search trees.

In this paper we present new size- and order unique representations of dictionaries that allow to perform search operations in time $O(c \cdot n^{1/c})$ and updates in time $O(\sqrt{n})$ time when c is even and $O(n^{(c-1)/2c})$ when c is odd, for any $c \geq 2$. As primitives for creating and changing structures we allow pointer changes and to create and dispose nodes as Snyder [5] did. Similar to the structures of Snyder [5] and of Sundar and Tarjan [6], our structures are not "pure" tree structures but can be viewed as trees embedded in graphs. We also give a lower bound stating that $\Omega(n^{1/3})$ time is necessary per operation when a dictionary is size- and order-uniquely represented in a directed graph of bounded outdegree and pointer changes are the basic primitives to manipulate graphs.

Hence, we show that there is a huge gap between unique and non-unique representations of dictionaries. We feel that these findings point to a fundamental fact in the theory of data structures.

2. Model of Computation. We consider size- and order-unique representations of dictionaries by graphs of bounded outdegree ($\leq k$) and assume that for a given n there is only one graph of n nodes. Furthermore, we assume that for each graph the nodes are constrained by a fixed total order. The elements of a given set of size n are stored in the nodes of the graph in such a way that the i-th element is stored in the i-th node, for each i.

Each search starts at one specified node, called the *root*, and follows a number of edges until the searched element is found or the search ends unsuccessfully, because some termination condition has become true. All elements must be reachable from the root, and hence each node (except the root) has to have at least one incoming edge. The cost of a search equals the number of traversed edges plus one.

When performing an update a graph may be changed by one of the following operations:

- create/remove a node;
- change/add/remove one outgoing edge from a node (pointer change);
- Exchange elements between two nodes;

Each operation requires a cost of $\Theta(1)$. After a creation the node contains an element and has no outgoing edges. (Since the graph has its outdegree bounded by a constant k, we may add k outgoing edges in constant time.)

It should be pointed out that this cost somewhat underestimates the real cost of pointer changes and element exchanges, since we do not include the time required to locate the node where a pointer change or an exchange has to be performed. However, when presenting our upper bound we will not hide any costs by this simplification.

3. Semidynamic c-level jump lists. Snyder [5] has introduced the jelly-fish as a size-unique representation of dictionaries. We propose first a slightly different view of his structure. It leads to the 2-level jump list, which has the same $O(\sqrt{n})$ worst case time bound for all three dictionary operations. In order to simplify the presentation of 2-level jump lists we assume that $i^2 \leq n < (i+1)^2$. Hence, we assume that the size n of the dictionary does not vary unboundedly under insertions and deletions but stays in between the given bounds $i^2 \leq n < (i+1)^2$, for some fixed i.

A 2-*level jump list of size* n consists of a doubly linked list of n nodes $1, \ldots, n$; that is, for each p, $1 \leq p < n$, the nodes p and $p+1$ are linked together by a pair of 1-*level links*. We call the sequence of nodes linked together by 1-level links the 1-*level list*. Furthermore, the nodes 1, $i+1$, $2i+1$, $\ldots$, $\lfloor n/i \rfloor \cdot i + 1$ are linked together into the 2-level list which we also call the *top-level list*. Figure 1 displays the structure of 2-level jump lists.

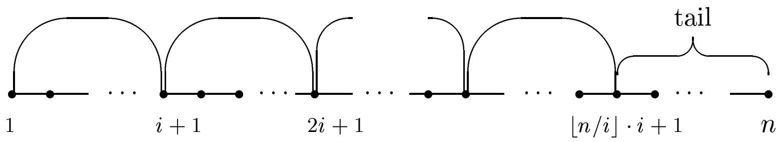

FIG. 1. 2-*level jump list of size* n

We require the elements of a set of n items to be stored in ascending order in the nodes 1, 2, $\ldots$, n. Thus, we obtain a size- and order-unique representation of dictionaries.

It should be clear how to *search* for an item by performing at most $2i$ key comparisons: Use the top-level list to determine the sequence of at most i nodes which may contain the item and perform a linear search among them following 1-level pointers. As long as n stays in the range $i^2 \leq n < (i+1)^2$, *updates* can also be performed in $O(i)$ steps: First, determine the position of the item which has to be inserted or deleted in the 1-level list. This takes $O(i)$ steps. Then insert (or delete) the element in the 1-level list. This is a constant time operation and has the effect that one sequence of nodes in the 1-level list spanned by a top-level-list pointer has become either too long by one (after an insertion) or too short by one (after a deletion). Therefore, some pointers of the top-level list have to be shifted by one position to the left or one position to the right.

Figure 2 show an example of an insertion of item 9 into a 2-level jump list of size 11 storing the set $\{2, 3, 5, 7, 8, 10, 11, 12, 14, 17, 19\}$. Note that an insertion increases the length of the tail of the 2-level jump list by 1. Therefore, the top-level list has to grow by one element as soon as the length of the tail exceeds i. Similarly, a deletion may require the shortening of the top-level list by one element. Adjusting the top-level list after an insertion or deletion takes time $O(i)$ in the worst case.

So far, 2-level jump lists are only semidynamic, because we did not allow n to vary freely. It is not difficult to see that the structure can be made fully dynamic without destroying its features. Maintain a pair of structures, a primary structure and a shadow structure, corresponding to two adjacent ranges of n. Smoothly construct or destruct both structures during insertions or deletions, and switch between the structures appropriately. Whenever n exceeds the upper bound of the interval $i^2 \leq n < (i+1)^2$ we must have a 2-level jump list of size n ready with top-level pointers of length $(i+1)$ instead of length i. Similarly, whenever n falls below i^2, a 2-level jump list with top-level pointers of length $(i-1)$ must be available. Because we can spend additional time i per update operation in order to construct (respectively, destruct) part of the structures for n in one of the neighboring intervals $(i+1)^2 \leq n < (i+2)^2$, or $(i-1)^2 \leq n < i^2$ and because the length of the interval $i^2 \leq n < (i+1)^2$ is $\Theta(i)$,

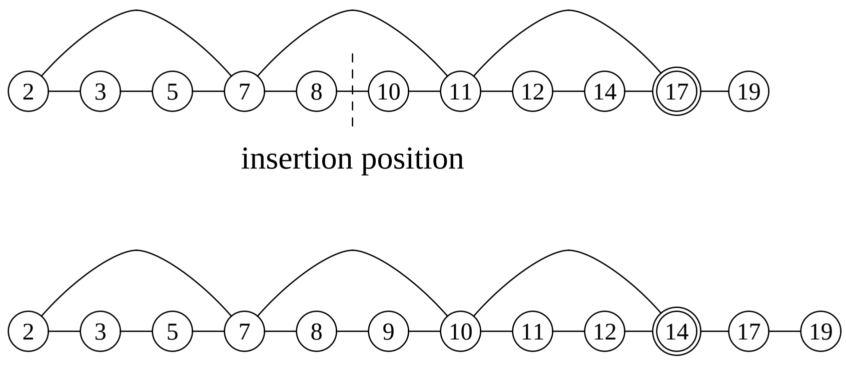

FIG. 2. *Insertion of 9 into a 2-level jump list*

we have time $\Theta(i^2) = \Theta(n)$ to construct the desired structures. This is certainly sufficient. We avoid a more detailed description, because we will show in the next section that a much more elegant solution to the problem of fully dynamizing the structure is possible.

The 2-level jump list and Snyder's jelly-fish are similar. The jelly-fish consists of a binary search tree of size $\Theta(\sqrt{n})$ where each leaf is associated with a circular list of size $\Theta(\sqrt{n})$, cf. Figure 9a, the 2-level jump list is illustrated in Figure 9c. A minor difference is that the tree is replaced by a linked list; a major difference is that in our structure no nodes are distinguished as leaves.

We will now introduce c-level jump lists, for every $c \geq 3$, as natural generalizations of 2-level jump lists. For convenience, let us assume that $i^c \leq n < (i+1)^c$ for some fixed i. A *c-level jump list of size n* consists of n nodes $1, \ldots, n$. The nodes are linked together by pointers, arranged in levels:

Lower levels: For each j, $1 \leq j \leq \lceil c/2 \rceil$, and for each p, $1 \leq p \leq n - i^{j-1}$, the nodes p and $p + i^{j-1}$ are linked together by a pair of *j-level links*.

Upper levels: For each j, $\lceil c/2 \rceil + 1 \leq j \leq c$, the nodes $1, 1 \cdot i^{j-1} + 1, 2 \cdot i^{j-1} + 1, 3 \cdot i^{j-1} + 1, \ldots$ are linked together, leaving at most $i^{j-1} - 1$ nodes in a tail.

The sequence of nodes of a c-level jump list linked together by j-level links is called a *j-level list*. A j-level list has maximal length $\lfloor n/i^{j-1} \rfloor = O(i^{c-j+1})$. Note the difference between the lower and upper levels. In the lower levels, each node is involved in a j-level list, while the upper levels only contain one j-level list each, involving only a few nodes.

Figure 3 shows the structure of a 3-level jump list of size 30.

Note that a c-level jump list of size n requires $O(c \cdot n)$ space.

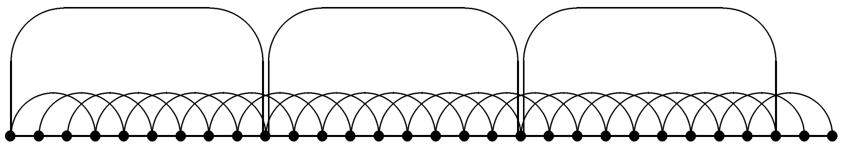

FIG. 3. 3-*level jump list of size* 30

Again, we require the elements of a set of items of size n to be stored in ascending order in the nodes $1, \ldots, n$ in a c-level jump list of size n. This gives a size- and order-unique representation of dictionaries.

In order to *search* for an item start with the topmost list and determine the sequence of at most i^{c-1} nodes which may contain the searched item. Then, for each $j = c-1,\ c-2, \ldots, 1$ follow a sequence of j-level pointers to determine the position of the searched item in the j-level list until it has been found or j has become 1 and the item has not been detected at its expected place in the 1-level list. Note that for each j, $c-1 \geq j \geq 1$, searching is restricted to a j-level sublist of length at most i. In this way a successful or unsuccessful search can be performed in $O(c \cdot i) = O(c \cdot n^{1/c})$ time in the worst case.

In Figure 4 a possible search path in the 3-level jump list of Figure 3 has been displayed by means of bold drawn arrows and links.

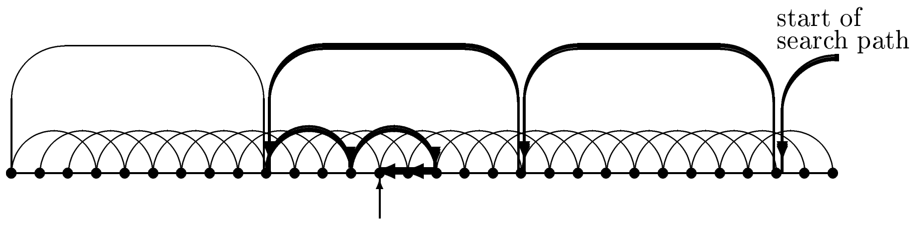

FIG. 4. *Example of a possible search path*

In order to *insert* an item into a c-level jump list first determine the expected position of the new element by a search as explained above. Then, the element is inserted into *all* j-level lists, $1 \leq j \leq \lceil c/2 \rceil$. For each j, $1 \leq j \leq \lceil c/2 \rceil$, all j-level links "jumping over" the inserted element will be adjusted, see Figure 5. That is, an insertion may be considered as the simultaneous insertion of the same new element into i^{j-1} ordered doubly linked lists, for all j, $1 \leq j \leq \lceil c/2 \rceil$. This takes time $O(1 + i + i^2 + \ldots + i^{\lceil c/2 \rceil - 1}) = O(i^{\lceil c/2 \rceil - 1})$ altogether. Next, the pointers of all nodes in the upper level lists to the right of the insertion position have to be shifted by one position to the left. This takes time $O(\sum_{j=\lceil c/2 \rceil + 1}^{c} n/i^{j-1}) = O(\sum_{j=\lceil c/2 \rceil + 1}^{c} i^{c-j+1}) = O(i^{\lfloor c/2 \rfloor})$ in the worst case. The total cost is $O(i^{\lceil c/2 \rceil - 1} + i^{\lfloor c/2 \rfloor})$ which splits into two cases, depending on whether c is even or odd. Thus, it takes $O(\sqrt{n})$ time when c is even and $O(n^{(c-1)/2c})$ when c is odd to insert a new element into a c-level jump list of size n such that a c-level jump list of size $n+1$ is obtained.

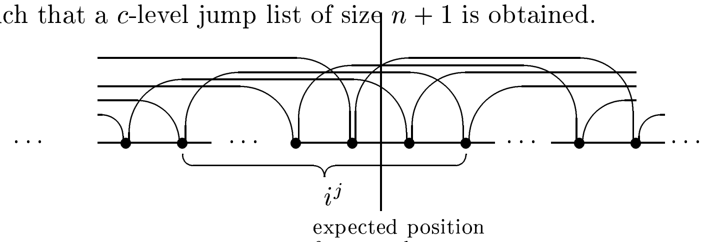

FIG. 5. *j-level lists affected by an insertion*

Deletion can be performed in the analog manner at the same cost; we just make an insertion backwards.

Again, the structure can be fully dynamized by simultaneously maintaining a pair of structures without destroying its features. We summarize our discussion by the following theorem.

THEOREM 3.1. *For each $c \geq 3$ the c-level jump lists yield a size-and order-unique representation of dictionaries requiring space $O(c \cdot n)$. Dictionary operations are supported at the following worst case costs:*

search: $O(c \cdot n^{1/c})$;

insertion and deletion: $O(\sqrt{n})$ *time when c is even and* $O(n^{(c-1)/2c})$ *when c is odd.*

Choosing $c = 3$ in this theorem balances search and update time and leads to a somewhat surprising result in the light of Snyder's [5] lower bound of $\Omega(\sqrt{n})$:

COROLLARY 3.2. *3-level jump lists are a size- and order-unique representation of dictionaries which allow to perform all three dictionary operations (search, insert, delete) in time* $O(n^{1/3})$ *in the worst case.*

It should be noted that any odd value of c gives us a violation of the previous lower bound. The relation between this lower bound and our new upper bound is discussed at the end of this article.

4. A fully dynamic 3-level structure. In this section we will give a second proof of Corollary 1 by designing a fully dynamic 3-level structure. Hence, we will not refer to the global technique of fully dynamizing the semidynamic 3-level jump list. Instead we will convince the reader by an explicit construction that it is possible to "beat" Snyder's lower bound. The new structure, simply called the *jump list*, consists of a directed graph in which a binary search tree is contained. As we will show, both the structure and its maintenance are quite simple.

4.1. Data Structure. A jump list of size n consists of n nodes $1, \ldots, n$. These nodes are linked together by three types of pointers:

1st level: The nodes $1, \ldots, n$ are linked together in sorted order in a doubly-linked list;

2nd level: For each i, $1 \leq i \leq n - \lfloor n^{1/3} \rfloor$, there is a pointer from node i to node $i + \lfloor i^{1/3} \rfloor$;

3rd level: The nodes $1^3, 2^3, 3^3, \ldots, \lfloor n^{1/3} \rfloor^3$ are linked together by backward pointers, that is, there is a pointer from node i^3 to node $(i-1)^3$ for each i, $1 < i \leq \lfloor n^{1/3} \rfloor^3$.

The above specification gives a size- and order-unique representation of dictionaries. It is not hard to see that the structure may be viewed as a binary search tree with some additional pointers. The nodes linked together on the 3rd level make up a left path with the first one of them (from the right) as the root. From each node on this path there is a right path of 2nd level pointers. Finally, from each node on a right path there is a left path of 1st level pointers.

More detailed: The length of the 3rd level path, that is the total number of 3rd level pointers, is $\lfloor n^{1/3} - 1 \rfloor$. The right subtree of the node at position i^3 contains the elements between this node and its parent (at position $(i+1)^3$). The number of elements in this interval (including position i^3) is $3i^2 + 3i + 1$ and each 2nd level pointer in the interval points i positions forward. Thus, starting from position i^3 there is a path of right pointers consisting of $3i + 3$ 2nd level pointers, ending at position $(i+1)^3 - 1$. Finally, from each node on the right path there is a left path of length $i - 1$, consisting of 1st level pointers.

The elements at positions $\lfloor n^{1/3} \rfloor^3 + 1 \ldots n$ make up the right subtree of the root. Following a chain of 2nd level pointers from the root, we may not end up at the very last node of the doubly-linked list. Thus, the rightmost path in the tree may contain a tail of up to $\lceil n^{1/3} \rceil$ nodes.

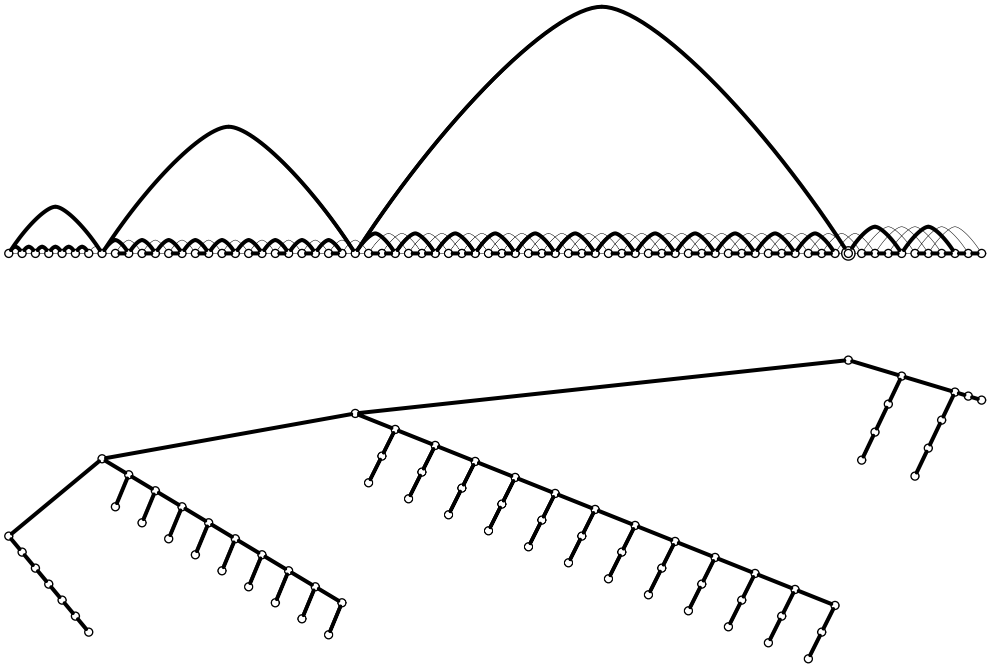

FIG. 6. *A jump list containing 74 elements and the embedded binary search tree.*

We call this tree an *embedded binary search tree.* Note the similarity between this structure and a threaded binary search tree [4]. In both cases the unused pointers at the bottom of the tree point to nodes instead of being nil pointers. An example of a jump list and its embedded binary search tree is given in Figure 6.

The jump list as described here requires 4 pointers per node. It is possible to use them either by specifying which pointers to use on each level, or by specifying which pointers to use as left and right pointers in the embedded tree. If desired, with some effort the number of pointers per node may be decreased to 3.

LEMMA 4.1. *A search in a jump list (using the embedded binary search tree) requires $O(n^{1/3})$ time in the worst case.*

Proof. Each root-to-leaf path down the embedded binary search tree is composed of three parts, corresponding to the three levels of the pointer structure. Each part has a length of $O(n^{1/3})$. From this the lemma follows. ☐

4.2. Maintenance Algorithms. When inserting/deleting an element, we follow a search path down the embedded binary search tree in order to find the update position. Then, the element is added to/removed from the 1st level list. The rest of the structure is adjusted by changing some 2nd level pointers in the neighbourhood of the update position and reconstructing a part (maybe all) of the 3rd level list.

In detail, during an *insertion* the data structure is modified in the following way:

1. The new element is inserted into the 1st level list (at position p);
2. A 2nd level pointer outgoing from p is created and each 2nd level pointer outgoing from a node to the left of p and ending to the right of p is shifted one position to the left, that is, it points now to the left neighbor of its

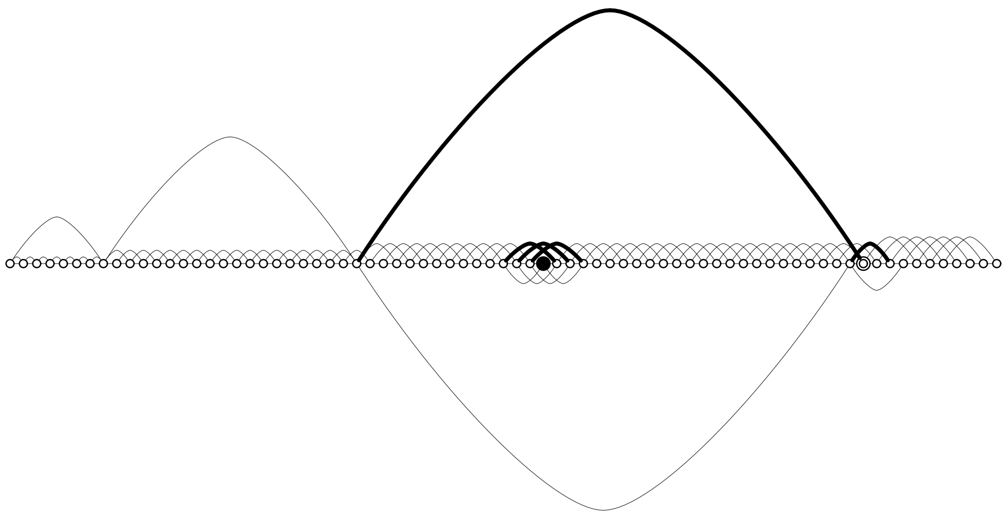

FIG. 7. *Performing an insertion at position 41 in the data structure of Figure 6.*

previous target node.

3. The 3rd level list is reconstructed such that each node in this list to the right of p is replaced by its predecessor. That is, the 3rd level pointer jumping over the insertion position has to be "shortened" by one and as a result of this all other 3rd level pointers to the right have to be shifted by one position to the left. If the tail becomes too long, eventually one new 3rd level pointer has to be created also. At each position where the 3rd level list is shifted we also have to change the 2nd level pointer.

The *deletion* algorithm works analogously by performing an insertion backwards.

Example: If we insert a new element between position 40 and 41 into the structure in Figure 6, we have to change the pointers that are marked by bold lines in Figure 7. The new pointers that occur are drawn downwards. □

LEMMA 4.2. *The cost of an update is $O(n^{1/3})$ in the worst case.*

Proof. From the description of the maintenance algorithms it follows that after locating the update position, $O(1)$ pointer changes are made on the first level and $O(n^{1/3})$ are made on the other two levels. Each pointer change requires $O(1)$ time, except when the length of the 3rd level path is increased. In that case a search is required to locate the last node. The total cost of this is $O(n^{1/3})$, which completes the proof. □

Altogether we have

THEOREM 4.3. *Jump lists are a representation of dictionaries with the following properties:*

- *they are size- and order-unique;*
- *a dictionary of size n requires $O(n)$ space;*
- *the cost to perform a search, insert or delete operation in a jump list which stores a dictionary of size n is $O(n^{1/3})$.*

Note that the operation of exchanging elements between nodes has not been used in our upper bound construction. We only use creation or deletion of nodes and pointer changes. Neither are there any "hidden" costs required to locate nodes where pointer changes should be made. All these locations can be found within the time bound of $O(n^{1/3})$.

5. A Lower Bound on Unique Representation. Below we give a lower bound on maintaining size- and order-unique representations of dictionaries. Recall that we presuppose graphs with bounded outdegree ($= k$) only. We assume the size of the graph to be unchanged and as an update we regard the operation of deleting one element and inserting another. That is, after an update the same graph must occur, only the stored elements may change. Recall also that there is a one-to-one mapping between nodes and elements. Thus, when arguing about the graph we may argue about elements connected by edges instead of nodes.

We say that element y is a *parent* of element x if there is an edge from y to x. The set of all x's parents is called the *parent-set* of x.

We assume that the elements are taken from an ordered universe and by the *rank* of an element x, denoted $\mathrm{rank}(x)$, we mean the number of elements smaller than x in the stored set plus one. Let x, y, and z be three elements such that $\mathrm{rank}(x) < \mathrm{rank}(y) < \mathrm{rank}(z)$. An edge from x to z has a *length* of $\mathrm{rank}(z) - \mathrm{rank}(x)$ and *covers* the element y. An edge from z to x has a negative length (and does also cover y). The *incoming pattern* of an element is given by the lengths of its incoming edges. Since the graph is unique, the incoming pattern of an element x is exactly determined by the rank of x. Thus, each rank is associated with a specific incoming pattern. We say that a rank r is *critical* if the pattern associated with r differs from the pattern associated with $r + 1$. An element that has a critical rank must change its incoming pattern when an update causes its rank to increase by one.

Note the difference between parent-sets and incoming pattern. The parent-set of x tells *which elements* are the parents of x, while the incoming pattern of x deals with *differences in ranks* between x and its parents.

In a graph G, the longest distance from the root to an element is denoted D_G, the largest number of nodes covering one node is denoted C_G, and the number of critical ranks is denoted P_G.

Example: In Figure 8 is shown a graph G with $k = 3$. The nodes are labeled with the ranks of the contained elements. The root is 2. The parent-set of element 3 is the set of elements 1 and 4. Elements 3 and 4 have the same incoming pattern (edges of lengths 2 and -1). This implies that the rank 3 is not critical, while all other ranks (except the last rank) are critical. All elements can be reached from the root by traversing at most two edges, thus $D_G = 2$. The element 3 is covered by four edges, ($1 \to 5$, $2 \to 4$, $2 \to 5$, and $4 \to 2$). However, two of the edges lead to the same element, so the number of elements covering 3 is three (2, 4, and 5). This is the maximum number of covering elements, thus $C_G = 3$. □

We start by showing some properties of a graph in Lemmas 5.1 and 5.2. Next, in Lemma 5.3 we compute the cost of structural changes. In Lemmas 5.4 - 5.7 we show that "bad" updates will enforce a high restructuring cost. Finally, the lower bound is given in Theorem 5.8.

LEMMA 5.1. *If $0 \neq 3D_GC_G < n$, then in each set of $3D_GC_G$ consecutive elements not containing the root there is at least one element with a critical rank.*

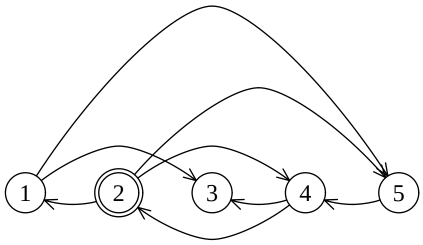

FIG. 8. *A graph.*

Proof. If there is no critical rank all elements must have the same incoming pattern. We prove the lemma by showing that this may not occur.

First, assume that to each element there is an edge of length L, $L > C_G + 1$. Then, the first element in the set is covered by at least $C_G + 1$ elements. In a similar way, if to each element there is an edge of length L, $L < -(C_G + 1)$, the last element in the set is covered by at least $C_G + 1$ elements. In both cases we get a contradiction with the definition of C_G.

Second, assume that to each element there is no edge longer than $C_G + 1$. Let p be the element with the median rank in the set. In order to reach p from outside the set we must pass a distance of $1.5 D_G C_G$. Doing this by following edges of length at most $C_G + 1$, we would have to follow more than $\frac{1.5 D_G C_G}{(C_G+1)} > D_G$ edges. We get a contradiction with the definition of D_G.

Thus, our assumption that all elements have the same incoming pattern leads to a contradiction. This completes the proof. □

LEMMA 5.2. *If* $0 \neq 3D_G C_G < n/2$ *then* $D_G \cdot C_G \cdot P_G = \Omega(n)$

Proof. From Lemma 5.1 we know that all (but one) sets of $3D_G C_G$ consecutive elements contain one element with a critical rank. This gives that

$$P_G \geq \frac{n}{3D_G C_G} - 1 = \Omega\left(\frac{n}{D_G C_G}\right) \tag{1}$$

The last equality follows since $3D_G C_G < n/2$. □

LEMMA 5.3. *To change the parent-sets of* m *elements requires a cost of* $\Omega(m)$.

Proof. We prove the lemma by showing that each restructuring operation changes the parent-set of $O(k) = O(1)$ elements. From this fact it follows that $\Omega(m)$ restructuring operations are required in order to change the parent-sets of m elements. Recall that from the model of computation we have the following possible operations:

- create a node. This operation does not affect the parent-set of any element. (If we allow the new node to have k outgoing edges, the number of changed parent-sets would be k).
- remove a node. This operation affects the parent-set of at most $k+1$ elements: the element that was in the node and the elements reached from that node.
- change/add/remove one outgoing edge from a node (pointer change). This operation changes the parent-set of at most 2 elements: one looses a parent and one gets a new one.
- exchange elements between two nodes. This operation changes the parent-set of at most $2k + 2$ elements: the two exchanged elements and their children.

Thus, each operation changes the parent-set of $O(1)$ elements, which completes the proof. □

LEMMA 5.4. *Let x be an element with critical rank. Then, after the following update*

1. *delete the largest element*
2. *insert a new smallest element*

x can not have the same parent-set as before.

Proof. Let X denote the set of $n-1$ (consecutive) elements that is stored in the graph both before and after the update.

Assume that the parent-set of x is the same before and after the update. This implies that neither the deleted element nor the inserted element can be a parent of x. Thus, the parent-set only consists of elements in X. (Note that x also belongs to X.)

The update described in the lemma will cause the rank of each element in X to increase by 1, and hence the difference in rank between any two elements in X will be the same after the update as before the update. Thus, if x has the same parent-set before and after the update, each incoming edge will have the same length, and thus the incoming pattern of x will be the same. We get a contradiction with the definition of a critical rank, which completes the proof. □

LEMMA 5.5. *There is an update which requires a cost of $\Omega(P_G)$.*

Proof. In order to prove the lemma we delete the largest element and insert a new smallest element. This implies that the rank of each element (except the first one) will increase by one. Thus, from Lemma 5.4 follows that each element that had a critical rank before the update must change its parent-set. This together with Lemma 5.3 completes the proof. □

LEMMA 5.6. *Let x and y be two elements, $rank(x) < rank(y)$, such that x is y's smallest parent. Then, after the following update*

1. *delete the smallest element*
2. *insert a new element between x and y (or anywhere if x was the smallest element)*

the parent-set of y must be different than before the update.

Proof. The lemma is trivially true if x was the smallest element and therefore deleted.

Otherwise, we prove the lemma by showing that after the described update, there can not be an edge from x to y.

After the described update, the rank of y is unchanged, and thus y has the same incoming pattern. This implies that the difference in rank between y and its smallest parent (i. e. the length of y's longest incoming edge) must be the same as before the update. However, after the update the rank of x is decreased by one. Thus, if there would be an edge from x to y after the update, the length of the longest incoming edge of y would change. We get a contradiction, which completes the proof. □

Note that there is a symmetric version of Lemma 5.6 concerning y's largest parent (with a larger rank). Because of the w.l.o.g.-assumption below, this symmetric lemma is not needed.

LEMMA 5.7. *There is an update which requires a cost of $\Omega(C_G)$.*

Proof. W. l. o. g. we assume that there is an element p which is covered by $\Theta(C_G)$ elements $q_1, q_2, \ldots$ with larger ranks. In order to prove the lemma we delete the smallest element and insert a new element which becomes the immediate successor of p in the ordered set. This will have the effect that the situation described in Lemma 5.6 will occur for each element q_i. This fact together with Lemma 5.3

completes the proof. □

THEOREM 5.8. *For any size- and order-unique graph representation of a dictionary there is a dictionary operation which requires $\Omega(n^{1/3})$ time.*

Proof. If $C_G = 0$ the graph G must be a linked list with $D_G = \Omega(n)$. Hence, w.l.o.g. we may assume that $C_G > 0$. From Lemma 5.2 follows that at least one of the three D_G, C_G, and P_G has to be $\Omega(n^{1/3})$. Since the search cost is $\Omega(D_G)$ and the update cost is $\Omega(\mathrm{Max}(C_G, P_G))$, either a search or an update has to require $\Omega(n^{1/3})$ time. □

Note that if an update involves a search for the update position, then the update requires $\Omega(n^{1/3})$ time.

Note also that our lower bound for size-uniqueness may be transformed to be valid for set-uniqueness by using Ramsey's theorem [2] in the same way as Sundar and Tarjan [6] did.

6. Comments and open problems. We feel that our findings point to a fundamental fact in the theory of data structures. In particular, in the presence of earlier lower bounds for unique representations [5, 6], stating that $\Theta(\sqrt{n})$ time is required per dictionary operation, the new upper bound presented above might seem surprising.

The lower bound given by Snyder [5] is based on the implicit assumption that changing the status of an element from being stored in a leaf of a binary tree to being stored in an internal node (or vice versa) requires $\Omega(1)$ time. As we have shown, this change can be achieved without explicitly performing any operation at the node, if the binary search tree is embedded in a directed graph appropriately.

The lower bound by Sundar and Tarjan, stating that $\Theta(\sqrt{n})$ CONS operations are required per update, is roughly based on the following argument: By chosing an "adversary" sequence of updates in a uniquely represented binary search tree we can at each update enforce the occurrence of $\Omega(\sqrt{n})$ new subtrees, which have never existed before. From the assumption that each occurring subtree has to be constructed at some moment, and that the construction requires $\Omega(1)$ time per subtree, a lower bound of $\Omega(\sqrt{n})$ time per update follows.

The first argument, that $\Omega(\sqrt{n})$ new trees may occur per update, is true for all uniquely represented dictionaries by binary search trees, also for the "embedded" trees presented here. However, all new subtrees may not need to be explicitly constructed. The assumption that $\Omega(1)$ cost per new subtree is needed seems reasonable if CONS is the only primitive to manipulate trees. But it does not hold if search trees are embedded in directed graphs which can be updated by pointer changes.

It is intuitively clear that CONS operations may be implemented by creating or removing nodes and changing the pointers of a set of nodes in a directed graph. Hence, it is possible to obtain a size- and order-unique representation of dictionaries which is based on the ideas of Sundar and Tarjan but uses pointer changes as primitives. This leads to the notion of *shared search trees* [3]. Shared search trees are a size- and order-unique representation of dictionaries requiring space $O(n \log n)$. For a shared search tree storing a dictionary of size n, a search operation can be performed in time $O(\log n)$ and updates in time $O(\sqrt{n})$. The shared search tree is a directed graph with nodes of unbounded degree $\Theta(\log n)$. It is not known whether there exists a set- and order-unique representation of dictionaries such that the underlying graph has bounded node-degree, a search operation can be performed in time $O(\log n)$, and updates take time $O(\sqrt{n})$ or less in the worst case using pointer changes as primitive operations.

There is a uniform way to evolve all the structures discussed in this paper, which

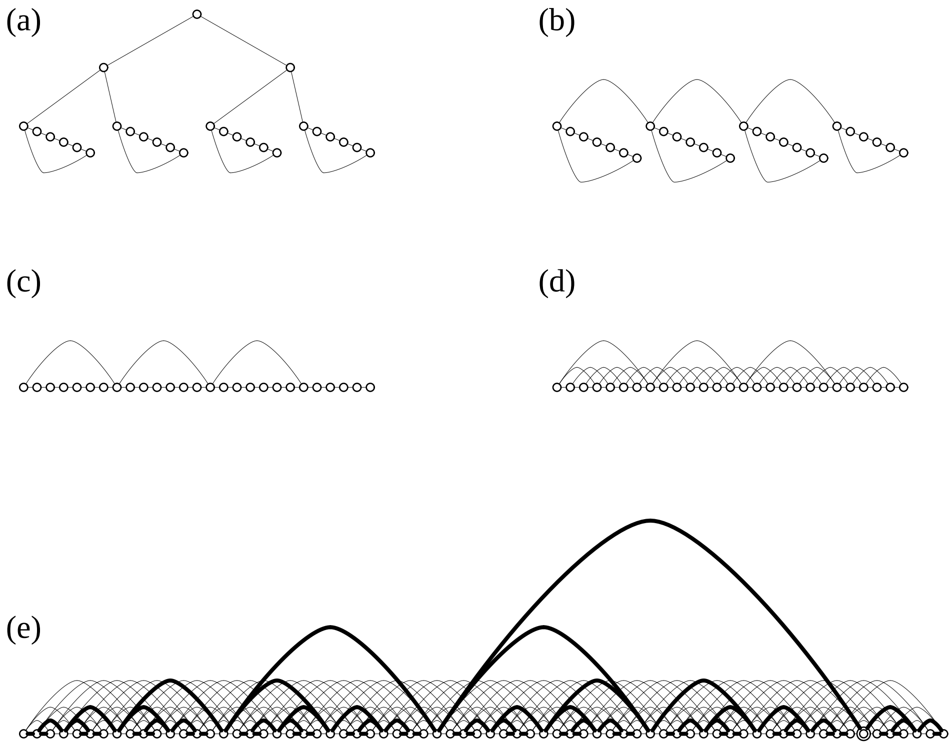

FIG. 9. *Evolution of structures.*

is illustrated in Figure 9.

(a) The jelly-fish structure of Snyder [5]. Each "tentacle" consists of a circular list.
(b) The same structure with the tree-shaped "body" replaced by a list.
(c) The edges connecting the top and bottom of each tentacle is replaced by a forward-link. We achieve a *2-level* jump list.
(d) The 2nd level links are added and we have the 3-level jump list.
(e) Using $\Theta(\frac{\log n}{2})$ levels we obtain the shared search tree, a structure similar to the one described by Sundar and Tarjan [6] with a search cost of $O(\log n)$ and an update cost of $O(\sqrt{n})$. The root is marked by a double circle.

Our results also raise the following more general problem: All known classes of (balanced) search trees are not set-unique representations of dictionaries because in general the same set of items may be represented by exponentially many different trees. Thus, there is a large gap between unique representations and commonly used classes of trees to represent dictionaries. The question is how efficiently search and update algorithms can operate if a dictionary of size n is represented by $f(n)$ different structures. In this paper we have dealt with the case $f(n) = O(1)$. The classical theory of (balanced) binary search trees is based on the assumption $f(n)$ is exponential. Nothing is known for any function f strictly in between these two extremes.

Acknowledgements. We would like to thank Bengt Nilsson, Sven Schuierer, and Svante Carlsson for comments and discussions. The comments by the anonymous referees have led to improvements both of the content and the presentation.

REFERENCES

[1] C. R. Aragon and R. G. Seidel, *Randomized search trees*, in Proc. 30th Annual IEEE Sympos. FOCS, 1989, pp. 540–545.

[2] R. L. Graham, B. L. Rotschild, and J. Spencer, *Ramsey Theory*, John Wiley & sons, 1980.

[3] T. Ottmann, *Trees - a personal view*, in New Results and New Trends in Computer Science, 1991. LNCS vol. 555.

[4] A. J. Perlis and C. Thornton, *Symbol manipulation by threaded lists*, Communications of the ACM, 1 (1960), pp. 195–204.

[5] L. Snyder, *On uniquely representable data structures*, in Proc. 18th IEEE FOCS, 1977, pp. 142–146.

[6] R. Sundar and R. E. Tarjan, *Unique binary search tree representations and equality-testing of sets and sequences*, in Proc. 22nd ACM STOC, 1990, pp. 18–25.