In Proc. 32nd IEEE Sympos. Foundations of Computer Science, pages 642--649, 1991.

Faster Uniquely Represented Dictionaries

Arne Andersson

Department of Computer Science
Lund University
S-221 00 Lund, Sweden
email: Arne.Andersson@dna.lth.se

Thomas Ottmann

Institut für Informatik
Universität Freiburg
Rheinstr. 10–12, W-7800 Freiburg, Germany
email: ottmann@informatik.uni-freiburg.de

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 binary search tree representations.

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, which stores the set in its nodes, one item per node. The nodes represent the storage locations. Pointer paths correspond to access paths for the stored items.

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 state 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. This problem has been studied first by Snyder [3]. He considers unique representations by "tree-like" structures of bounded degree and allows pointer changes in combination with the creation and abolition of nodes as primitives. He shows that $\Theta(\sqrt{n})$ is both necessary and sufficient to carry out an insert, delete or search operation on dictionaries of size n. Recently Sundar and Tarjan [4] have studied the problem in a different context. They use the 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 abstract we present a new size- and order-unique representation that allows to perform the dictionary operations in $O(n^{1/3})$ time in the worst case. As primitives for creating and changing structures we allow pointer changes and to create and dispose nodes as Snyder [3] did. Similar to the structures of Snyder, Sundar and Tarjan, our structures are not "pure" tree structures but can be viewed as trees embedded in a graph. We also give a matching lower bound stating that $\Omega(n^{1/3})$ time is necessary per operation when a dictionary is size- and order-uniquely represented in a graph of bounded outdegree.

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 An Improved Upper Bound on Unique Representation

The new data structure, the *jump list*, consists of a graph in which a binary search tree is contained. As we will show, both the structure and its maintenance are quite simple.

3.1 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.

We call this tree an *embedded binary search tree*. Note the similarity between this structure and a threaded binary search tree [2]. 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 1.

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 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. □

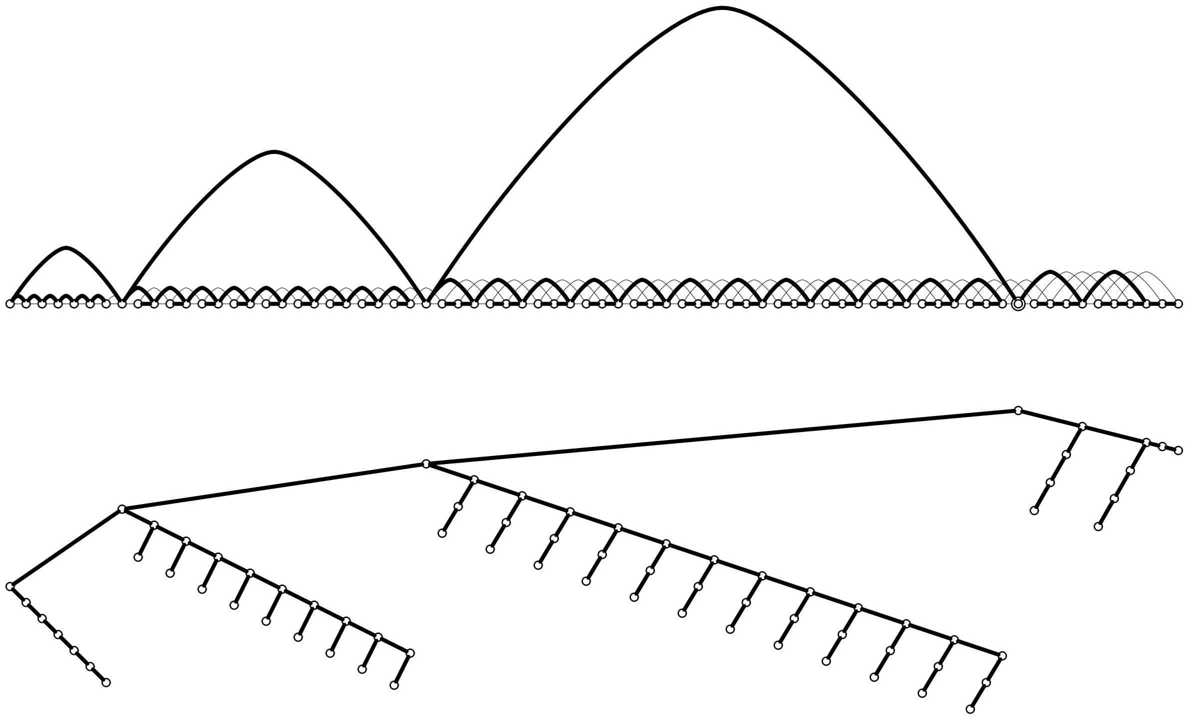

Figure 1: *A jump list containing 74 elements and the embedded binary search tree.*

3.2 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 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 1, we have to change the pointers that are marked by bold lines in Figure 2. The new pointers that occur are drawn downwards. □

Lemma 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

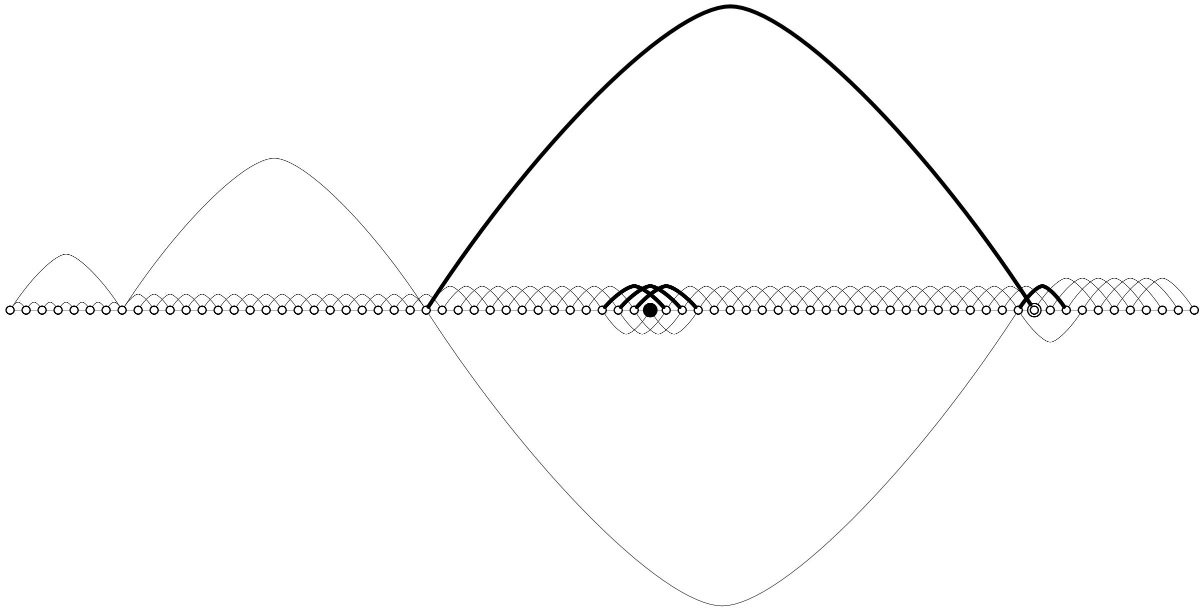

Figure 2: *Performing an insertion at position 41 in the data structure of Figure 1.*

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 1 *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 $O(n^{1/3})$.

4 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 constant 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.

By the *rank* of an element x, denoted $\text{rank}(x)$, we mean the rank of x in the stored set. Let x, y, and z be three elements such that $\text{rank}(x) < \text{rank}(y) < \text{rank}(z)$. An edge from x to z has a *length* of $\text{rank}(z) - \text{rank}(x)$ 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.

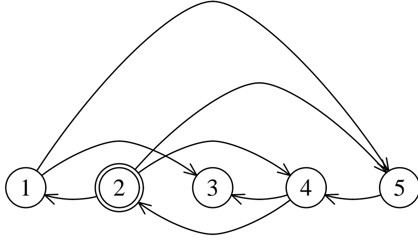

Figure 3: *A graph.*

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 3 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 two edges, thus $D_G = 2$. The element 3 is covered by four edges, (1→5, 2→4, 2→5, and 4→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 unique graph representation in Lemmas 3 and 4. Next, in Lemma 5 we compute the cost of structural changes. In Lemmas 6 - 9 we show that "bad" updates will enforce a high restructuring cost. Finally, the lower bound is given in Theorem 2.

Lemma 3 *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.*

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, $L > C_G$. 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, $L < -C_G$, 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. 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 $1.5D_GC_G$. Doing this by following edges of length at most C_G, we would have to follow more than $\frac{1.5D_GC_G}{C_G} > D_G$ 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 4 *The following is true for a graph G containing n elements:*

$$D_G \cdot C_G \cdot P_G = \Omega(n) \tag{1}$$

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 3 we know that all (but one) sets of $3D_GC_G$ consecutive elements contain one element with a critical rank. This gives that

$$P_G \geq \frac{n}{3D_GC_G} - 1 \tag{2}$$

From this the proof follows immediately. □

Lemma 5 *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 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 parents.

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

Lemma 6 *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 7 *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 6 follows that each element that had a critical rank before the update must change its parent-set. This together with Lemma 5 completes the proof. □

Lemma 8 *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 8 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 9 *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 immediately to the right of p. This will have the effect that the situation described in Lemma 8 will occur for each element q_i. This fact together with Lemma 5 completes the proof. □

Theorem 2 *For any size- and order-unique graph representation of a dictionary there is a dictionary operation which requires $\Omega(n^{1/3})$ time.*

Proof: From Lemma 4 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 $\Omega(\text{Max}(C_G, P_G))$, 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 [1] in the same way as Sundar and Tarjan [4] did.

5 Comments

In the presense of the claimed lower bounds for unique representations given in [3, 4], stating that $\Theta(\sqrt{n})$ time is required per dictionary operation, the new upper bound presented above might seem surprising. We might say that the given lower bounds in the referred papers are true in some special cases, while the claimed general implications are not.

The lower bound given by Snyder [3] is based on the implicit assumption that changing the status of an element from being stored in a unary node to being stored in a binary 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. Therefore, Snyder's assumption seems to be too restrictive.

The lower bound by Sundar and Tarjan, stating that $\Theta(\sqrt{n})$ Cons operations are required per update, is based on the following argument: By chosing a "bad" 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, 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 binary search trees, also for the tree presented here. However, all new subtrees may not need to be explicitly constructed. Therefore, the assumption that $\Omega(1)$ cost per new subtree is needed seems also be too restrictive in general (though reasonable if Cons is the only primitive to manipulate trees).

It is possible to extend the jump list used in our upper bound construction to h-level jump lists for arbitrary h. There is a uniform way to evolve all these structures, which is illustrated in Figure 4.

(a) The jelly-fish structure by Snyder. 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 get a structure similar to the one described by Sundar and Tarjan with a search cost of $O(\log n)$ and an update cost of $O(\sqrt{n})$ (details are left as an exercise). The root is marked by a double circle.

Acknowledgements

We would like to thank Bengt Nilsson, Sven Schuierer, and Svante Carlsson for comments and discussions.

This work was supported by grants from the National Swedish Board for Technical Development and from the Deutsche Forschungsgemeinschaft.

References

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

[2] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. *Communications of the ACM*, 1(2):195–204, 1960.

[3] L. Snyder. On uniquely representable data structures. In *Proc. 18th IEEE FOCS*, pages 142–146, 1977.

[4] R. Sundar and R. E. Tarjan. Unique binary search tree representations and equality-testing of sets and sequanses. In *Proc. 22nd ACM STOC*, pages 18–25, 1990.

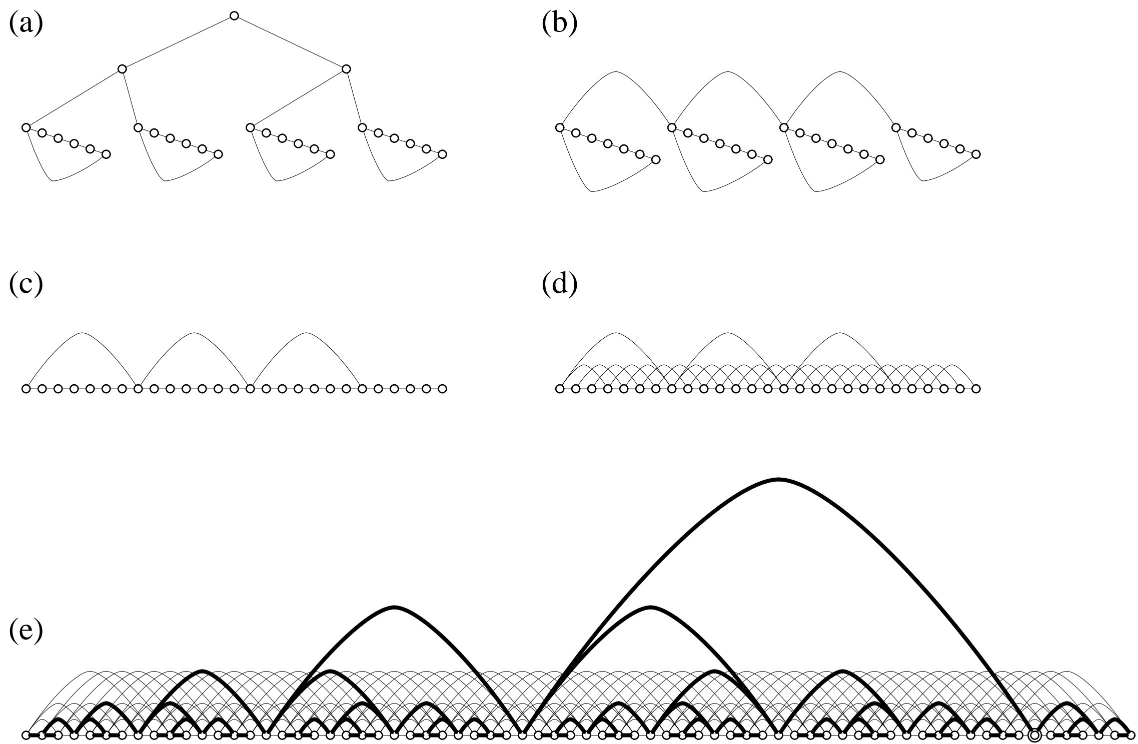

Figure 4: *Evolution of structures.*