Institute for Language, Logic and Information

DYNAMIC DEFERRED DATA STRUCTURES

Michiel H.M. Smid

ITLI Prepublication Series
For Computation and Complexity Theory CT-89-01

University of Amsterdam

DYNAMIC DEFERRED DATA STRUCTURES

Michiel H.M. Smid
Centre for Mathematics and Computer Science, Amsterdam

Received January 1989

Dynamic Deferred Data Structures

Michiel H.M. Smid*

January 10, 1989

Abstract

It is shown how dynamization techniques can be used to design dynamic versions of the deferred data structures of Karp, Motwani and Raghavan.

Keywords: Analysis of algorithms, deferred data structures, membership problem, balanced binary trees.

1 Introduction

The theory of searching problems is concerned with the design of algorithms and data structures that solve these problems efficiently. In the static case, we are given a data set S of n elements, and we are asked to answer a sequence of queries about this set.

Normally, we build a data structure storing S, and perform the queries on this structure. If $P(n)$ resp. $Q(n)$ denote the building time of the structure resp. the time to answer one query, a sequence of k queries takes time $O(P(n) + k \times Q(n))$.

Often a single query can be solved in $O(n)$ time, by just walking along the set S. So a sequence of k queries can be solved in this way in total time $O(k \times n)$.

Clearly, the first solution is only efficient if k is sufficiently large, whereas the second solution is only efficient for small values of k.

This leads to the problem of designing a strategy for solving a sequence of queries, where the length of this sequence is not known. So we are given a set n objects, and we have to perform an unknown number of k queries. The queries have to be carried out *on-line*, i.e. each query must be answered before the next one becomes available.

Karp, Motwani and Raghavan [5,7] consider this problem, and they give a technique, which they call *deferred data structuring*. In this technique, the search

*Bureau SION, Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. This work was supported by the Netherlands Organization for Scientific Research (NWO).

structure is built 'on-the-fly' during query answering. Each time a query is answered, parts of the data structure that do not exist at that moment, but that are necessary to answer the query, are built. These parts can then be used to answer future queries. They show e.g. that using this approach a sequence of k membership queries can be solved in total time $O(n \log k)$ if $k \leq n$. They also show that this is optimal: The number of comparisons needed to perform $k \leq n$ membership queries is $\Omega(n \log k)$. (In fact they prove that this lower bound even holds in the *off-line* case, i.e. in case the queries are known in advance.) Remark that by using a balanced binary search tree, the first mentioned solution takes total time $O((n+k) \log n)$, whereas the second solution takes $O(k \times n)$ time. So neither of these solutions achieves the lower bound for many values of k.

In [5], the authors ask for deferred data structures for dynamic data sets in which insertions and deletions are allowed concurrently with queries.

In this note we show that it is often possible to design dynamic deferred data structures by using well-known dynamization techniques. The ideas are illustrated by considering dynamic deferred structures for the membership query problem. We show that deferred binary search trees—if properly chosen—can be maintained as in the ordinary case, i.e. by means of rotations. We also adapt the partial rebuilding technique of Lueker [6], to get another maintenance algorithm for deferred search trees. Finally, we give a trivial solution, based on the ideas of decomposable searching problems and global rebuilding (see Bentley [2] and Overmars [9]).

2 The membership query problem

2.1 The static deferred binary search tree

In this subsection we first repeat the static solution of [5] for the membership problem.

Let S be a set of n elements drawn from some totally ordered universe U. We are asked to perform—on-line—a sequence of membership queries. In each such query we get an element q of U, and we have to decide whether or not $q \in S$.

The algorithm that answers these queries builds a binary search tree as follows. Initially there is only the root, containing the set S. Consider the first query q. We compute the median m of S, and store it in the root. Then we make two new nodes u and v. Node u will be the left son of the root, and we store in it all elements of S that are smaller than m. Similarly, v will be the right son of the root, and we store in it the elements of S that are greater than m. Then we compare the query object q with m. If $q = m$ we know that $q \in S$, and we stop. Suppose $q < m$. Then we proceed in the same way with node u. That is, we find the median of all elements stored in u, we store this median in u, we give u two sons with the appropriate elements, and we compare q with the new median. This procedure is repeated until we either find a node in which the 'local' median is equal to q, in which case we are finished, or end in a node storing only one element not equal to q, in which case we know that $q \notin S$.

Clearly, the first query takes time $O(n + n/2 + n/4 + \cdots) = O(n)$, since in each node we have to find a median, which takes linear time [3,10]. During this first query, however, we have built some structure that can be used by future queries: in the second query, we have to perform only one comparison in the root to decide whether we have to proceed to the left or right son. In fact, in any node we visit that is visited already before, we spend only one comparison.

This is the general principle in deferred data structuring: If we do a lot of work to answer one query, we do it in such a way that we can take advantage from it in future queries.

We now describe the algorithm in more detail. Each node v in the structure contains a list $L(v)$ of elements, two variables $N(v)$ and $key(v)$, and two pointers. Some of these values may be undefined. The value of $N(v)$ is equal to the number of elements that are stored in the subtree with root v. The meaning of the other variables will be clear from the algorithms below. (Strictly speaking, the variable $N(v)$ is not needed in the static case.)

Initialization. At the start of the algorithm there is one node, the root r. The list $L(r)$ stores all elements of S. The value of $N(r)$ is equal to n, which is the cardinality of S, and the value of $key(r)$ is undefined.

Expand. Let v be a node having an undefined variable $key(v)$. In this case, the list $L(v)$ will contain at least 2 elements, and the value of $N(v)$ will be equal to $|L(v)|$. The operation *expand* is performed as follows:

First we compute the median m of $L(v)$, and we determine the sets $S_1 = \{x \in L(v) : x < m\}$ and $S_2 = \{x \in L(v) : x > m\}$. Then we set $key(v) := m$, $L(v) := \emptyset$. Next we make two new nodes v_1 and v_2. Node v_1 will be the left son of v, so we store in v a pointer to v_1. If $|S_1| > 1$, we set $L(v_1) := S_1$, $N(v_1) := |S_1|$, $key(v_1) :=$ undefined. If $|S_1| = 1$, we set $L(v_1) := \emptyset$, $N(v_1) := 1$, $key(v_1) := s$, where s is the (only) element of S_1. (Of course, if $S_1 = \emptyset$, we do not create the node v_1.) Similarly for v_2.

Answering one query. Let q be a query object, i.e. we want to know whether or not $q \in S$. Then we start at the root, and we follow the appropriate path in the deferred tree, by comparing q with the values of key in the nodes we encounter. If one of these key values is equal to q we know that $q \in S$ and we are finished.

If we encounter a node v having an undefined variable $key(v)$, we expand node v, as described above. Then we proceed our query by comparing q with the value of $key(v)$. If $q = key(v)$, we know that $q \in S$, and we can stop. Otherwise, if $q < key(v)$, we expand the left son of v, and we continue in the same way. If this left son does not exist, we know that $q \notin S$. Similarly, if $q > key(v)$.

The following theorem gives the complexity of the algorithm. For a proof, see [5], or Subsection 2.2.

Theorem 1 ([5]) *A sequence of k membership queries in a set of n elements can be solved in total time $O(n \log k)$ if $k \leq n$, and $O((n+k)\log n)$ if $k > n$.*

2.2 Dynamic solutions

We only consider sequences of at most n queries, insertions and deletions. Clearly, this suffices, since after n operations we will have spent already $\Omega(n \log n)$ time. Therefore, in the n-th operation, we can build a complete data structure—in $O(n \log n)$ time—and continue in the standard way.

Consider the deferred tree of the preceding subsection. At some point in the sequence of queries, the structure consists of a number of nodes. Take such a node v.

Suppose $key(v)$ is defined. Then the list $L(v)$ is empty, the value of $N(v)$ is equal to the number of elements of S that are stored in the subtree with root v, and the value of $key(v)$ is equal to the median of all elements stored in this subtree.

If $key(v)$ is undefined, node v contains a list $L(v)$ storing a subset of S (those elements that 'belong' in the subtree of v), and the variable $N(v)$ has the value $|L(v)|$, which is greater than one.

Suppose we have to insert an element x. Then we start searching for x in the deferred tree, using the *key* values stored in the encountered nodes. In each node v we encounter, we increase the value of $N(v)$ by one, since the element x has to be inserted in the subtree of v.

If we end in a leaf, we insert x in the standard way, by creating a new node for it, and we set the variables L, N and *key* to their correct values. (A node v in the deferred tree is called a leaf if $N(v) = 1$. So a node that is not expanded—such a node does not have any sons—is not a leaf.) Note that if x is already present in the deferred tree, we will have encountered it.

Otherwise, we reach a node w with an undefined *key* value. Since we have to check whether x is already present in the structure, we have to walk along the list $L(w)$. (The list $L(w)$ is not sorted.) If x is a new element we add it to the list, and increase $N(w)$ by one. Note that this will take $O(|L(w)|)$ time. Hence a number of such insertions would take a lot of time. Then, our general principle—if we do a lot of work, we do it in such a way that it saves work in future operations—is violated. Therefore, after adding x to the list $L(w)$, we expand node w. So if we again have to insert an element in the subtree of w, the time for this insertion will be halved.

Clearly, we have to take care that the deferred tree remains balanced. We will consider this problem below.

A deletion of element x is performed in a similar way. We start searching for x.

First suppose we find a node v with $key(v) = x$. Then we search in the left subtree of v for the maximal element y. Clearly, we know the path that leads to this maximal element. If we end in a leaf, we interchange x and y, i.e., we set $key(v) = y$, and the *key* value of the leaf is set to x. Then we delete this leaf in the standard way. During the search we decrease the N values in all nodes we encounter by one. If we do not end in a leaf during our search for y, we reach a node w with an undefined *key* value. Then we remove y from the list $L(w)$, we set $key(v) = y$, and we expand node w. Just as in the insertion algorithm, if we again have to delete an element in the subtree of w, the time for this deletion is halved.

If we do not encounter a node v with $key(v) = x$ during our search for x, we might reach a node v with an undefined *key* value. If x is present in the deferred tree, it is stored in the list $L(v)$. So we delete x from this list, and we expand node v.

Remark that if x is not present in the tree we will find this out. Again we have to consider the problem of balancing the deferred tree.

There are various types of balanced binary search trees that can be maintained after insertions and deletions. The oldest are the AVL-trees [1]. The balance condition for these trees depends on the exact heights of subtrees. Since in our deferred trees, several subtrees are not complete during the sequence of operations, their exact heights will not be known. So AVL-trees seem not appropriate for deferred trees.

There is, however, a class of balanced trees, for which the balance criterion depends only on the size of its subtrees. For our deferred trees, the size of each subtree—whether it has been completely built already or not—is known at each moment: it is stored in the variable $N(v)$.

Definition 1 (Nievergelt and Reingold [8]) *Let α be a real number, $0 < \alpha < 1/2$. A binary tree is called a* BB[α]-tree, *if for each internal node v, the number of nodes in the left subtree of v divided by the total number of nodes in the subtree of v lies in between α and $1 - \alpha$.*

In this definition, nodes that contain only a small number of nodes in their subtree—say at most 5—do not have to satisfy the balance condition.

It was shown by Blum and Mehlhorn [4], that for a proper choice of α, the BB[α]-tree can be maintained after insertions and deletions in logarithmic time by means of single and double rotations.

Our deferred search tree will be a BB[α]-tree. That is, for each internal node v for which the value of $key(v)$ is defined, we require that $\alpha \leq N(v_l)/N(v) \leq 1 - \alpha$, where v_l is the left son of v. (Again, nodes that contain only a small number of elements in their subtree do not have to satisfy this balance condition.)

Updates are performed as described above. After the insertion or deletion, we walk back to the root of the deferred tree. Each node we encounter that does not satisfy the balance condition is rebalanced by rotations, as described in [4]. If a node is involved in a rotation that does not 'exist', i.e. its *key*-value is undefined, we first expand it. Therefore, the time for rebalancing after one single update can be linear. However, as was to be expected, future updates—and queries—take advantage from this.

Theorem 2 *A sequence of $k \leq n$ membership queries, insertions and deletions in a set of initially n elements can be performed in total time $O(n \log k)$.*

Proof. Let $f(n,k)$ denote the total time to perform a sequence of k queries and updates in a set of initially n elements, with the above algorithms. Then

$$f(n,k) \leq \max_{0 \leq k_1 \leq k} \{f(n/2, k_1) + f(n/2, k - k_1)\} + O(n) + O(k).$$

Here the term $O(n)$ is the time required to expand the root of the deferred tree, which takes linear time (see [3,10]) and which has to be done only once. The term $O(k)$ is the total number of comparisons made in the root in the k operations to guide searches, and the time to carry out rotations in the root. If k_1 operations visit the left subtree of the root, we spend a total amount of time there bounded by $f(n/2, k_1)$, since the left subtree initially contains $n/2$ elements. In the right subtree we spend $f(n/2, k - k_1)$ time. It can easily be shown that the solution of this recurrence satisfies $f(n,k) = O(n \log k + k \log n)$. This proves the theorem, since the function $x/\log x$ is increasing. □

There is another technique to achieve the result of Theorem 2. It is a generalization of the partial rebuilding technique [6,9].

Suppose we have a perfectly balanced BB[α]-tree storing a set of n elements. Here perfectly balanced means that for each internal node the sizes of its left and right subtree differ by at most one.

We can perform updates in this tree as follows. If we want to insert or delete an element, perform this update in the usual way. This gives a search path from the root to the leaf where the update has been carried out. Now walk back to the root, and find the highest node v that does not satisfy the balance condition of Definition 1 anymore (if such a node does not exist we are finished). Then rebalance the tree by replacing the entire subtree of v by a perfectly balanced subtree.

Clearly, if v is high in the tree this takes lot of time. In this case, however, it takes a lot of updates before node v is again the highest node that is out of balance.

We adapt this partial rebuilding technique to deferred data structures. Again the data structure is a deferred BB[α]-tree. Updates are performed as described above. Now, rebalancing is carried out as follows. After the insertion or deletion, we walk back to the root of the deferred tree to find the highest node v that is out of balance. Then we *dismantle* the subtree with root v. That is, we collect all elements that are stored in this subtree, and put them in the list $L(v)$. Furthermore, we set $key(v) :=$ undefined. (The value of $N(v)$ is already equal to $|L(v)|$.) Finally we discard all nodes below v. Such a dismantling operation takes $O(N(v))$ time.

Lemma 1 *Consider a node v at the moment it gets out of balance. Let N be the number of elements that are stored in the subtree of v at that moment. Then there must have been $\geq (1-2\alpha)N - 2$ updates since v was expanded.*

Proof. The proof can be found in [9, page 53]. Note that v was in perfect balance at the moment it was expanded, since we always split the list $L(v)$ along the median. □

Let $g(n,k)$ denote the total time to perform a sequence of k membership queries, insertions and deletions in a set of initially n elements, using the just described dismantling technique.

By Lemma 1, there is a constant c such that the root of the deferred tree cannot get out of balance in a sequence of $\leq cn$ updates. So in a sequence of $k \leq cn$ queries and updates, the root of the tree is expanded exactly once. The total time we spend in the root in such a sequence is therefore bounded by $O(n+k) = O(n)$. If k_1 operations are performed in the left subtree, we spend an amount of time there bounded by $g(n/2, k_1)$, since the left subtree initially contains $n/2$ elements. Similarly, we spend an amount of time $g(n/2, k-k_1)$ in the right subtree. It follows that

$$g(n,k) \leq \max_{0 \leq k_1 \leq k} \{g(n/2,k_1) + g(n/2,k-k_1)\} + c_1 n \quad \text{if } k \leq cn,$$

for some constant c_1.

Clearly, each query or update takes $O(m)$ time if m is the number of elements. So a sequence of k operations take $O(k(n+k))$ time, since the number of elements is always $\leq n+k$. It follows that

$$g(n,k) \leq c_2 k^2 \quad \text{if } k \geq cn,$$

for some constant c_2. (This upper bound will be overly pessimistic. In fact, by a more careful counting argument it should be possible to decrease this bound.)

It can easily be shown by induction that $g(n,k) = O(n \log k + k^2)$. So a sequence of $k \leq \sqrt{n}$ queries and updates takes $O(n \log k)$ time.

After $\sqrt{n}$ operations, we have spent already $\Omega(n \log n)$ time. Therefore, we build—in $O(n \log n)$ time—a binary tree for the elements that are present at this moment. The future operations are performed in this complete structure in the standard way.

This gives an alternative proof of Theorem 2.

Finally, we give yet another proof of Theorem 2. The method is based on the ideas of decomposable searching problems and global rebuilding [2,9].

We maintain two structures M and I. The main structure M is a static deferred binary search tree in which we store the n elements that are initially present. Each node v in this deferred tree for which the *key*-value is defined, also has a boolean variable $b(v)$, which says whether or not $key(v)$ is present. The structure I is an ordinary—i.e. non-deferred—balanced binary search tree, in which we store all new points. Initially, I is empty.

Suppose we have to insert element x. Then we do a membership query in the deferred tree M. If we find x, say in node v, we set $b(v) :=$ true. Otherwise, we insert x in I in the standard way.

A deletion of element x is performed as follows. First we do a membership query in the deferred tree M. If we find x, say in node v, we set $b(v) :=$ false. So we do not delete x, we only 'cross it out'. If we do not find x in M, we delete it from the tree I in the standard way.

To perform a query x, we first query the deferred tree M. If we find x, say in node v, we infer from $b(v)$ whether or not x is present. If we do not find x, we perform a membership query in the tree I.

Suppose we perform a sequence of $k \leq n$ operations in this way. In the tree M we perform k queries. By Theorem 1, the total time we spend there is $O(n \log k)$. In the tree I we perform a sequence of at most k queries and updates. Clearly, each such operation takes $O(\log k)$ time, since I stores at most k elements. Hence we spend $O(k \log k)$ time in the tree I. It follows that the total time for $k \leq n$ operations is bounded by $O(n \log k + k \log k) = O(n \log k)$. This yields a third proof of Theorem 2.

3 Concluding remarks

We have shown that known dynamization techniques can be used to design dynamic deferred data structures. We have illustrated this by means of the membership query problem.

An important problem in the design of dynamic data structures is to keep the structure balanced. By using BB[α]-trees, we do not need the complete structure to know whether a node is out of balance or not. So BB[α]-trees are the appropriate trees for dynamic deferred data structures.

The ideas given in this note can easily be extended to other searching problems. For example, for decomposable counting problems (see [9]) we can generalize the last solution of Subsection 2.2. This leads to a dynamic deferred solution of the ECDF-problem in [5].

The dismantling technique, which was an adaptation of the partial rebuilding technique [6,9], can be applied to get other dynamic deferred data structures. As an example, using this technique we again get a dynamic deferred ECDF-tree.

References

[1] G.M. Adel'son-Vel'skii and Y.M. Landis. *An algorithm for the organization of information.* Soviet Math. Dokl. **3** (1962), pp. 1259-1262.

[2] J.L. Bentley. *Decomposable searching problems.* Inform. Proc. Lett. **8** (1979), pp. 244-251.

[3] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest and R.E. Tarjan. *Time bounds for selection.* J. Comput. System Sci. **7** (1973), pp. 448-461.

[4] N. Blum and K. Mehlhorn. *On the average number of rebalancing operations in weight-balanced trees.* Theor. Comp. Sci. **11** (1980), pp. 303-320.

[5] R.M. Karp, R. Motwani and P. Raghavan. *Deferred data structuring.* SIAM J. Comput. **17** (1988), pp. 883-902.

[6] G.S. Lueker. *A data structure for orthogonal range queries.* Proc. 19-th Annual IEEE Symp. on Foundations of Computer Science, 1978, pp. 28-34.

[7] R. Motwani and P. Raghavan. *Deferred data structuring: query-driven preprocessing for geometric search problems.* Proc. 2nd Annual ACM Symp. on Computational Geometry, 1986, pp. 303-312.

[8] J. Nievergelt and E.M. Reingold. *Binary search trees of bounded balance.* SIAM J. Computing **2** (1973), pp. 33-43.

[9] M.H. Overmars. *The design of dynamic data structures.* Springer Lecture Notes in Computer Science, Vol. 156, Springer Verlag, 1983.

[10] A.M. Schönhage, M. Paterson and N. Pippenger. *Finding the median.* J. Comput. System Sci. **13** (1976), pp. 184-199.

The ITLI Prepublication Series

1986

86-01	The Institute of Language, Logic and Information
86-02 Peter van Emde Boas	A Semantical Model for Integration and Modularization of Rules
86-03 Johan van Benthem	Categorial Grammar and Lambda Calculus
86-04 Reinhard Muskens	A Relational Formulation of the Theory of Types
86-05 Kenneth A. Bowen, Dick de Jongh	Some Complete Logics for Branched Time, Part I Well-founded Time, Forward looking Operators
86-06 Johan van Benthem	Logical Syntax

1987

87-01 Jeroen Groenendijk, Martin Stokhof	Type shifting Rules and the Semantics of Interrogatives
87-02 Renate Bartsch	Frame Representations and Discourse Representations
87-03 Jan Willem Klop, Roel de Vrijer	Unique Normal Forms for Lambda Calculus with Surjective Pairing
87-04 Johan van Benthem	Polyadic quantifiers
87-05 Víctor Sánchez Valencia	Traditional Logicians and de Morgan's Example
87-06 Eleonore Oversteegen	Temporal Adverbials in the Two Track Theory of Time
87-07 Johan van Benthem	Categorial Grammar and Type Theory
87-08 Renate Bartsch	The Construction of Properties under Perspectives
87-09 Herman Hendriks	Type Change in Semantics: The Scope of Quantification and Coordination

1988

Logic, Semantics and Philosophy of Language:

LP-88-01 Michiel van Lambalgen	Algorithmic Information Theory
LP-88-02 Yde Venema	Expressiveness and Completeness of an Interval Tense Logic
LP-88-03	Year Report 1987
LP-88-04 Reinhard Muskens	Going partial in Montague Grammar
LP-88-05 Johan van Benthem	Logical Constants across Varying Types
LP-88-06 Johan van Benthem	Semantic Parallels in Natural Language and Computation
LP-88-07 Renate Bartsch	Tenses, Aspects, and their Scopes in Discourse
LP-88-08 Jeroen Groenendijk, Martin Stokhof	Context and Information in Dynamic Semantics
LP-88-09 Theo M.V. Janssen	A mathematical model for the CAT framework of Eurotra

Mathematical Logic and Foundations:

ML-88-01 Jaap van Oosten	Lifschitz' Realizabiility
ML-88-02 M.D.G. Swaen	The Arithmetical Fragment of Martin Löf's Type Theories with weak Σ-elimination
ML-88-03 Dick de Jongh, Frank Veltman	Provability Logics for Relative Interpretability
ML-88-04 A.S. Troelstra	On the Early History of Intuitionistic Logic
ML-88-05 A.S. Troelstra	Remarks on Intuitionism and the Philosophy of Mathematics

Computation and Complexity Theory:

CT-88-01 Ming Li, Paul M.B.Vitanyi	Two Decades of Applied Kolmogorov Complexity
CT-88-02 Michiel H.M. Smid	General Lower Bounds for the Partitioning of Range Trees
CT-88-03 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas	Maintaining Multiple Representations of Dynamic Data Structures
CT-88-04 Dick de Jongh, Lex Hendriks Gerard R. Renardel de Lavalette	Computations in Fragments of Intuitionistic Propositional Logic
CT-88-05 Peter van Emde Boas	Machine Models and Simulations (revised version)
CT-88-06 Michiel H.M. Smid	A Data Structure for the Union-find Problem having good Single-Operation Complexity
CT-88-07 Johan van Benthem	Time, Logic and Computation
CT-88-08 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas	Multiple Representations of Dynamic Data Structures
CT-88-09 Theo M.V. Janssen	Towards a Universal Parsing Algorithm for Functional Grammar
CT-88-10 Edith Spaan, Leen Torenvliet Peter van Emde Boas	Nondeterminism, Fairness and a Fundamental Analogy
CT-88-11 Sieger van Denneheuvel Peter van Emde Boas	Towards implementing RL

Other prepublications:

X-88-01 Marc Jumelet	On Solovay's Completeness Theorem

1989

Computation and Complexity Theory:

CT-89-01 Michiel H.M. Smid	Dynamic Deferred Data Structures