Data structures in a real-time environment

Patrick Lentfert and Mark H. Overmars

RUU-CS-88-1
January 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Data structures in a real-time environment

Patrick Lentfert and Mark H. Overmars

Technical Report RUU-CS-88-1
January 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

Data structures in a real-time environment

Patrick Lentfert and Mark H. Overmars

January 1988

Abstract

When a data structure is used in a real-time environment it must be able to handle updates immediately (without delay). As a result updates cannot always be performed on the data structure but must be placed in a buffer such that they can be performed later. We show that for a large number of query problems queries can still be answered reasonably efficient when a number of updates are in a buffer. Problems like member queries, nearest neighbor queries, k-th element and rank queries, range queries, etc. will be discussed. For most problems a query time of $O(N + \log n)$ is obtained where n is the number of objects in the data structure and N the number of updates in the buffer.

1 Introduction.

When we want to use a data structure in a real-time environment, we cannot use standard update techniques. New updates come in at some arbitrary speed and we must be able to handle them immediately. Hence, there might not be time to spend the e.g. $O(\log n)$ work that is normally required. The only thing we can do is put the update in a buffer and try to perform updates from the buffer when we have time to do so.

To formalize this a bit, consider the following situation. We have a processor P that holds and maintains some data structure D. There is a system A (possibly a user) that asks P to perform updates. These updates can appear in any speed. There is second system B (most likely a user) that asks queries about the data structure. B will only ask a new query when the previous one has been answered. The answer should be about the situation in the data structure at the moment the query is asked. Any updates coming in from A while processing the query are not taken into account (although they have to be stored in the data structure).

To handle updates quickly, P has a buffer BUF in which it stores the updates. BUF is organised as a FIFO (First In First Out). Whenever P has nothing to do, it is busy performing updates from BUF on D. This can be interrupted by new updates, that are just placed in BUF. It can also be interrupted by queries.

In this case, P has to take care that it leaves D in a good state. Moreover, the update currently being performed should not be forgotten. In most cases this means that, as long as we are still searching with the update in the data structure it should remain in the buffer. At the moment we have performed it, and we are only busy rebalancing, it should be removed from the buffer. We won't go into these implementation details here. So whenever we want to perform a query we can assume we have a data structure and a buffer.

Clearly, having a number of updates in a buffer will make the query time worse. We will have to look at all updates in the buffer. But we have to be even more carefull. For example, when we are interested in the maximal element in a set the following might happen: We search in D and find m to be the maximal. Next we look at the buffer. Now we might find out that m has been deleted in the meantime. This would mean that we again have to search in D to find a new maximum, etc. In this paper we will show how such things can be avoided and most problems can be solved in time $O(N+\log n)$ where n is the number of objects in D and N is the size of BUF.

The paper is organized as follows. In Section 2 we discuss a number of 1-dimensional searching problems like member searching, rank searching, nearest neighbor searching and k-th element searching. For all these problems we obtain a bound of $O(N+\log n)$. In Section 3 we concentrate on range searching and show that in a real-time environment we can solve the d-dimensional range searching problem in time $O(N\log N+\log^d n)$. Finally, in Section 4 we give some concluding remarks, extensions and direction for further research.

2 1-dimensional searching problems.

We will now show that most common 1-dimensional searching problems can be solved efficiently in a real-time environment. In all cases our structure consists of a balanced tree to solve the searching problem (see, e.g., [1]) and a list of insertions and deletions. We assume the list is divided into two lists L_d of deletions and L_i of insertions. We assume the tree contains n objects and there have been N updates, being N_d deletions and N_i insertions. We will assume that all updates are correct in the sense that when we insert an object it is not present and when we delete it it is present.

2.1 Member searching.

The member searching problem asks whether a given object p is present in the current set. We first prove a simple lemma on sets:

Lemma 2.1 *Given two (unsorted) sets A and B, we can determine in time $O(|A|+|B|)$ whether an element p is in $A-B$.*

Proof. Just count the number of occurences of p in both A and B. Let these numbers be N_a and N_b. If $N_a > N_b$ p is present, otherwise it is not. □

Theorem 2.2 *In a real-time environment the member searching problem can be solved in time $O(N + \log n)$.*

Proof. Search for the object p in the tree. If we find it let $A = L_i \cup \{p\}$, otherwise let $A = L_i$. Let $B = L_d$. Now we have to determine whether p is in $A - B$. The correctness follows from the assumption that all updates are correct. The time bound follows immediately. □

2.2 Rank searching.

The rank searching problem asks for the rank of a given object p in an ordered set of objects, i.e., the number of objects that is $\leq p$.

Theorem 2.3 *In a real-time environment the rank searching problem can be solved in time $O(N + \log n)$.*

Proof. Search for the rank of object p in the tree. Let this be r. This can be done in $O(\log n)$ time by usual techniques. Now determine the rank of p in L_i and L_d. Let these be r_i and r_d. This can be done by walking along both lists in time $O(N_i + N_d)$. Now the rank of p in the total set is $r + r_i - r_d$. □

This result can easily be extended to a whole class of searching problems. In e.g. [4] a class of so-called decomposable counting problems is defined:

Definition 2.1 *A searching problem PR is called a* decomposable counting problem *if and only if for each partition $A \cup B$ of the set V and any query object x,*

$$PR(x, A \cup B) = \square(PR(x, A), PR(x, B))$$

and

$$PR(x, V - A) = \triangle(PR(x, V), PR(x, A))$$

where both $\square$ and $\triangle$ can be computed in constant time.

In other words, answers to queries can be added (using $\square$) and subtracted (using $\triangle$). See [4] for examples of decomposable counting problems and their properties.

Theorem 2.4 *Let S be a data structure for a decomposable counting problem PR that has a query time of $Q(n)$. In a real-time environment we can solve PR in query time $O(N + Q(n))$.*

Proof. The method is trivial. We determine the answer a using the data structure S. Next we determine the answer a_i to the problem over the list L_i and the answer a_d over the list L_d. (Determining the answers over the lists can be done in time $O(N)$ by solving the problem for the individual elements and adding them up using $\square$.) Now the answer to the query is $\triangle(\square(a, a_i), a_d)$. □

2.3 Maximum searching.

The maximum searching problem asks for the maximum element in a set. When we maintain a pointer to the largest element in a tree this problem can normally be solved in time O(1). In a real-time environment this is a lot harder. The reason is that the largest element might have been deleted in the meantime but this deletion is still in the buffer. We will again first prove a lemma on sets:

Lemma 2.5 *Given two (unsorted) sets A and B with $B \subseteq A$, we can determine in time $O(|A| + |B|)$ the maximum element in $A - B$.*

Proof. The method works as follows. If A is empty then there is no maximum. Determine the median m of $A \cup B$. Split A in three sets $A_<$ of elements smaller than m, $A_=$ of elements equal to m and $A_>$ of elements larger than m. Similar we split B in sets $B_<$, $B_=$ and $B_>$ of elements smaller, equal and larger than m. If $|A_>| > |B_>|$ the maximum element will be in $A_>$ and we repeat the process with $A_>$ and $B_>$. Else, if $|A_=| > |B_=|$ then m will be the maximum element. Else, repeat the process with $A_<$ and $B_<$.

The correctness of the method follows from the fact that $B \subseteq A$. As a result it is impossible that $|A_>| < |B_>|$ and if $|A_>| = |B_>|$ then $A_> = B_>$ and, hence, we do not have to consider these elements anymore.

Finding the medium and splitting the two sets takes time $O(|A| + |B|)$. When we repeat the process, the total number of objects in the sets has been halved. Hence the total complexity is $O((|A| + |B|)(1 + 1/2 + 1/4 + ...)) = O(|A| + |B|)$. □

Theorem 2.6 *In a real-time environment the maximum searching problem can be solved in time $O(N + 1)$.*

Proof. If the tree contains less than $N_d + 1$ elements we put all the elements in the tree, together with all the insertions in a set A and all the deletions in a set B. Compute the maximum element in $A - B$ as above.

Otherwise we determine the $N_d + 1$ largest elements in the tree. Put these in a set A. This can be done in time $O(N_d + 1)$. Let p be the smallest of them. Also take all elements from L_i that are $\geq p$ and add them to A. This can be done in time $O(N_i + 1)$. Take all element from L_d that are $\geq p$ and put them in set B. This will take time $O(N_d + 1)$. Now obvious the maximum element in the set is the maximum element in $A - B$ and $B \subset A$. The bound follows from Lemma 2.5. □

2.4 Nearest neighbor searching.

The 1-dimensional nearest neighbor searching problem asks for the element of the set nearest to a given element p. This obviously is either the largest element $\leq p$ or the smallest element $> p$.

Theorem 2.7 *In a real-time environment the nearest neighbor searching problem can be solved in time $O(N + \log n)$.*

Proof. We will only show how to determine the largest element $\leq p$. First search for p in the tree which takes time $O(\log n)$. Determine the $N_d + 1$ largest elements $\leq p$. Put these in a set A. Let p' be the smallest among them. To A add all insertions in L_i that are $\geq p'$ and $\leq p$. Form a set B of all deletion $\geq p'$ and $\leq p$. (If there are no $N_d + 1$ elements $\leq p$ we put in A all elements $\leq p$ and all insertions $\leq p$ and in B all deletion $\leq p$.) Now determine the largest element in $A - B$. (Note that $B \subseteq A$.)

In a similar way we determine the smallest element $> p$. The time bound follows from Lemma 2.5. □

2.5 k-th element searching.

The k-th element searching problem asks to store an ordered set of keys such that one can efficiently determine the k-th element in the set for any given k.

Lemma 2.8 *Given two (unsorted) sets A and B with $B \subseteq A$, we can determine in time $O(|A| + |B|)$ the k-th element in $A - B$.*

Proof. We assume the k-th element does exist. (This can easily be checked.) Determine the median m of $A \cup B$. Split A in three sets $A_<$ of elements smaller than m, $A_=$ of elements equal to m and $A_>$ of elements larger than m. Similar we split B in sets $B_<$, $B_=$ and $B_>$ of elements smaller, equal and larger than m.

If $|A_<| - |B_<| \geq k$ the k-th element will be in $A_<$ and we repeat the process with $A_<$ and $B_<$. Else set $k := k - |A_<| + |B_<|$. Now if $|A_=| - |B_=| \geq k$ then m will be the k-th element and we are done. Else set $k := k - |A_=| + |B_=|$ and repeat the process with $A_>$ and $B_>$.

The correctness of the method follows immediately from the fact that $B \subseteq A$. The time bound follows in the same way as in Lemma 2.5. □

Theorem 2.9 *In a real-time environment the k-th element searching problem can be solved in time $O(N + \log n)$.*

Proof. First determine the k-th element p in the tree. Now take the N_i elements before p and the N_d elements after p in the tree and put them in a set A. Let p_1 be the smallest and p_2 the largest element among them. In L_i determine all elements in between p_1 and p_2 and add them to A. Make a set B of all elements in L_d that lie between p_1 and p_2. Finally, let k_d be the number of deletions that are smaller than p_1 and k_i the number of insertions that are smaller than p_1. Now it is easily verified that the k-th element in the set is the $k - k_i + k_d$-th element in $A - B$.

The time bound follows from Lemma 2.8. □

3 Range searching.

In this section we will concentrate on the range searching problem. Given a set of points in some d-dimensional space, the *range searching problem* asks for all points that lie in some range $([a_1..b_1], ..., [a_d..b_d])$, i.e., those points $p = (p_1, ..., p_d)$ with $a_1 \leq p_1 \leq b_1$ and ... and $a_d \leq p_d \leq b_d$.

Many data structures have been proposed for solving the range searching problem. Here we will use the *range tree* (also called e.g. super B-tree). (See e.g. [2,3,5,6].) We will briefly describe the structure here.

A 1-dimensional range tree is a simple binary tree storing the keys in sorted order in its leaves. To perform searching efficiently we link the leaves in a double linked list. To perform a query with a range $[a..b]$ we search with a in the tree to determine the smallest key p larger or equal to a. Next, starting at p, we walk along the list of leaves until we find a key larger than b. This obviously takes time $O(k + \log n)$ where k is the number of reported answers.

A 2-dimensional range tree again is a balanced binary tree storing the points in the leaves, ordered by first coordinate. With each internal node δ we store an associated structure T_δ that contains all the points in the subtree rooted at δ in the form of a 1-dimensional range tree, sorted on second coordinate.

To perform a query with range $([a_1..b_1], [a_2..b_2])$ we search with both a_1 and b_1 in the main tree. For some time a_1 and b_1 will follow the same search path but at some internal node δ a_1 will go to the left son and b_1 will go to the right son. Let $\beta_1 \ldots \beta_i$ be the nodes that are rightson of a node on the search path of a_1 below δ and do not lie on the search path themselves. Silmilar, let $\gamma_1 \ldots \gamma_j$ be the nodes that are leftson of a node on the search path of b_1 below δ that do not lie on the search path themselves. All points with x-coordinate between a_1 and b_1 lie in exactly one of the subtrees rooted at the β and γ nodes. Of these points we have to find the ones whose y-coordinate lies between a_2 and b_2. For this we use the structures $T_{\beta_1} \ldots T_{\beta_i}$ and $T_{\gamma_1} \ldots T_{\gamma_j}$. All points in the x-range are stored in exactly one such structure and all points in these structures lie in the x-range. On each of these T-structures we perform a one-dimensional range query with range $[a_2 : b_2]$. It can easily be seen that such a query takes time $O(k + \log^2 n)$.

Theorem 3.1 *In a real-time environment, the 2-dimensional range searching problem can be solved in time $O(k + N \log N + \log^2 n)$.*

Proof. As a first step we take from both L_i and L_d the elements that lie in the range and put them in sets A and B, respectively. Next we sort both A and B lexicographical and we remove from A and B those elements that are in both sets. The elements that are left in A lie in the range and have not been deleted so we can report them.

What remains to be done is to remove from the answers to the query in the tree the elements that are left in B. We could do this by sorting all the answers

but this would take time $O(k \log k)$ and as k might be much larger than the size of B we want to avoid this.

Note that we already have the points in B sorted by first coordinate. Now we search with a_1 and b_1 in the tree to determine the β and γ nodes as described above. Next, we split B in sublists $B_{\beta_1} ... B_{\beta_i}$ and $B_{\gamma_1} ... B_{\gamma_j}$ of the deletions that lie in the different subtrees. Each of these sublists we order by second coordinate.

Next we search in each of the subtree. For each subtree below a β_l we get a list A_{β_l} of answers, sorted by second coordinate. Walking along both A_{β_l} and B_{β_l} simultaneously we can remove the deleted answers in linear time. Similar for the subtrees below a node γ_l. The time bound follows. □

(Note that, if we followed the trivial method of first performing all the updates in the buffer and then performing the query, the bound would be $O(k + N \log^2 n + \log^2 n)$.)

A d-dimensional range tree is a simple generalization of the 2-dimensional range tree. Again we build a binary tree on the first coordinate and with each internal node δ we associate a $d-1$-dimensional range tree of the points in the subtree rooted at δ, restricted to their last $d-1$ coordinates. Queries will take time $O(k + \log^d n)$.

Theorem 3.2 *In a real-time environment, the d-dimensional range searching problem can be solved in time $O(k + N \log N + \log^d n)$.*

Proof. The method works in the same way as in the two-dimensional case. Again we construct the lists A and B of insertions and deletions that lie in the range. We sort both sets by first coordinate and remove from A all elements that are deleted. Next we search in the tree. Every time we search in some associated structure T we have a list of deleted elements in T, sorted on the same coordinate as T. We either find subtrees of T and split the list in sublists (and resort these) or T is a 1-dimensional subtree and we find a list of answers from which we remove the deleted ones. Details are left to the reader. □

4 Conclusions.

We have shown how a number of searching problems can be solved in a real-time environment where updates have to be performed in constant time. In such a situation a buffer can be used to store the updates when we have no time to perform them immediately on the data structure. Methods were given to handle queries in such a case still reasonably efficient.

Searching problems like member searching, nearest neighbor searching, maximum searching, k-th element and rank searching, range searching, etc. were treated. For most problems a query time of $O(N + \log n)$ was obtained where n is the size of the data structure and N the size of the buffer.

Many interesting directions of research remain. We currently study other ways of storing the updates to replace the FIFO to make the size of buffers smaller (on the average). Also other searching problems, especially the multi-dimensional ones that come from computational geometry and its applications, are being studied. Another question is whether the overall run-time can be improved by using knowledge about the buffer, obtained while performing previous queries.

References

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The design and analysis of computer algorithms,* Addison-Wesley, Reading Mass., 1974.

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

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

[4] Overmars, M.H., *The design of dynamic data structures,* Lect. Notes in Computer Science 156, Springer-Verlag, 1983.

[5] Willard, D.E., *Predicate-oriented database search algorithms,* Garland Publishing Company, New York, 1979.

[6] Willard, D.E., and G.S. Lueker, Adding range restriction capability to dynamic data structures, *J. ACM* 32 (1985), 597-617.