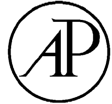

ACADEMIC PRESS

J. Parallel Distrib. Comput. 62 (2002) 1617–1628

Journal of Parallel and Distributed Computing

www.academicpress.com

Fully dynamic distributed search trees can be balanced in $O(\lg^2 N)$ time ☆

Fabio Barillari[a], Enrico Nardelli[b,c,*], and Massimo Pepe[a]

[a] *Dipartimento di Matematica Pura ed Applicata, Universitá degli Studi di L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italy*

[b] *Dipartimento di Informatica, Universitá degli Studi di L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italy*

[c] *Istituto di Analisi dei Sistemi ed Informatica, C.N.R., Viale Manzoni 30, I-00185 Rome, Italy*

Received 1 September 1997; received in revised form 13 October 1998; accepted 19 July 2002

Abstract

In this paper we consider the dictionary problem in a message-passing distributed environment. We introduce a new version, based on AVL-trees, of distributed search trees, the first to be fully scalable, that is, able to both grow and shrink as long as keys are inserted and deleted. We prove that in the worst case a key can be inserted, searched, or deleted with $O(\lg^2 N)$ messages. We show that for the introduced distributed search tree this bound is tight. Since the defined structure maintains the relative order of the keys, it can also support queries that refer to the linear order of keys, such as nearest neighbor or range queries.

© 2002 Published by Elsevier Science (USA).

Keywords: Scalability; Distributed data structures; Distributed memory machines

1. Introduction

In the past years more and more work has been devoted to the study and design of search structures that perform efficiently in a distributed environment e.g., [Dev93,JK93,KW94,LNS93,MS91,SPW90]. The rationale is that, due to the striking advance of communication technology, it is now feasible to gain computing power by running applications on a network of workstations [KW94,LNS94,LNS97]. Following the approach pioneered in [LNS93], we focus only on the efficiency aspects of distributing a search structure. The fundamental measure of the efficiency of an operation in this distributed context is the number of messages exchanged. A

☆ Parts of this work were done while Fabio Barillari was visiting the Eidgenössische Technische Hochschule, Zürich, whose financial support is gratefully acknowledged. Research described here is partially supported by the "Algoritmi, Modelli di Calcolo e Strutture Informative" 40%-Project of the Italian Ministry for University and Scientific & Technological Research (MURST) and by the "Chorochronos" Research Network of the ESPRIT Research Programme of the European Union.

*Corresponding author. Dipartimento di Informatica, Universitá degli Studi di L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italy.

E-mail address: nardelli@univaq.it (E. Nardelli).

0743-7315/02/$ - see front matter © 2002 Published by Elsevier Science (USA).
PII: S0743-7315(02)00009-6

critical requirement for the efficiency of a distributed search structure is its *scalability*. This implies that no server can play a master role, i.e., act as a bottleneck for access to data, and that the number of servers sharing the management of the data structure adapts to the (varying) number of keys stored to keep performance level almost constant.

Litwin, Neimat, and Schneider were the first to present and to discuss the paradigm of scalable distributed data structures, by proposing a distributed linear hashing, namely LH* [LNS93,LNS97]. Distributed hashing meeting the scalability requirement was investigated also by Devine [Dev93]. While offering good performances for exact search, hash-based techniques do not perform well for range queries, since they do not maintain the given order of data items. Comparison-based techniques, i.e., search trees, have to be considered for this purpose. Kröll and Widmayer [KW94,KW95] were the first to introduce an order-preserving scalable distributed data structure, by defining distributed random trees (DRT), a generalization to the distributed environment of binary search trees. Litwin, Neimat, and Schneider defined a distributed one-dimensional order-preserving data structure, namely RP* [LNS94], that can be seen as a generalization of B^+-trees. Litwin and Neimat presented also a k-dimensional distributed data structure [LN96]. Nardelli, Barillari, and Pepe [Nar95,Nar96,NBP97,NBP98] introduced and discussed distributed k-dimensional trees, an order-preserving structure suitable for the management of k-dimensional points, that can be used, via the mapping technique, to manage also extended objects defined in a lower dimension.

In the above-described proposals based on binary search trees no explicit action is taken to maintain the structure balanced. Hence a theoretical worst-case bound of $\Omega(N)$ holds for the search of a given key out of a set of N keys. The theoretical study of the characteristics of scalable distributed search trees conducted in [KW95] showed that if all the hypotheses used to efficiently manage search structures in the single-processor case are carried over to a distributed environment, then a lower bound of $\Omega(\sqrt{N})$ holds for the height of balanced binary search trees.

In this paper we relax some of these hypotheses (see Section 2 for more details) and show that binary search trees can be maintained balanced in a distributed environment so that search can be executed with $O(\lg^2 N)$ messages. We also prove that this bound is optimal for the introduced structure.

Also, while previous proposals explicitly considered only the semi-dynamic case, that is, the case where keys are only inserted and never deleted, we discuss deletion of keys and prove that it can be managed within the $O(\lg^2 N)$ bound. Hence this paper presents the first efficient distributed search structure to be fully dynamic and order-preserving.

The paper is organized as follows. In Section 2, distributed search trees and existing results are discussed. In Section 3, we describe the balanced and relaxed binary search tree, the data structure we have introduced to obtain the main result. Section 4 describes algorithms for insertion and deletion and Section 5 contains conclusions.

2. Scalable search structures in a distributed environment

The framework of this paper is the same for all schemes proposed in the literature for scalable search structures in a message-passing environment. A more detailed description can be found in [KW94,KW95,LNS97,NBP98]. We have a given set of *sites* (processor or nodes) connected by a network. Every site in the network is either a *server*, which manages data, or a *client*, which requests access to data. Each server

manages data items belonging to some part of the data domain. Sites communicate by sending and receiving *point-to-point* messages. We assume network communication is free of errors. Every server can store a single block of at most b data items, for a fixed constant b. We do not care whether a server stores keys in main or secondary memory, since our only concern is the number of messages exchanged. The case of a server storing more than one block of data items to improve overall performances is introduced and discussed in [VBW94].

The overall data organization scheme we focus on is a distributed binary search tree: the overall indexing structure is a binary search tree whose management is shared among all servers. Clients have partial copies of the index to guide the search process and to avoid starting all searches from the server managing the root. This means that:

- servers manage both nodes containing data items (*leaf nodes*) and nodes guiding the search process (*internal nodes*);
- clients are not, in general, up-to-date with the evolution of the structure, in the sense they have a local indexing structure, but do not know, in general, the overall status of the data structure;
- a server sends an update to a client index by communicating to the client, when answering a query, what the server itself knows about the overall structure: different clients may therefore have different and incomplete views of the data structure.

A client uses its index to send requests to servers according to its needs. Therefore the server managing the root of the overall tree is not a bottleneck since each client, after its first query, has some partial knowledge of the whole index. When a server receives a message from a client, it either directly serves the request or forwards it to some other server managing a more proper interval of the search space. Updates to client index are sent back to clients together with answers. As a consequence of these actions the distributed structure evolves and adapts itself to data, and clients adjust their views. For more details on the execution of operations in distributed search structures see [LNS93,LNS94,KW94,Nar96].

From an abstract point of view we can view the network as a complete graph with bi-directional links. The measure of the efficiency of an operation is the number of messages exchanged between sites [Gra88], that is, the number of times arcs in the graph are visited to execute the given operation. Every message is processed from a server in a finite time after its arrival. The data distribution and management policy determines how data are distributed among the servers; there are no preconditions as to where the data can be stored.

In the centralized case a search tree is a binary tree such that every node represents an interval of the data domain. Moreover, the overall data organization satisfies the invariant that the interval managed by a child node lies inside the parent node's interval. Hence the search process visits a child node only if the searched key is inside the parent node's interval. Kröll and Widmayer call this behavior the *straight guiding property* [KW95].

In the above-described proposals for a distributed search structure based on binary search trees, no explicit action is taken to maintain the distributed structure balanced. Hence a theoretical worst case bound of $\Omega(N)$ messages holds for the search of a given key out of a set of N keys.

Kröll and Widmayer observed [KW95] that it is not possible, in the distributed case, to directly make use of rotations for balancing a distributed search tree while guaranteeing the straight guiding property. They proved that a lower bound of $\Omega(\sqrt{N})$ holds for the height of distributed balanced search trees if the straight

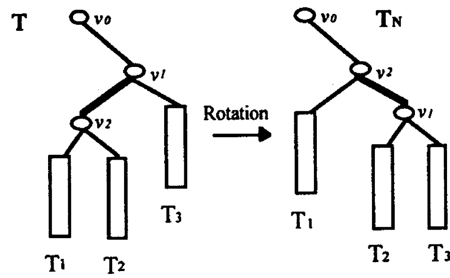

Fig. 1.

guiding property has to be satisfied. The importance of the straight guidance property lies in the fact that it guarantees the search process has an upper bound in terms of messages which is given by the height of the tree.

To understand why the straight guiding property is not automatically true in distributed search structures when rotations are used, consider Fig. 1. Assume that in the search tree **T** the server s_0 manages internal node v_0, the server $s_1 \neq s_0$ manages v_1, and the server $s_2 \neq s_1$, $s_2 \neq s_0$, manages v_2. Assume now that a rotation is needed at v_1 to rebalance the tree.

$\mathbf{T}_N$ is the tree after the rotation, where we assume the assignment of internal nodes to servers has not changed. Note that the set of keys visiting v_1 in the search tree **T** (i.e., before the rotation) is a superset of the set of keys visiting v_1 in the search tree $\mathbf{T}_N$ (i.e., after the rotation). As clients knew that an interval of the data domain containing keys in, e.g., T_1, was associated, to server s_1, it may happen that after the rotation s_1 receives the request for a key whose search path ends in T_1. Such a request from a client would therefore visit node v_1. But this is not correct since, after the rotation, server s_1 should not manage any search path for keys in T_1. The same problem would exist if we exchanged the assignment of nodes to servers between v_1 and v_2, beyond the fact that we would waste time to move keys between servers. In fact, in this case it is server s_2 that may receive the request for a key whose search path ends in T_1. Hence whether we maintain the assignment of servers s_1 and s_2 to nodes v_1 and v_2 in $\mathbf{T}_N$ or we exchange such an assignment, we fail in any case to guarantee the straight guiding property.

This example shows that it is not possible to directly make use of rotations for balancing a distributed search tree while guaranteeing the straight guiding property. In [KW95] Kröll and Widmayer proved that a lower bound of $\Omega(\sqrt{N})$ holds for the height of balanced search trees if the straight guiding property has to be satisfied. This result of Kröll and Widmayer therefore shows that the straight guiding property is too weak to obtain an efficient processing of queries in a distributed environment.

To obtain efficiency in processing search requests in distributed search trees we therefore renounce satisfying the straight guidance property. We can then apply balancing to the overall structure, to keep its height bounded by $O(\lg N)$, and by means of a slightly different data organization we are able to prove that searches and updates can be managed in $O(\lg^2 N)$. In the following we introduce a data structure called *relaxed binary search tree* by relaxing the requirement of the straight guiding property. We show that a relaxed binary search tree can be kept balanced in a distributed framework during insertions and deletions in $O(\lg^2 N)$ worst-case time and that the cost of the search process is upper bounded by $O(\lg^2 N)$.

3. How to balance distributed search trees

To prove the main results we need some preliminary definitions.

Definition 1 (Binary Search Tree). A tree T is a *binary search tree* if:

1. it is a node r, called *root* of T, together with two possibly empty substructures. If both substructures are empty, then r is called *leaf*. A nonempty substructure is a binary search tree and is called *left* (*right*) *subtree* of r. The root of the left (right) subtree of r is called the *left* (*right*) *child* of r. Node r is called the *parent node* of the root of the left (right) subtree;
2. all keys in the left (resp., right) subtree of the root are smaller (resp., greater) than the key in the root.

Definition 2 (Relaxed Binary Search Tree). A binary search tree T is *relaxed* if:

1. each node but the root of T has a pointer (*parent pointer*) to its parent node;
2. each node that is not a leaf has a pointer (*left pointer*) to a node in the left subtree and one (*right pointer*) to a node in the right subtree. If the subtree is empty, the pointer is nil.

From now on we use the term RBST to denote a relaxed binary search tree.

Definition 3 (Height of a RBST). Let T be an RBST. We define the *height* of T as the length of the longest path along parent pointers from a leaf to the root of T.

Definition 4 (Balanced RBST). Let T be an RBST. We say T is *balanced* if for each node x in T the difference between the height of the right and the left subtrees of x is not greater than one.

From the above definitions and the standard properties of balanced binary search trees [Knu73] the following result can be easily derived.

Lemma 1. *The height of a balanced RBST with N nodes is upper bounded by* $O(\lg N)$.

Searching in a balanced RBST is characterized by the following two theorems. With the term "worst-case time" we mean the worst-case number of messages exchanged in the distributed structure.

Theorem 1. *Searching for a key in a balanced RBST with N nodes requires in the worst-case* $O(\lg^2 N)$ *time*.

Proof. Let T be a balanced RBST and let x be a node of T. We first prove that it is possible to reach a child of x in $O(\lg N)$ time. In fact, assume without loss of generality that we want to reach the right child y of x. From x we can reach a node, say z, in the subtree rooted at y in $O(1)$ (by property 2 in Definition 2). Now, two cases are possible: (i) the parent of z is x, that is, $z = y$, and we are done; (ii) we reach in $O(1)$ from z its parent $z' \neq x$. We now follow from z' the chain of parent pointers until the parent of the current node is x; that is, we have reached y: from Lemma 1 in $O(\lg N)$ time we are done. Since we have used $O(\lg N)$ time to go from a node to one of its children, using again Lemma 1 we have that the overall search process in a balanced RBST costs $O(\lg^2 N)$ time. $\square$

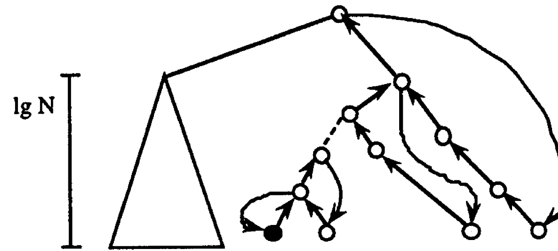

Fig. 2.

Theorem 2. *Searching for a key in a balanced RBST with N nodes requires in the worst-case* $\Omega(\lg^2 N)$ *time.*

Proof. We show in Fig. 2 a tree where the search from the root to a given leaf node (painted in black) costs $\Omega(\lg^2 N)$ time. In the tree shown both the left and the right pointers of a node point to the rightmost node in the corresponding subtree. Hence to search until to the black node requires: (i) one step, using the right pointer, from the root to the rightmost leaf; (ii) a chain of upward steps along parent pointers from the leaf to the right child of the root. Now the right child of the root is the current node and steps (iii) and (iv) below have to be repeated until the black node is reached. Step (iii) goes, using the left pointer, from the current node to the rightmost leaf in its left subtree. In step (iv) a chain of upward steps along parent pointers goes from the leaf to the left child of the current node.

The cost is 1 for step (i), $\lg N$ for step (ii), always 1 for all executions of step (iii), $\lg N - 1$ for the first execution of (iv), $\lg N - 2$ for the second one, and so on. We repeat steps (iii) and (iv) for $\lg N - 1$ times. Then the overall cost is

$$[1 + (\lg N)] + [1 + (\lg N - 1)] + [1 + (\lg N - 2)] + \cdots$$
$$+ [1 + (\lg N - \lg N + 1)] + 1$$
$$= 1 + \lg N + \sum_{i=0}^{\lg N} i = \Omega(\lg^2 N). \quad \square$$

From Theorems 1 and 2, the following corollary directly follows.

Corollary 1. *Searching for a key in a balanced RBST with N nodes requires in the worst-case* $\Theta(\lg^2 N)$ *time.*

Note that a balanced RBST is not violating the lower bound of Kröll and Widmayer, but overcomes it, since a balanced RBST assumes the search process is guided by hypotheses different from those holding for the lower bound.

4. Insertion and deletion in a balanced RBST

We now describe how to perform insertion and deletion in a balanced RBST.

Please note that when a balanced RBST is used in a distributed environment the cases of *insertion* and *deletion* that are meaningful to consider refer, respectively, to the *creation of new server* that receives part of the keys previously managed by an existing server that is now in overflow and to the *release of an existing server* that is now in underflow and sends all its keys to an existing server. By *overflow* we mean

that a server, with bucket capacity b, receives a request to insert one more key in its bucket while it is currently managing exactly b keys. By *underflow* we mean that a server, with bucket capacity b, receives a request to delete one of the $\frac{b}{2}$ keys it is currently managing in its bucket. Insertion and deletion of data items that do not cause, respectively, overflow and underflow do not require any rebalancing action and are treated in the standard way of searching for the server managing the bucket where the item has to be inserted or deleted and executing the needed actions.

Assume the creation of a new server t is triggered by the overflow of leaf node x, managed by server s. In the overall index x is then substituted by an internal node v, with two leaf nodes w and z. Keys in the bucket associated to x are split between buckets associated to w and z. New server t will manage the new internal node v and one of the two leaves, while old server s will manage the remaining leaf.

When the bucket associated to the leaf node x, managed by server s, goes in underflow and no redistribution of keys can recover the underflow, leaf x is deleted from the overall index and server s and the bucket are released. Keys that are in the bucket at this time are transferred to another bucket, but what is important from the point of view of the overall index is that also the internal node v managed by s is deleted.

Note that many different policies can be defined to improve the average server load and to decrease the overhead work associated with the transfer of data items deriving from overflows and underflows, as is done for *B*-trees and its variants or for distributed linear hashing [VBW94]. In this paper we do not consider these issues but only analyze how to update the index structure.

As a consequence of the creation of a new server or the release of an existing one, nodes are inserted into or deleted from the overall index, which can therefore become unbalanced. Our approach to balancing RBST closely follows the technique used for AVL-trees [Knu73]. We assume, as for AVL-trees, that each node x of a balanced RBST records the value of balance factor, that is, the difference between the heights of the left and the right subtrees of x. The execution of algorithms for insertion and deletion of nodes in the overall index in the message-passing distributed environment is not a big issue, since most of the actions are executed within a single server. The only delicate point is rebalancing, since it involves more servers at once. In such a case a lock mechanism is needed to ensure a correct synchronization of the operations. Then any mechanism devised for relaxed balance (i.e., uncoupling update and balance) either for concurrent AVL-trees [NSW87,LSW97] or for concurrent Red-Black-trees [NS96,BL94] can be used.

4.1. Algorithm for insertion

Step 1: *Insert*. We search for the place where the new key has to be inserted and insert it.

Step 2: *Adjust balance factors*. We move upward from the leaf just added until we reach the first node s with a balance factor different from 0. Node s is the node that may possibly need a rebalancing action. While climbing up the tree we update the balance factors of encountered nodes as it is routinely done in AVL-trees [Knu73].

Step 3: *Balance subtree*. Let k be the key just inserted, let $B(s)$ be the balance factor of node s, and let $Key(s)$ be the key in the node s. The proper rebalancing action is implemented by executing the following algorithm:

```
if k < Key(s) then
  temp ← −1
  r ← left_child(s)
```

else
  temp ← +1
  $r \leftarrow right_child(s)$
if $B(s) = 0$ or $B(s) = -temp$ then no balance is needed
if $B(s) = temp$ then
  if $B(r) = +temp$ then execute step 4 {**single** rotation}
  if $B(r) = -temp$ then execute step 5 {**double** rotation}

Step 4: *Single rotation*. Without loss of generality consider the tree in Fig. 3, where $B(s) = 1$, $B(r) = 1$, $k > Key(s)$, $temp = +1$, and the insertion is done to the subtree γ.
The single rotation is executed by means of the following operations:

1. $right_pointer(s) \leftarrow left_pointer(r)$
2. $left_pointer(r) \leftarrow s$
3. $parent(r) \leftarrow parent(s)$
4. $parent(s) \leftarrow r$
5. Find the root of β by following upward parent pointers starting at the node pointed by *right_pointer*(*s*) and stopping when the current node y is such that $parent(y) = r$.
6. Update to s the parent pointer of the root of β.

Note that nodes in the balanced RBST pointing to s need not be updated, given in property 2 of Definition 2.

Step 5: *Double rotation*. Without loss of generality consider the tree in Fig. 4, where $B(s) = 1$, $B(r) = -1$, $k > Key(s)$, $temp = +1$, and the insertion is done to the subtree γ.
The double rotation is executed by means of the following operations:

1. $right_pointer(s) \leftarrow left_pointer(x)$; $left_pointer(r) \leftarrow right_pointer(x)$
2. $left_pointer(x) \leftarrow s$; $right_pointer(x) \leftarrow r$
3. $parent(x) \leftarrow parent(s)$
4. $parent(s) \leftarrow parent(r) \leftarrow x$
5. Find the root of β (resp. of γ) by following upward parent pointers starting at the node pointed by *right_pointer*(*s*) (resp. by *left_pointer*(*r*)) and stopping when the current node y is such that $parent(y) = x$.
6. Update to s (resp. to r) the parent pointer of the root of β (resp. of γ).

Once again, note that nodes in the balanced RBST pointing to s and to r need not be updated.

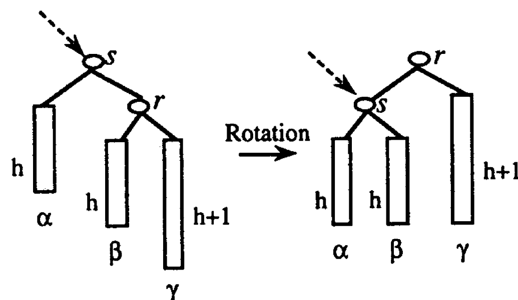

Fig. 3.

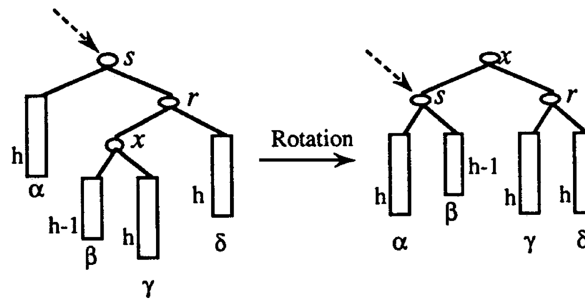

Fig. 4.

Theorem 3. *Insertion in a balanced RBST with N nodes costs in the worst-case* $O(\lg^2 N)$ *time.*

Proof. From the algorithm above we have the following costs for the various steps:

Step 1: From Theorem 1 this costs $O(\lg^2 N)$ time in the worst case.

Steps 2 *and* 3: Given Property 1 of Definition 2 and Lemma 1, this costs $O(\lg N)$ time in the worst case.

Steps 4 *and* 5: We can execute every assignment operation (1–4) in $O(1)$, we can find the root of the subtree (operation 5) in $O(\lg^2 N)$ time, and we can update its parent pointer (operation 6) in $O(1)$.

An overall $O(\lg^2 N)$ time is therefore required in the worst case. □

4.2. *Algorithm for deletion*

Let us assume that the ordering of nodes in the RBST is such that for each given node any node in its right subtree has a larger key value and any node in its left subtree a smaller one. If the node to be deleted is a leaf or an internal node whose either left subtree or right subtree is empty, then deletion is easy. The node is simply deleted and its parent is possibly rebalanced. But if the node to be deleted, say S, has both children, then, as it is done with standard AVL trees, we find the next larger node to S, say T, and since certainly the left subtree of T is empty we first substitute the key value of S with the one of T and afterward delete T. This preserves the ordering of keys.

Let then Q be a variable which points to the node of a balanced RBST that has to be deleted. We assume R points to the root of the balanced RBST and P points to the parent of Q. If Q is the root, then P is nil. We denote with *next*(Q) the next larger node to Q.

The algorithm has therefore three cases:

Case 1: Is the node to be deleted the largest in its subtree?

```
if right_pointer(Q) = nil then
  {Q is the largest node in its subtree}
  if Q = R then R ← left_pointer(Q)    {delete the root}
           else if Q = left_pointer(P) then left_pointer(P) ← left_pointer(Q)
                                            rebalance(P) if B(P) = +1
                                       else right_pointer(P) ← left_pointer(Q)
                                            rebalance(P) if B(P) = −1
```

Case 2: Is the node to be deleted the smallest in its subtree?
if $left_pointer(Q) = nil$ then
Q is the smallest node in its subtree
if $Q = R$ then $R \leftarrow right_pointer(Q)$ delete the root
else if $Q = left_pointer(P)$ then $left_pointer(P) \leftarrow right_pointer(Q)$
rebalance(P) if $B(P) = +1$
else $right_pointer(P) \leftarrow right_pointer(Q)$
rebalance(P) if $B(P) = -1$

Case 3: The node to be deleted has both children: find the next largest node and delete it.
$T \leftarrow Q$ save the original value of T in Q and find $next(T)$
$P \leftarrow Q$; $Q \leftarrow right_pointer(Q)$; $find_next(Q, P)$;
{this returns in Q the pointer to $next(T)$ and in P the father of Q}
if $Q = left_pointer(P)$
then $left_pointer(P) \leftarrow right_pointer(Q); key(T) \leftarrow key(Q)$
rebalance(P) if $B(P) = +1$
else $right_pointer(P) \leftarrow right_pointer(Q); key(T) \leftarrow key(Q)$
rebalance(P) if $B(P) = -1$

It is well known that after a deletion of an element in an AVL-tree $O(\lg N)$ rebalancing actions may be required [Knu73]. For this purpose one has to maintain, e.g., in a stack, the sequence of pointers specifying the path to the node that was finally deleted. Rebalancing actions are similar to those described in steps 4 and 5 for insertion and details can be easily filled in by the reader on the basis also of [Knu73].

Algorithm $find_next(Q, P)$ is simply:

$find_next(Q, P)$ {returns in Q the pointer to $next(T)$ and in P the father of Q}
if $left_pointer(Q) = \text{nil}$
then {Q is the smallest node in the subtree rooted at Q: we are done}
return
else $P \leftarrow Q$;
$Q \leftarrow left_pointer(Q)$;
$find_next(Q, P)$

Theorem 4. *Deletion in a balanced RBST with N nodes requires in the worst case* $O(\lg^2 N)$ *time.*

Proof. Searching for the node to be deleted costs $O(\lg^2 N)$ by Theorem 1. Each of the $O(\lg N)$ rebalancing actions that may be required can be executed in $O(\lg N)$ time, according to the description in steps 4 and 5 for the proof of Theorem 1; hence we have the claimed bound. □

5. Conclusions

We have shown in this paper that a fully dynamic and order-preserving distributed search structure, that is, a structure that maintains the relative order of data items and is able to grow and shrink as long as data items are inserted and deleted, can be implemented in a message-passing distributed environment almost as efficiently, namely with a $\Theta(\lg^2 N)$ worst-case bound, as in the single-processor case.

We have obtained this result by defining and analyzing in this paper a relaxed version of binary search trees, named RBST for relaxed binary search tree, that is suitable for an efficient management of both insertion and deletion of data items in a message-passing distributed environment. In fact, RBSTs can be kept balanced during insertion and deletion of elements almost as efficiently as standard binary search trees. Moreover, since an RBST is an order-preserving search structure, it can also support queries referring to an interval of the linear order of keys, such as range queries.

Acknowledgments

Enrico Nardelli thanks Witold Litwin, Marie-Anne Neimat, and Donovan Schneider for having introduced him to the field of scalable distributed data structures. Discussions with Brigitte Kröll and Peter Widmayer on various issues regarding the efficient management of scalable distributed data structures were very useful and provided many valuable insights. Comments from the referees were helpful. Many detailed and careful observations from one of them greatly helped in improving the quality of presentation.

References

[BL94] J. Boyar, K.S. Larsen, Efficient rebalancing of chromatic search trees, J. Comput. System Sci. 49 (1994) 667–682.

[Dev93] R. Devine, Design and implementation of DDH: a distributed dynamic hashing algorithm, in: Proceedings of the Fourth International Conference on Foundations of Data Organization and Algorithms (FODO), Chicago, 1993.

[Gra88] J. Gray, The cost of messages, in: Proceedings of the Seventh ACM Symposium on Principles of Distributed Systems, Toronto, Ontario, Canada, 1988, pp. 1–7.

[JK93] T. Johnson, P. Krishna, Lazy updates for distributed search structures, in: ACM SIGMOD International Conference on Management of Data, Washington, DC, 1993, pp. 337–346.

[Knu73] D. Knuth, in: Sorting and Searching, The Art of Computer Programming, vol. 3, Addison–Wesley, Reading, MA, 1973.

[KW94] B. Kröll, P. Widmayer, Distributing a search tree among a growing number of processors, in: ACM SIGMOD International Conference on Management of Data, Minneapolis, MN, 1994, pp. 265–276.

[KW95] B. Kröll, P. Widmayer, Balanced distributed search trees do not exist, in: Proceedings of the Fourth International Workshop on Algorithms and Data Structures (WADS'95), S. Akl et al., (Eds.), Kingston, Canada, 50–61, Lecture Notes in Computer Science, Vol. 955, Springer-Verlag, Berlin, August 1995.

[LSW97] K. Larsen, E. Soisalon-Soininen, P. Widmayer, Relaxed balance through standard rotations, in: Workshop on Algorithms and Data Structures (WADS'97), Halifax, Nova Scotia, Canada, August 1997.

[LN96] W. Litwin, M.-A. Neimat, *k*-RP*s: a high performance multi-attribute scalable data structure, in: Proceedings of the Fourth International Conference on Parallel and Distributed Information System, Miami Beach, FL, USA, December 1996, pp. 120–131.

[LNS93] W. Litwin, M.-A. Neimat, D.A. Schneider, LH*—linear hasing for distributed files, in: ACM SIGMOD International Conference on Management of Data, Washington, DC, 1993.

[LNS94] W. Litwin, M.-A. Neimat, D.A. Schneider, RP*—a family of order-preserving scalable distributed data structures, in: Proceedings of the 20th Conference on Very Large Data Bases, Santiago, Chile, 1994.

[LNS97] W. Litwin, M.-A. Neimat, D.A. Schneider, LH*—a scalable distributed data structure, *ACM* Trans. Database System 21 (4) (1996) 480–525.

[MS91] G. Matsliach, O. Shmueli, An efficient method for distributing search structures, in: Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS'91), Miami Beach, 1991.

[Nar95] E. Nardelli, Some issues on the management of k-d trees in a distributed framework, Technical Report No. 76, Dipartimento di Matematica Pura ed Applicata, Universita' di L'Aquila, January 1995.

[Nar96] E. Nardelli, Distributed k-d trees, in: XVI International Conference of the Chilean Computer Science Society (SCCC'96), Valdivia, Chile, November 1996.

[NBP97] E. Nardelli, F. Barillari, M. Pepe, Design issues in distributed searching of multidimensional data, in: Proceedings of the Third International Symposium on Programming and Systems (ISPS'97), Algiers, Algeria, Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin, April 1997.

[NBP98] E. Nardelli, F. Barillari, M. Pepe, Distributed searching of multi-dimensional data: a performance evaluation study, J. Parallel and Distrib. Comput. 49 (1998) 111–134.

[NS96] O. Nurmi, E. Soisalon-Soininen, A structure for concurrent rebalancing, Acta Inform. 33 (1996) 547–557.

[NSW87] O. Nurmi, E. Soisalon-Soininen, D. Wood, Concurrency control in database structures with relaxed balance, in: ACM Conference on Principles of Database Systems, San Diego, CA, 1987, pp. 170–176.

[SPW90] C. Severance, S. Pramanik, P. Wolberg, Distributed linear hashing and parallel projection in main memory databases, in: VLDB Conference, Barcelona, 1991.

[VBW94] R. Vingralek, Y. Breitbart, G. Weikum, Distributed file organization with scalable cost/performance, in: ACM SIGMOD International Conference on Management of Data, Minneapolis, MN, 1994.