A Simple Balanced Search Tree
with $O(1)$ Worst-Case Update Time

by

Rudolf Fleischer

ABSTRACT In this paper we show how a slight modification of $(a, b)$-trees allows us to perform member and neighbor queries in $O(\log n)$ time and updates in $O(1)$ worst-case time (once the position of the inserted or deleted key is known). Our data structure is quite natural and much simpler than previous worst-case optimal solutions. It is based on two techniques: 1) bucketing, i.e. storing an ordered list of $2 \log n$ keys in each leaf of an $(a, b)$ tree, and 2) lazy splitting, i.e. postponing necessary splits of big nodes until we have time to handle them. It can also be used as a finger tree with $O(\log^* n)$ worst-case update time.

1. Introduction

One of the most common (and most important) data structures used in efficient algorithms is the balanced search tree. Hence there exists a great variety of them in literature. Basically, they all store a set of $n$ keys such that location, insertion and deletion of keys can be accomplished in $O(\log n)$ worst-case time.

In general, updates (insertions or deletions) are done in the following way: First, locate the place in the tree where the change has to be made; second, perform the actual update; and third, rebalance the tree to guarantee that future query times are still in $O(\log n)$. The second step usually takes only $O(1)$ time, whereas steps 1 and 3 both need $O(\log n)$ time. But there are applications which do not need the first step because it is already known where the key has to be inserted or deleted in the tree. In these cases we would like to have a data structure which can do the rebalancing step as fast as the actual update, i.e. in constant time. Good worst-case behaviour is especially important in real-time applications.

One such example are dynamic planar triangulations. In [M91] Mulmuley examined (among others) point location in dynamic planar Delauney triangulations. The graph of the triangulation is stored such that at each node of the planar graph the adjacent triangles are stored (sorted in clockwise radial order) in a balanced search tree. But now, whenever a point $v$ is deleted or inserted, all points in the neighborhood of $v$ can be affected by the retriangulation because their sequence of adjacent triangles might have changed. However, these changes are only local in the sense that one triangle (which is known at that time and has not to be searched in the radial tree) must be deleted or get some new neighbors (see [M91], 3.2 for details). To guarantee a worst-case update time for the triangulated point set which is proportional to the structural change (i.e. the number of deleted or newly created triangles) one needs search trees at the nodes which can handle updates in constant worst-case time.

It has been well known for a long time that some of the standard balanced search trees can achieve $O(1)$ amortized update time once the position of the key is known.

---

1 This work was supported by the ESPRIT II program of the EC under contract No. 3075 (project ALCOM)
2 Max-Planck-Institut für Informatik, D66123 Saarbrücken, Germany, e-mail: rudolf@mpi-sb.mpg.de
But for the worst-case update time the best known method had been an \(O(\log^* n)\) algorithm by Harel \([HL],[H80]\). It has also been known that updates can be done with \(O(1)\) structural changes (e.g. rotations) but the nodes to be changed have to be searched in \(\Omega(\log n)\) time \([T83],[DSST]\). Levcopoulos and Overmars \([LO]\) have only recently come up with an algorithm achieving optimal \(O(1)\) update time (similar results had been obtained by \([DS]\) and \([vE]\)). They use the bucketing technique of \([O82]\) : Rather than storing single keys in the leaves of the search tree, each leaf (bucket) can store a list of several keys. Unfortunately, the buckets in \([LO]\) have size \(O(\log^2 n)\); so they need a 2-level hierarchy of lists to guarantee \(O(\log n)\) query time within the buckets. They show that this bucket size is sufficient if after every \(\log n\) insertions the biggest bucket is split into two halves and then the rebalancing of the search tree is distributed over the next \(\log n\) insertions (for which no split occurs).

Our paper simplifies this approach considerably: We, too, distribute the rebalancing over the next \(\log n\) insertions into the bucket which was split, but allow many buckets to be split at consecutive insertions (into different buckets). This seems fatal for internal nodes of the search tree: they may grow arbitrarily big because of postponed (but necessary) splits. But we show that internal nodes will never have more than twice the allowed number of children; hence queries can be done in \(O(\log n)\) time. Furthermore, our buckets can grow only up to size \(2\log n\), which means that we only need an ordered list to store the keys in a bucket. Also, the analysis of our algorithm seems simpler and more natural than in \([LO]\).

Since the buckets are organized as a linear list, our data structure does not allow efficient finger searches, i.e. given a pointer to some known element and a key in distance \(d\) from this element we can not locate the key in time \(O(d)\). However, iterating the construction, i.e. using our tree recursively in the buckets instead of linear lists, gives a data structure which allows efficient finger searches; but then the worst-case update time increases to \(O(\log^* n)\). This matches the best previous bounds \([HL],[H80]\).

We note that Dietz and Raman \([DR]\) recently presented a variant of the Levcopoulos/Overmars algorithm where the buckets are organized in a more complicated way such that efficient finger searches are possible with constant update time. However, their solution involves bit manipulations and table lookup and therefore works only in the RAM model, whereas all other results mentioned in this paper, as well as our result, are achieved in the pointer machine model.

The paper is organized as follows. In Section 2 we define the data structure and give the algorithms for find and insert. In Section 3 we prove their efficiency. Then we conclude with some remarks in Section 4.

## 2. The Data Structure

In this Section we will describe a simple data structure which maintains a set \(S\) of ordered keys and allows for operations query, insert and delete. Queries are the so-called neighbor queries: given a key \(K\), if it is in the current set \(S\) report it, otherwise, report one of the two neighbors in \(S\) according to the given order. Insert and delete assume that we have previously located the key (to be deleted) or one of its neighbors (if we insert a new key) in the data structure. As was illustrated by the triangulation example in Section 1 where we have two nested data structures, this does not necessarily mean that we must perform a query in our data structure to locate this key. Our data structure is basically a balanced search tree, a variant of an \((a,b)\)-tree (\(4 \leq 2a \leq b\) and \(b\) even).

The main problem with update operations in a balanced search tree (and all other