Journal of Statistical Planning and Inference 140 (2010) 2335–2345

Contents lists available at ScienceDirect

Journal of Statistical Planning and Inference

journal homepage: www.elsevier.com/locate/jspi

Deletions in random binary search trees: A story of errors ☆

Wolfgang Panny

Applied Computer Science, WU-Wien, Augasse 2–6, A-1190 Vienna, Austria

ARTICLE INFO

Available online 21 January 2010

Keywords:
Random binary search tree
Analysis of algorithms
Deletions

ABSTRACT

The usual assumptions for the average case analysis of binary search trees (BSTs) are random insertions and random deletions. If a BST is built by n random insertions the expected number of key comparisons necessary to access a node is $2\ln n+O(1)$. This well-known result is already contained in the first papers on such 'random' BSTs. However, if random insertions are intermixed with random deletions the analysis of the resulting BST seems to become more intricate. At least this is the impression one gets from the related publications since 1962, and it is quite appropriate to speak of a story of errors in this context, as will be seen in the present survey paper, giving an overview on this story.

1. Introduction and prerequisites

Binary search trees (BSTs) are among the most prominent and commonly used data structures for symbol table algorithms (Knuth, 1998, p. 426). The usual search- and insertion algorithms for BSTs (cf. e.g., Knuth, 1998, p. 429) are quite efficient for this purpose. BSTs are especially suitable for applications where (apart from accessing and dynamically inserting the symbols into the table) it is also required to linearly process them according to their sort sequence, e.g., to print a sorted list of the symbols, say. The first publications on BSTs are due to Windley (1960), Booth and Colin (1960), and Hibbard (1962). Each of these papers comprises among other things a description of binary tree insertion and the expected number of key comparisons thereby incurred.[1]

Hibbard (1962) first showed how to realize deletions from a BST in a reasonable way, thereby considerably extending the application range of BSTs. The BST structure is not restricted to access the nodes 'by key'. They alternatively can be retrieved 'by rank', which only requires straightforward modifications. In this way BSTs can be extended to a versatile and efficient data organization for the representation and manipulation of linear lists (Knuth, 1998, pp. 471–475). Other refinements aim at protecting against the $O(n)$ worst case behavior of the original structure. This is achieved by imposing additional constraints on the shape of the trees to insure an access time of $O(\log n)$ even in the worst case. The most prominent species of such balanced search trees is due to Adelson-Velsky and Landis (1962), who devised their scheme as early as 1962. Essentially the same purpose can also be achieved by randomization. To this approach belong Seidel and

☆ A preliminary German version of this paper has appeared in Geyer-Schulz and Taudes (2003, pp. 75–88). However, the present version has been enlarged and reworked to shed more light on the technical aspects.

E-mail address: wolfgang.panny@wu-wien.ac.at

[1] Additionally, *Windley*'s paper contains a comprehensive discussion of tree insertion sorting, *Booth* and *Colin* consider the effect of arranging the first 2^i-1 elements to form a complete binary tree. As predecessors one should also mention A.I. Dumey, D.J. Wheeler, and C.M. Berners-Lee (cf. Knuth, 1998, p. 453; Douglas, 1959, p. 5; Windley, 1960, p. 84).

0378-3758/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.jspi.2010.01.028

Aragon (1996) 'treaps', which in fact are a reinvention of Vuillemin (1980) 'cartesian trees'. Other proponents of this approach are Martinez and Roura (1998) with their 'randomized binary search trees'.

In this paper only the original BST structure with access functions *search*, *insert* and *delete* will be considered. The usual (and reasonable) assumption for the average case analysis of binary search trees are *random insertions*. In a BST built by n random insertions the expected number of key comparisons necessary to access a node is $2\ln n+O(1)$, which is a well-known result already contained in the first papers on BSTs (Windley, 1960; Booth and Colin, 1960; Hibbard, 1962). However, if random insertions are intermixed with *random deletions* the analysis of the resulting BST seems to become much more intricate and involved. At least this is the impression one gets from the publications on the subject since 1962, and it is quite appropriate to speak of a story of errors in this context. In this survey we shall take a closer look at this story, which will be done in Section 2. In the remainder of the present section some conceptual and notational prerequisites are compiled.

1.1. Binary search trees and access functions

A *binary tree* can be defined as a finite set of nodes that is either empty, or consists of a root and the elements of two disjoint binary trees called the left and right subtrees of the root (Knuth, 1997, p. 312).

It is clear from this definition that a non-empty binary tree always has at least one empty subtree. Sometimes it is convenient to consider these empty subtrees as additional (special) nodes, referred to as *external* nodes. A binary tree with explicit external nodes is called an *extended* binary tree (Knuth, 1997, p. 399). In this case the 'ordinary' nodes are called *internal* nodes. A binary tree with n (internal) nodes has $n+1$ external nodes. In an extended binary tree the external nodes coincide with the leaves (Fig. 1).

In a *binary search tree* (BST) a key is attached to every node such that the following condition holds: For every node the keys in its left (right) subtree are smaller (greater) than the key attached to the node.

Implementation of the *search* operation follows immediately from the above search condition. If the BST contains a matching key, the search is *successful*. Otherwise it is an *unsuccessful* search (Fig. 2).

Implementation of the *insertion* operation is straight forward, too: The actual insertion is preceded by a search for the key to be inserted. Since multiple keys are not allowed, the search should be unsuccessful. Hence it eventually terminates at an empty subtree. As final step the corresponding external node is replaced by the new node (with empty left and right subtrees).

The *delete* operation is a bit more complicated. To describe it more easily the following notation will be used: Let T denote a BST, and let $v \in T$ be a node of T. Then the left and right subtree of v will be denoted by $L(v)$ and $R(v)$. The key of node v is denoted by $k(v)$ and the references to the root nodes of $L(v)$, $R(v)$ are dentoted by $\ell(v)$ and $r(v)$, respectively. The father of v is symbolized by $f(v)$ (to simplify matters it will be assumed that also the root of T has a father w such that $L(w)=T$). The actual deletion must be preceded by a search to localize the node to be deleted. This search must be

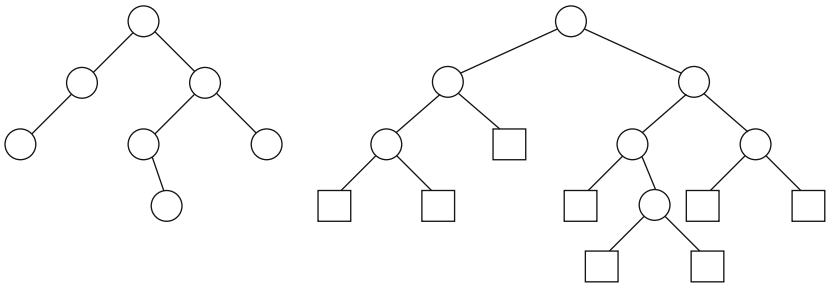

Fig. 1. Binary tree and corresponding extended binary tree.

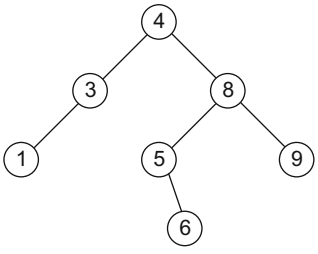

Fig. 2. Binary search tree.