The Hidden Binary Search Tree:

A Balanced Rotation-Free Search Tree in the AVL RAM Model

Saulo Queiroz
Email: sauloqueiroz@utfpr.edu.br

Academic Department of Informatics
Federal University of Technology (UTFPR)
Ponta Grossa, Brazil

November 21, 2017

Abstract

In this paper we generalize the definition of "Search Trees" (ST) to enable reference values other than the key of prior inserted nodes. The idea builds on the assumption an n-node AVL (or Red-Black) relies on to assure $O(\log_2 n)$ worst-case search time, namely, a single comparison between two keys takes constant time. This means the size of each key in bits is fixed to $B = c \log_2 n$ ($c \geq 1$) once n is determined, otherwise the $O(1)$-time comparison assumption does not hold. Based on this we calculate *ideal* reference values from the mid-point of the interval $0..2^B - 1$. This idea follows 'recursively' to assure each node along the search path is provided a reference value that guarantees an overall logarithmic time. Because the search tree property works only when keys are compared to reference values and these values are calculated only during searches, we term the data structure as the Hidden Binary Search Tree (HBST). We show elementary functions to maintain the HSBT height $O(B) = O(\log_2 n)$. This result requires no special order on the input – as does BST – nor self-balancing procedures, as do AVL and Red-Black.

1 Introduction

A Search Tree (ST) evolves upon insertions/deletions according to the nodes' key field value. Such field works as a kind of *reference* to help placing a new unique value k in the tree rooted by the key value k_r. For a Binary Search Tree (BST), k is placed to the left or right subtrees of k_r if $k < k_r$ and $k > k_r$, respectively. *This definition forces the reference value to always be the key field of previously inserted nodes.* Hence, if a sequence consisted of n 'bad' keys is given as input, the resulting worst-case search performance is linear, missing the opportunity to build a $O(\log_2 n)$-height tree.

1

A classical way to preventing BST to unbalance consists in walking the insertion/deletion path back to the root to check/update height-related informations and trigger node rotation(s) if necessary, the so-called self-balanced BSTs e.g. AVL [1], "Red-Black" [4]. Those tasks increase programming complexity in comparison to BSTs. Also, they might cause the self-balanced BSTs to perform worse than a common BST when the sequence of insertion keys leads the latter to be "naturally" balanced. Indeed, Knuth [5] shows that BSTs requires only $2 \ln n \approx 1.386 \log_2 n$ comparisons if keys are inserted in a random order.For this dilemma, he suggests a 'balanced attitude' considering self-balanced trees for large n – due to the BST's 'annoying (linear) possibility' – and BSTs for lower n because of its reduced overhead and simpler programming.

In this paper we wonder about the condition(s) under which (if any) it is possible to design a rotation-free BST with $O(\log_2 n)$-height regardless of the n-size sequence of insertion keys. To achieve that we generalize the definition of "Search Tree" by enabling *reference* values other than the key values of prior inserted nodes. We refer to them as *hidden reference values* because *they guide search procedures but requires no kind of permanent registration*. To derive the proposed's tree worst-case search, insertion and deletion time complexities we consider the Random-Access Machine (RAM) model [2] under the same assumptions taken by AVL and Red-Black. With that, after the maximum number n of keys in the tree is determined, the maximum number of bits B to represent a key is computed as a constant bounded to $\lceil \log_2(n) \rceil$. Otherwise, if n is not known in advance, B can not be assumed as constant. Thus, the constant time assumed for comparison between keys "clearly becomes an unrealistic scenario" [3] and the $O(\log_2 n)$ complexity no more hold for the self-balanced BSTs.

Given an input key, we rely on B to calculate hidden reference values for each node over the search path. These values correspond to the *ideal* sequence of insertion keys to build a balanced BST with at most n nodes. Because new incoming keys are placed in the tree based on those reference values – rather than on the values of prior inserted keys– the "search tree" property does not hold. However, the resulting tree can be viewed as a "search tree" in the sense that the *hidden reference value* of an arbitrary node is always greater (less) than any *key value* in its own left (right) subtree[1]. For this reason we term this data structure as the Hidden Binary Search Tree (HBST). We present elementary algorithms to maintain the HBST's height bounded to $O(B) = O(\log_2 n)$. The algorithms assume neither special order keys nor any kind self-balancing rotation procedure.

The reminder of this paper is organized as follows. In Section 2 we discuss the basic idea behind the HBST. In Section 3 and 4 we present the elementary functions and their worst-case complexities, respectively. Finally in Section 5 we summarize this work and discuss future directions.

---

[1]The insertion case where the input key is equal to the hidden reference value is a matter of design choice.

2 The Reference Hidden BST Algorithm

Let n be the number of nodes of a BST, each of which uniquely identified by a key k from the integer interval $[0, n-1]$[2]. In several practical scenarios, n is determined in advance when the data structure programmer determines a data type with B bits for the nodes' key field. From the interval $[0, 2^B]$, one can build a balanced BST following an idea reminiscent to Merge sort algorithm. This resulting BST is illustrated in Fig. 1 for $B = 4$. Firstly, the algorithm takes the interval $[0, 2^B]$ as input and choose its mid-point $k_m = \lfloor (0 + 2^B)/2 \rfloor$ to insert in the BST. The same idea applies recursively to the root's left and right subtrees with the intervals $[a, k_m]$ and $[k_m, b]$, respectively. Note that $a = 0$ and $b = 2^B$ in the first iteration.

Figure 1: BST recursively built from mid-point intervals. Starting point $[1, 15]$.
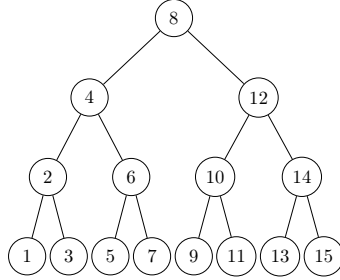
The idea just described may not seem to be valuable because the insertion sequence is not known *a priori*. However, one can benefit from it if the "search tree" property can be relaxed (actually generalized) to include reference values other than prior inserted keys. These reference values need not to be stored in nodes. They can be computed, for example, taking as reference an ideal insertion sequence to guide which subtree the search must follow in each iteration (or recursive call). Then, for a given reference search value, all nodes at its left subtree have value less than it whereas all nodes in the right subtree have values greater than it. The insertion case where the input key is equal to the hidden reference value is a matter of design choice. A binary tree that satisfy the search property this way we name as Hidden Binary Search Tree (HBST)

An HBST built from the insertion sequence $0, 1, 2, \ldots, 15$ with $B = 4$ is illustrated in Fig. 2 where values equal to the reference value are insert to the right. The first insertion is the trivial case. After that, the second input consists of the key $k = 1$ along with the interval[3] $[0, 2^4[$. The algorithm check that there is a node in the current level (the root, in this case), and calculates the hidden search reference value k_m (shown in the center of the node's interval) from the given interval doing $\lfloor (0+16)/2 \rfloor$. The same idea applies recursively to the root's left and right subtrees with the intervals $[a, k_m[$ and $[k_m, 2^B[$, respectively. Note

---

[2] $k \in [0, n-1] = \{k \in \mathbb{N} | 0 \leq k \leq n-1\}$.
[3] $k \in [0, n[= \{k \in \mathbb{N} | 0 \leq k < n\}$.

that $a = 0$ in the first iteration and the interval signs are merely illustrative.

Figure 2: HBST built from the insertion sequence $0, 1, 2, 3, 4, \ldots, 15$. $B = 4$. The upper and lower bounds interval values are passed from one iteration to another; their mid-point is the hidden search reference value. The interval signs are merely illustrative.
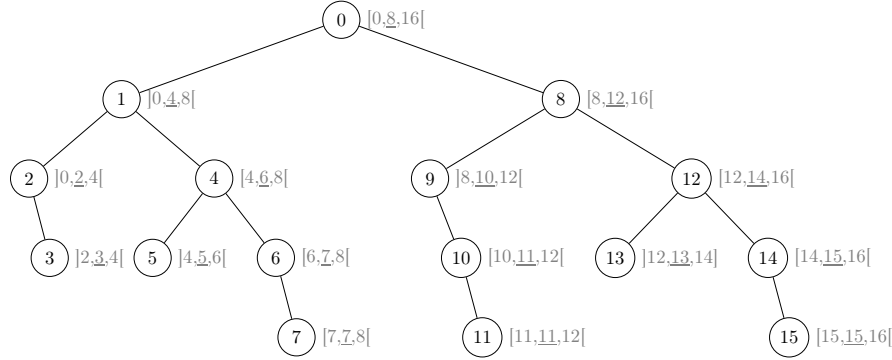
2.1 HBST's Search Property

In BST or variants thereof, the ST property is always checked considering the *same field* of different nodes, usually the key field. That said, it is clear the ST property does not hold in HBST, as one can easily see in Fig. 2. However, the hidden reference tree associated to the reference values of the interval $0 \ldots 2^B - 1$, does satisfy the property. Besides that, and most important, if h_{ref} is found to be the root's hidden reference value in the HBST (sub)tree T_H, then *the HBST's insertion rule for an arbitrary key k mandates that k must be inserted to the left subtree of T_H if $k < h_{ref}$ or to the right, otherwise.*

3 HBST: First Elementary Functions in C

In this Section we present the first elementary functions `insert`, `search` and `lazyDel` for inserting, searching and deleting a given input key in the HBST. The deleting function employs an lazy strategy: node is only removed logically (key field assigned to flag `-1`) such that the space can be reused later by the insertion function. A 'hard deletion' function works just like in standard BSP unless the node has two children. In HBST there is no need to find the minimum from right subtree nor maximum from left subtree: the substitute can be any descendant node with less than 2 children. We choose mnemonic name for the nodes' fields just as in a typical BST. The remainder set of assumptions for them are embedded as comment in the code itself.

```
/* Assumptions:
 *   unique key values, HBSTNode typical BST structure, r is
```

```
 *  valid ref pointer to the root, alocateNewNode is
 *  a function to allocate and connect new node.
 *  keys are signed 32-bit integer (negatives discarded)
 *  i.e. B = 32, n<=2^30. First call: min=0, max=2^30, in C: 1<<30.
 */
#include <stdlib.h>
HBSTNode *insert(HBSTNode **r, int newKey, unsigned int min, unsigned int max)
{
  if (*r==NULL)
    //allocate new node as in a BST. Return pointer to it or NULL
    return alocateNewNode(r, newKey);

  // OPTIONAL: make use of space released by function lazyDel
  // for simplicity we assume newKey is not currently in the tree.
  if ((*r)->key == -1)
   { (*r)->key == newKey; return *r; }

  unsigned int hiddenRef = (min + max)/2;
  if (newkey < hiddenRef)
    return insert(&((*r)->left), newKey, min, hiddenRef);
  else
    return insert(&((*r)->right), newKey, hiddenRef, max);
}


HBSTNode *search(HBSTNode *r, int key, unsigned int min, unsigned int max)
{
  if (r == NULL || r->key == -1 || min > max) return NULL;
  if (r->key == key)
    return r;
  unsigned int hiddenRef = (min + max)/2;
  if (key < hiddenRef)
    return search(r->left, key, min,hiddenRef);
  else
    return search(r->right, key, hiddenRef, max);
}

/*
 * Assume dynamic environment. Avoid expensive
 * memory management. Employ mem reuse.
 */
HBSTNode *lazyDel(HBSTNode *r, int key, unsigned int min, unsigned int max)
{
  HBSTNode *killMe = search(r, key, min, max);
  if (killMe == NULL || r->key == -1) return NULL;
  killMe->key = -1;
```

```
    return killMe;
}
```

4 Complexity

The worst-case order of growth $T(n)$ for the HBST performance is dominated by the search strategy common to all functions presented in Section 3. $T(n)$ can be readily obtained by the recurrence equation (1), where n is the input parameter `max`. Recall that we are assuming the RAM model [2] in which the word size can not grow arbitrarily after n is chosen [3]. This is the same assumption under which AVL and Red-Black get running time is $O(1) \times O(\log_2 n)$, where the first term represent the constant time to perform a comparison between integers and the second the height of tree. As in binary search, AVL and Red-Black, each round of HBST function solves approximately half the input size at a $O(1)$ time cost. Once the number of bits B is assigned to the key field, maximum asymptotic height of HBST is logarithmic on the input (Eq. 2) or, alternatively, linear on B (Eq. 3). This performance requires no kind of balancing procedure nor "good" insertion sequences.

$$
\begin{aligned}
T(n) &= c + T(n/2), \text{ if } n > 1 & (1) \\
T(n) &= O(\log_2 n) & (2) \\
T(n) &= O(\log_2 2^B) & \\
T(n) &= O(B) & (3)
\end{aligned}
$$

4.1 Practical Considerations

The hidden tree underlying an HBST is nothing but a balanced tree composed of all values from $0..n$, where $n = 2^B - 1$. Since the resulting tree is balanced, its height is $O(\log_2 n)$. This reveals the three worst-case is bounded to $O(\log_2 2^B) = O(B)$. Considering a practical example in which the key field is declared as a 64-bit integer, the data structure supports no more than $2^{64} - 1 \approx 10^{19}$ distinct keys and a comparison between two keys takes $O(64) = O(1)$. For any quantity $n' < 2^{64}$, a balanced BST has its complexity bounded to $O(\log_2 n')$ while HBST is bounded to $O(B)$, i.e. ≈ 65 iterations to find/insert/delete a key in this case. Thus, HBST's performance may degrade if, for example, B grows on demand i.e. the value of the (really) largest key can not be estimated in advance. In this case, a single comparison takes $\omega(1)$ time and some kind of technology to increase B on-demand is required.

5 Conclusion and Future Work

In this work we showed that it is possible to relax the definition of search tree while keeping almost unchanged the main elementary functions of a typical

Binary Search Tree (BST). We achieved that by generalizing the "search tree" property allowing it to considering values other than the key field of prior inserted nodes. This concept based the design of the "Hidden Binary Search Tree (HBST)", a balanced rotation-free tree data structure. To successfully build a search path, HBST compares the input key against "hidden" reference values of a reference balanced BST ideally built on the interval $0 \ldots 2^B - 1$, where B is the size of nodes' keys in bits and 2^B is the size of input, i.e. size n of insertion keys.

We presented search, insertion and deletion algorithms that keep the HBST's height bounded to $O(B)$. Since B is dimensioned according to the insertion sequence size n in such a way that $B = O(\log_2 n)$, HBST achieves logarithmic worst-case running time under the assumption that B is fixed once n is given. This is the same assumption under which AVL and Red-Black BSTs achieve $O(\log_2 n)$ worst-case time. In fact, as pointed out by [3], if B can grow arbitrarily, a 'simple' $O(1)$ key comparison (as assumed by AVL and Red-Black) becomes unrealistic, preventing those trees' complexities to be solely explained by $O(\log_2 n)$. Under this same assumption, HBST achieves $O(B) = O(\log_2 n)$ time with no need to perform any kind of balancing/rotation (as required by AVL and Red-Black) nor to assume special order on the input, as required by BST to achieve logarithmic performance.

An important question left open in this work is about the feasibility of a linear time *in-order* traversal on HBST. Regarding the presented functions, lots of interesting refinements can be performed such as adaptive B according to the given key value, top-down insertion and hybrid BST-HBST structures. Finally, it would be interesting to check whether the "hidden search" property can improve the performance of other kind of structures such as external data structures (e.g. B-tree) and priority queues.

6   Acknowledgements

References

[1] G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

[2] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 73–80, New York, NY, USA, 1972. ACM.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[4] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, Oct 1978.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.