The Hidden Binary Search Tree

Saulo Queiroz[1], **Edimar Bauer**[1]

[1]Academic Department of Informatics
Federal University of Technology (UTFPR)
Ponta Grossa, PR – Brazil

sauloqueiroz@utfpr.edu.br, edimarbauer@alunos.utfpr.edu.br

Abstract. *In this paper we review and enhance the Hidden Binary Search Tree (HBST) presented in [Queiroz 2017]. The HBST idea builds on the assumption an n-node self-balanced tree (e.g. AVL) requires to assure $O(\log_2 n)$ worst-case search, namely, comparison between keys takes constant time. Therefore the size of each key in bits is fixed to $B = O(\log_2 n)$ once n is determined, otherwise the $O(1)$-time comparison assumption does not hold. HBST generalizes the search-tree property such that the position of a node in the tree results from comparing its key against 'ideal' reference values associated to its ancestors. The first ideal values comes from the mid-point of the interval $0..2^B$. The strategy follows recursively such that the HBST height is bounded by $O(B)$ regardless the input sequence of keys nor self-balancing procedures. In this paper we enhance the HBST to enable keys with arbitrary number of bits.*

1. Introduction

In a Binary Search Tree (BST) and variants thereof the 'search-tree' property relies on a field of the nodes' abstract data type to determine the position of a given key in the tree. That definition causes BST's height $h(n)$ to vary depending on the values of the given n-size input sequence of integer keys s=<k_i> ($0 \leq k_i < n$ and $1 \leq i \leq n$) such that $h(n) = O(n)$ in the worst case. Self-balanced BSTs (e.g. AVL, Red-black) decouple the $h(n)$ performance from s by means of self-balancing procedures. These procedures may walk the path back from an inserted (deleted) node to the root in order to update height-related informations and, if needed, trigger rotation(s) to ensure $h(n) = O(\log_2 n)$. Therefore, self-balanced BSTs are preferred than BSTs when the asymptotic worst-case scenario is the sole criterion. However, if n is not large enough or s 'naturally' causes BST's height to be logarithmic – as is the case of random uniform inputs – BSTs may be preferred because of its "*substantially less overhead and simpler programming*" [Knuth 1998].

[Queiroz 2017] presents the Hidden Binary Search Tree (HBST), a data structure characterized by a logarithmic worst-case height and the 'simpler programming' of BSTs (i.e. no self-balancing procedures). The first remark of the HBST design is that the $O(\log_2 n)$ search of self-balanced BSTs results from assuming that a comparison between keys takes constant time. This implies the number of bits B per key must be a constant bounded to $O(\log_2 n)$ after n is determined. From this, the author generalizes the search-tree property such that the position of a given key in the tree can result from comparisons against *reference* values other than the key of prior inserted nodes. Relying on the assumption that B is constant, HBST matches its search reference values to the sequence of keys that 'naturally' causes a BST to be balanced. In this work we review the HBST procedures (Section 2) and enhance its design to enable unbounded B (Section 3).

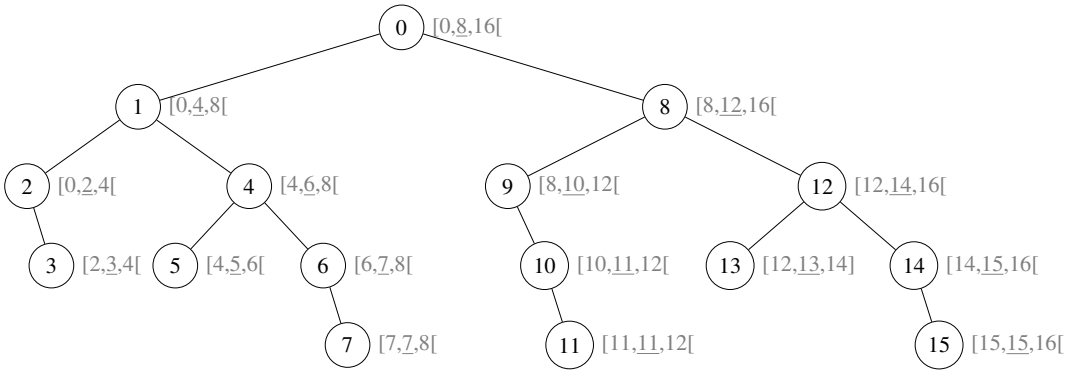

Figura 1. HBST built from the sequence $0, \ldots, 15$, $B = 4$. The upper and lower bounds intervals are computed across iterations. Search-property holds if a key is compared against the ideal hidden reference values of its ancestors (underlined values).

2. Elementary Procedures of HBST

The elementary HBST procedures exploits a widely known lesson from the design of algorithms, namely, an algorithm whose iteration (or recursive call) takes $O(1)$ time to halve an n-size problem has $O(\log_2 n)$ complexity. In several practical scenarios, the largest size n of a data structure is implicitly established when the programmer declares a B-bit key field, i.e. $n = 2^B$. From these points, HBST considers the lower and upper bounds of the interval $0..2^B$ to guide the insertion, search and deletion of a given input key $0 \leq k_i < 2^B$, as we describe next.

Insertion and Search. If the given (sub-)tree is empty, k_i is inserted as root. Otherwise the search reference value "idealRef" is set to $\lfloor (lower + upper)/2 \rfloor$. The variables $lower$ and $upper$ are respectively set to 0 and 2^B in the first iteration. If $k_i < idealRef$ the insertion follows recursively to the left subtree setting $upper$ to "idealRef". Otherwise it follows to the right subtree updating $lower$ to "idealRef". The interval $[lower, upper]$ halves at a cost $O(1)$ per iteration, leading to $O(B)$ iterations in the worst-case. An HBST built from the insertion sequence $0, 1, 2, \ldots, 15$ with $B = 4$ is illustrated in Fig. 1. Note that the search-tree property holds only if the keys at the left and right subtrees of an arbitrary node r are compared against the hidden (underlined) reference value associated to r. The same idea applies to find a key in the tree as illustrated in the recursive Algorithm 1.

Deletion. The first part of deleting a node r works just as in BST following the hidden search strategy to find it (Algorithm 1). If r is leaf, it is removed (of course the pointer to it is updated accordingly in the corresponding parent node). If r is non-leaf, it can be overwritten by one of its descendants leaf node[1].

3. Practical Concerns and HBST Enhanced Design

The careful reader may observe that HBST is nothing but a typical BST that exploits the constant number of bits of the key field to present an alternative search-tree property ensuring $O(B)$ height. For any quantity $n' \leq n$, HBST *worst-case* has $B + 1$ levels (i.e. $0, 1, ..., B$) while the height of AVL and Red-Black

[1] this fixes the statement "*the substitute can be any ... less than 2 children*" of [Queiroz 2017].

Algorithm 1: HBST Search.

```
1  Function Search (root, key, lower, upper)
2  # Assumptions: B-bit keys. Possible key values: 0, 1, ..2^B − 1.
3  # First call: lower=0, upper=n=2^B.
4  if root = nill OR root → key = key then
5  |  return root;
6  end
7  idealRef := ⌊(lower + upper)/2⌋;
8  if key < idealRef then
9  |  return Search (root → left, key, lower, idealRef);
10 else
11 |  return Search (root → right, key, idealRef, upper);
12 end
```

are $1.4405 \log_2(n' + 2) - 0.3277$ [Knuth 1998, Adelson-Velsky and Landis 1962] and $2 \log_2(n' + 1)$ [Guibas and Sedgewick 1978], respectively. For relatively small n', the *height* of self-balanced BSTs can outperform HBST's. However, the worst-case of insertion and deletion procedures might be added by rotations of those trees. Moreover, in the worst-case the path from the root to the node being inserted/removed is traversed twice (irrespective of rotations) to assure that balance factors (colors) of the ancestors are updated. Thus, the worst-case of insertion/deletion in self-balanced BSTs is at least twice their respective maximum heights plus rotations. Since HBST does not need self-balancing procedures nor rotations its actual performance compares to self-balanced BSTs. At the same time, its programming 'simplicity' and maintainability compare to the classical BST. We believe both these characteristics turn HBST very appealing for practical scenarios. Next we describe an enhanced HSBT design to handle keys with arbitrary number of bits.

The enhanced HBST data structure consists of a sequence of HSBTs as illustrated in Fig. 2. Given a non-negative integer key k_i for insertion, deletion or search, we firstly count the *minimum* number of bits $B(k_i)$ needed to represent k_i in binary. This can be done at a cost $O(B(k_i))$ by successively dividing k_i by two while the result is not zero. In practice, these divisions can be efficiently implemented by successive right shifts. We insert the number $B(k_i)$ in a sorted linked list. Each node of the list represents (i.e., has a pointer to) a specific HBST where all keys can be represented with the same minimum number of bits $B(k_i)$. For instance, keys 2 and 3 are inserted in the same HBST because $B(2) = B(3) = 2$ bits. Thus, $B(k_i)$ indicates which HBST k_i must be inserted into. If $B(k_i) = 1$ the hidden interval associated to the root node is $[0, 1]$. If $B(k_i) > 1$ the corresponding HBST interval is $[2^{B(k_i)-1}, 2^{B(k_i)}[$. Hence, the maximum number of nodes in the HBST of k_i is $n_{B(k_i)} = 2^{B(k_i)} - 2^{B(k_i)-1} = 2^{B(k_i)-1}$ (please, remark the base case $n_1 = 2$ keys) which yields a height of $O(\log_2(2^{B(k_i)-1})) = O(B(k_i))$.

The worst-case time for insertions in the sorted linked list of Fig. 2 is linear on the number of bits $B(k_i)$ of the key k_i. To make the HBST linked list to grow linearly on the number of nodes, one needs an input sequence in which each key is twice higher than its preceding key e.g. $s=<2^1, 2^2, 2^3, \ldots, 2^n>$. In this extreme case, there are n unitary

HBSTs and the largest key needs n bits. Thus an insertion takes $O(n) = O(B_{max})$ where B_{max} is the largest key in number of bits. In the opposite case where all n keys are in a single HBST, the size of the linked list is one (keys have the same size in bits) and the height of the tree is $O(B_{max})$. Accounting for the fact that a single key comparison in an arbitrary level of the tree is $O(B_{max})$ rather than $O(1)$, the overall search complexity is $O(B_{max}^2)$.

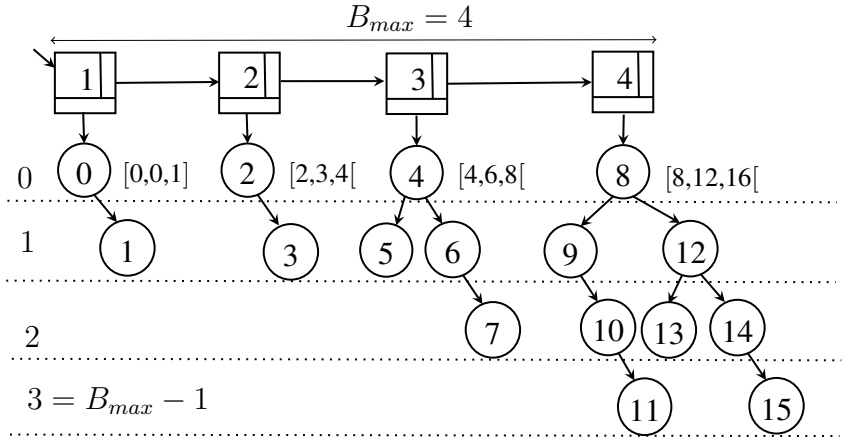

Figura 2. HSBT enhanced for keys with arbitrarily number of bits B. built from the sequence $0, \ldots, 15$. The first hidden interval for HSBTs with 1-bit and $B(k_i)$-bit keys are $[0, 1]$ and $[2^{B(k_i)-1}, 2^{B(k_i)}[$, respectively.

4. Summary

We reviewed and enhanced a variation of the Binary Search Tree (BST), named Hidden BST (HBST). Under the same assumptions of self-balanced BSTs (e.g. AVL, red-black), HBST's height is $O(B)$ where B is the size of keys in bits and n is a given number of keys. For keys with arbitrary number of bits, we show elementary $O(B_{max}^2)$ procedures where B_{max} is the currently known largest key in bits. Possible related topics for future work are linear-time in-order traversal, priority queues and external memory.

Referências

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

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

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

Queiroz, S. (2017). The hidden binary search tree: A balanced rotation-free search tree in the AVL RAM model. *CoRR*, abs/1711.07746.