An Immediate Approach to Balancing Nodes of Binary Search Trees

Chung-Chih Li

Dept. of Computer Science, Lamar University
Beaumont, Texas,USA

Abstract

We present an immediate approach in hoping to bridge the gap between the difficulties of learning ordinary binary search trees and their height-balanced variants in the typical data structures class. Instead of balancing the heights of the two children, we balance the number of their nodes. The concepts and programming techniques required for the node-balanced tree are much easier than the AVL tree or any other height-balanced alternatives. We also provide a very easy analysis showing that the height of a node-balanced tree is bounded by $c \log n$ with $c \approx 1$. Pedagogically, as simplicity is our main concern, our approach is worthwhile to be introduced to smooth student's learning process. Moreover, our experimental results show that node-balanced binary search trees may also be considered for practical uses under some reasonable conditions.

1 Introduction

In a typical Data Structures syllabus for undergraduates (CS3), the binary search tree seems to be the first real challenge for students, which is introduced after students have been familiar with linked lists and had preliminary understanding of recursion. Then, about two weeks (equivalently, six class hours) are needed for the general concepts of binary search trees including basic data structures, searching, traversals, insertion/delection, and some preliminary analysis. It certainly is not easy to manage all these concepts and necessary programming techniques in six hours of lecturing, especially when students after CS1 and CS2 are not yet well-trained to manage complicate algorithms under today's programming paradigm for beginners. Unfortunately, this is not yet enough. It makes only little sense to learn binary search trees without knowing how to balance them given the fact that a completely random data set is just a strongly theoretical assumption. There are many alternatives to the ordinary binary search tree. Many textbooks (from old textbooks [10, 6] to more recent ones like [5, 9]) introduce the AVL tree[1] as a supplement to binary search trees. This indeed is a natural choice as students realize that the advantage of binary search trees over linked lists disappears when the height grows out of control.

To keep a binary search tree balanced, there are two problems to solve: (i) How to determine which node is out of balance? and (ii) how to reconstruct the tree when needed? The standard solution suggested by the AVL tree is that: using the four rotations (RR, LL, RL, and LR) in accordance with the balance-factors (the difference between the heights of a node's lift-child and right-child) in the path of insertion/deletion. While the standard solution is not too difficult for an average student to understand, the implementation is involved. However, detailed implementations are often omitted from most textbooks or by many instructors due to the time constraints. Students are on their own to program their AVL trees. This may be reasonable because, assumably, students should have sufficient programming skill learned from CS1 and CS2 to handle the details. But this may not be the case in reality. Pedagogically, we should be aware of the potential difficulties students may face. In their implementations, students may use very inefficient code to determine the balance factor, e.g., recursively compute the heights of the two children, which will force the program to travel the entire tree. Students may introduce back-links in order to travel back to the parent for determining the type of rotation to perform, but the back-links will make it more difficult to have correct code for rotations. Deletion a node from an AVL tree is even worse. Many textbooks simply omit this important operation from their texts, i.e., [9].

[1] The AVL tree is a classic method to construct height-balanced binary search trees, which is the most often discussed method in the classroom. There are certainly many other alternatives, e.g., the Red-Black tree. Our arguments can as well apply to those alternatives.

Apart from the implementations, to help students appreciate binary search trees and AVL trees, some analyses are necessary; but they are not straightforward neither. Many analyses are out of the scope of the course. For example, the expected height of a random binary search tree is asymptotically in $O(\log n)$. The proof requires some nontrivial mathematical background which is more suitable for the algorithm analysis class (see [2] for the detailed proof). Moreover, the exact bound of the expected height of a random binary search tree is much more complicate, which has been an active research topic with some fundamental questions still open (e.g., see [4, 7]).

With these frustrations, as we have observed, balancing binary search trees is unfortunately the turning point students begin dropping the course and change their majors to those that demand less technical contents. This usually happens in the sixth week[2] of the semester when students are just about to begin to appreciate linked lists and binary search trees.[3] To smooth this learning difficulty in the classroom, an easier way to adjust unbalanced binary search trees is needed. Two papers with this attempt appear in SIGCSE 2005. In [8] Okasaki suggests a simple alternative to Red-Black trees, but his method can't be extended to deletion. In [1] Barnes et al. present Balanced-Sample Binary Search trees in hoping to improve the balance of binary search trees without demanding too many extra techniques. While their method is indeed simple and straightforward, the experimental results do not show significant improvement; only about 12% of improvement in heights on random data.

Before our investigation initiated, we had had the following observation. When asked what to do about an unbalanced binary search tree, students usually respond intuitively that we should move some nodes from the bigger child (the one with more nodes in it) to the smaller one. If we approve this why-not-idea and give them some time, it is not hard for students to figure out how to implement the idea. In fact, moving a node from one child to the other is the easiest operation for reconstructing binary search trees. Pedagogically, we should let students practice this operation before rush them into more advanced techniques. Therefore, we polish and analyze this intuitive idea and present what we call the Node-balanced Binary Search Tree in this paper.

2 Node-balanced Binary Search Trees

We first fix some notations. For every node α in a binary tree, we simply use α to denote the subtree rooted at α. Let α_n and α_h denote the number of nodes in α and the height of α. Let α-L and α-R denote the left-child and right-child of α, respectively. Similarly, let α-L_n and α-R_n denote the numbers of nodes in α-L and α-R. Also, we use α-L_h and α-R_h to denote their heights. The null node is denoted by $\emptyset$ that also represents the empty tree. For $\alpha \neq \emptyset$, we use α`.data` to refer to the data in α. Let $\alpha \leftarrow t$ mean to assign t to α`.data` and $\alpha \leftrightarrow t$ mean to swap the values of α`.data` and t.

Definition 1 *For $k \geq 1$, we say that a binary search tree is node-balanced with degree k if, for every node α in the tree, we have*

$$-\frac{\alpha_n}{k} \leq \alpha\text{-}L_n - \alpha\text{-}R_n \leq \frac{\alpha_n}{k}. \tag{1}$$

For convenience, we use NB_k to denote node-balanced binary search tree with degree k. Clearly, when $k = 1$, NB_1 is simply the ordinary binary search tree. We use BST to denote the ordinary binary search tree for the time being. To maintain an NB_k, we need to keep tracking the numbers of nodes in the left-child and right-child. Thus, we add an integer field in every node, which is the only extra information added. To adjust an unbalanced tree, the operation for NB_k is simply shifting a node from the bigger child to the smaller one. The function `shif_to_left(`α`)` that shifts a node from α-R to α-L is shown in Figure 1. The symmetric counterpart, `shif_to_right(`α`)`, can be obtained easily by changing t to the maximum data in α-L and exchanging α-L and α-R in the two function calls.

In the path starting from the root travelled during an inserting/deleting operation on an NB_k, whenever imbalance is found at node α, an appropriate shifting operation will be taken immediately. This is different from the AVL tree where the rotation should be performed at the last unbalanced node in the path. Thus, there is no need to have back-links in the nodes or maintain a stack of nodes for travelling through the insertin/deletion path backward. Consequently, the algorithms for insertion and deletion are straightforward. If we find that inserting a node to a subtree will make the subtree to have too many nodes, we shift out one node first. Similarly,

[2]The timing is based on the assumption that the topic of sorting techniques is not included or is introduced in the second half of the semester after the topic of Trees is finished.

[3]Most students can get by Stacks and Queues with a little help.

```
shif_to_left(α):
 1. Find t, the minimum data in α-R;
 2. insert(α-L, α.data);  // Insert α.data to α-L
 3. α ← t;
 4. delete(α-R, t ).          // Delete t from α-R
```

```
insert(α,t):
 1. If (α = ∅) then
     1)  new α;
     2)  α ← t;
     3)  return α.
 2. If (t < α.data) then              // t goes to α-L
     1)  If (α-L_n - α-R_n + 1 > α_n/k) then
             (1) shif_to_right(α);
             (2) If (t > α.data) then
                   α ↔ t;
     2)  α-L = insert(α-L,t);
    else                              // t goes to α-R
     1)  If (α-R_n - α-L_n + 1 > α_n/k) then
             (1) shif_to_left(α);
             (2) If (t < α.data) then
                   α ↔ t;
     2)  α-R = insert(α-R,t);
 3. return α.
```

```
delete(α,t):
 1. If (α = ∅) then return α.
 2. If (t < α.data) then              // Search α-L for t
     1)  α-L = delete(α-L,t);
     2)  If (α-R_n - α-L_n + 1 > α_n/k) then
             shif_to_left(α);
     3)  return α.
 3. If (t > α.data) then              // Search α-R for t
     1)  α-R = delete(α-R,t);
     2)  If (α-L_n - α-R_n + 1 > α_n/k) then
             shif_to_right(α);
     3)  return α.
 4. If (α_n = 1) then                 //  t found at α
      delete α and return ∅.          // α is a leaf
 5. If (α-L_n > α-R_n) then
     1)  Find β, the maximum node in α-L;
     2)  α ← β.data and delete(α-L, β.data);
    else
     1)  Find β, the minimum node in α-R;
     2)  α ← β.data and delete(α-R, β.data);
 6. return α.
```

Figure 1: Algorithms for insertion and deletion

if we find that deleting a node from a subtree will make the subtree to have too few nodes, we shift in one node. In Figure 1, `insert(`α`,`t`)` is a function that inserts data t to subtree α and `delete(`α`,`t`)` deletes a node with data t, if any, from subtree α. Both functions will return the resulting tree. We implement the two functions in a recursive fashion. For deletion, since we do not know if the data to be deleted actually exists in the tree, the balance will be checked after deletion. See Figure 1 for details.

3 Analysis and Experiments

The analysis of NB_k is every easy too. No particular advanced mathematical background is needed. Suppose α is a node-balanced tree with balance degree k and $\alpha_n = n$. Let a be the number of nodes in the smaller child. According to (1) in the definition of NB_k, in the worst case, we have

$$a + a + \frac{n}{k} = n - 1 \approx n.$$

Thus, $a = \frac{(k-1)n}{2k}$. In other words, the number of nodes in the bigger child of α is bounded by

$$a + \frac{n}{k} = n(\frac{k+1}{2k}).$$

Let h be the depth of a tree (the root is at depth 0). In the worst case, the upper bound of the number of nodes in a subtree at depth h is

$$n(\frac{k+1}{2k})^h.$$

Clearly, when

$$(\frac{k+1}{2k})^h < \frac{1}{n} \tag{2}$$

there will be no node left. From (2), we have

$$h(\log(k+1) - \log(2k)) < -\log n.$$

In other words, when

$$h > \frac{\log n}{\log(2k) - \log(k+1)} \approx \log n \tag{3}$$

there will be no nodes left for another subtree. Therefore, the height of NB_k with n nodes is bounded by $c \cdot \log n$ where $c \approx 1$ when k is sufficiently large.

For BST, it is clear that inserting n nodes in the worst case will result in a tree with height of n. For the average case (i.e., the random binary search tree, which means the data to be inserted arrives randomly), the expected heights is $O(\log n)$. The proof of this asymptotic result is comprehensible for upper level undergraduates, but to obtain the precise upper bound turns out to be an academic research topic. Here we display some results for comparison. Devroye and Reed [3] show that the expected height of a random binary search tree is bounded by

$$\alpha \log n + O(\log \log n), \text{ where } \alpha = 4.31107... \tag{4}$$

For the AVL tree, the analysis is easier. By solving a recurrence relation, we obtain the following result:

$$\log(n+1) \le h < \alpha \log(n+2) - \beta, \text{ where } \alpha = 1.4402, \beta = 0.32824, \tag{5}$$

where h is the height of an AVL tree of n nodes (detailed proof can be found in [5]). Comparing (3), (4), and (5), it is clear that we can construct a better binary search tree in NB_k. Also note that, since the result of our analysis is based on the worst case, it implies that the NB_k is more robust to the permutation of data, i.e., the order of data's arrivals does not substantially affect the heights of NB_k. Even for small k, the improvement is still significant. For example, when $k = 2$, we have $1/(\log(2k) - \log(k+1)) = 2.4094$, which is about 44% improvement to BST on random data. When $k = 4$, we have a result about the same as the AVL tree. In the next subsection, we will show the experimental results of constructing NB_k trees on some semi-random data.

Homework Exercises There are some immediate questions suitable for homework. We list some of them in the followings without detailed discussion due to the space constrains. These questions are not trivial; neither are they too difficult for students to solve by themselves.

1. For an NB_k with n nodes in total, how many nodes will be visited to shift a node at the root in average (or in the best/worst case)?
2. In average (or in the worst case), how many shift operations will be taken to insert/delete a node to/from an NB_k with n nodes in total?
3. Establish the relation between k in NB_k and ℓ in AVL_ℓ, where ℓ is the maximum difference of the heights of the two children in the AVL tree.

3.1 Experimental Results

To assess the computational cost of constructing NB_k trees, we implement a class for NB_k with necessary methods in MS Visual C++ on a PC with 2.66 GHz Pentium(R) 4 CPU and 1 GB of RAM under MS Windows XP. A node will be dynamically allocated (using `new`) for each newly arrived data. Necessary destructors are implemented to destroy trees that are tested and out of the scope to avoid swapping memory during another test. Also, we use 100% CPU power to run our programs.

Each data is a `struct` with three integer fields for three key values. The three keys determine the order of data. Since whether or not the data set is completely random is not our real concern, we simply use `rand()` to generate pseudo-random keys for data. Unless we manipulate the data set on purpose, the chance to have duplicate data in 1.2 million nodes is slim.

Data – Random and Unique: `Data_RU` The set, `Data_RU`, contains 1.2 million distinct records with uniform distribution in random order. Our first experiment is to construct BST, AVL, NB_2, and NB_8 trees using `Data_RU`. The results are compared in Figure 2. Using the heights of BST as the standard, we observe that NB_2 improves about 35%, while the AVL tree improves about 50%.[4] Increasing the value of balance degree k to 4 (not shown in Figure 2 for clarity), the heights will be close to the AVL tree's. When $k = 8$, the heights are slightly better than the AVL tree's and very close to optimums. Regarding the computational cost, our results show that NB_2 and BST on `Data_RU` are about the same, which is about 20% faster than the AVL tree. However, NB_8 runs slower but not by too far and NB_8 are trees with heights almost optimal.

[4] Comparing (4) and (5), we observe that the rate of improvement is not as good as the theoretical value. This is because our BST is "too" balanced due to the use of `rand()`.

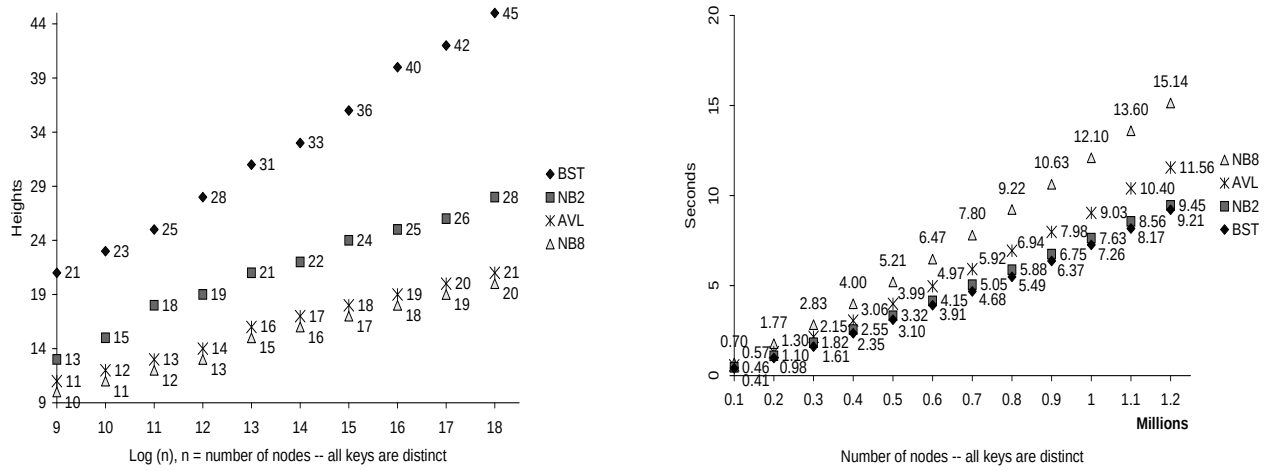

Figure 2: Using `Data_RU`

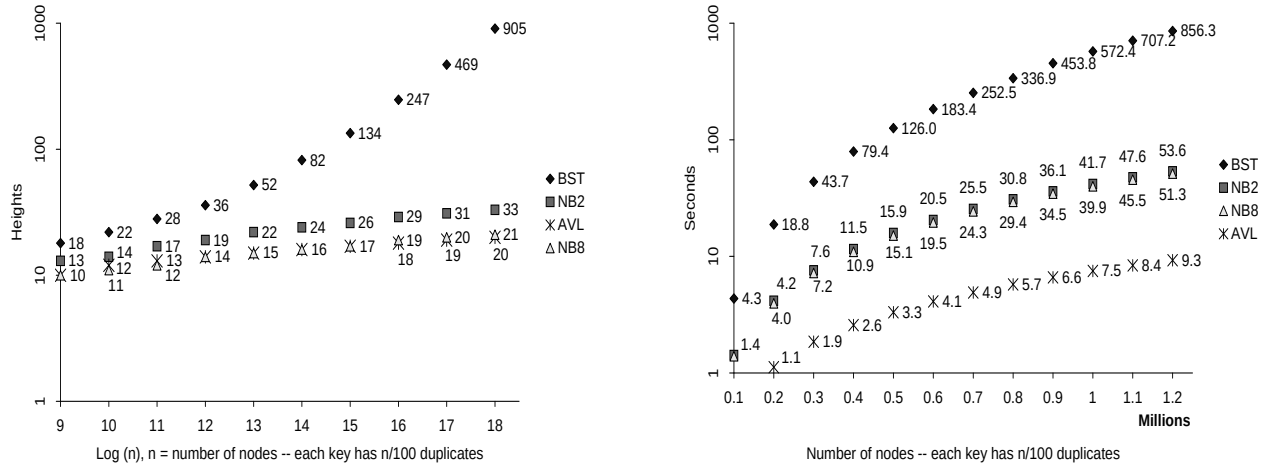

Figure 3: Using `Data_RD`

Data – Random and Duplicate: `Data_RD` In applications, there may be cases that we don't have enough key space to assign a distinct key to each record. For example, there is no need to distinguish among postal mails that are sent to the same region with the same zip code in a processing center. To reflect this situation, we create another data set, `Data_RD`, with 1.2 million records in which each key has 12,000 duplicates distributed uniformly in the data set. In other words, each key in a set of n records drawn from `Data_RD` is expected to have $n/100$ duplicates. Under this situation, BST is totaly out of control (see Figure 3). NB_k performs much better than BST even when k is as small as 2. While NB_k can be constructed much faster than BST (the time needed is only about 6% of the time needed for constructing BST), it is about 5 times slower than constructing the AVL tree.

Data – Sorted in Batches: `Data_SB` It may not be realistic to assume data to be inserted are distributed uniformly and completely random. There is another common situation in real applications, in which data may arrive in batches. Each batch may arrive randomly but each contains some sorted records. Instead of using batches, we are asked to insert each record as an independent node into a binary search tree. One can image that a postal mail procession center receives mails batch by batch, and each batch comes from a zip code region. Moreover, the mails in each batch are sorted by their regional office. To simulate this situation, we generate a data set named `Data_SB` also containing 1.2 million records. The size of each batch is fixed to 32. To generate a batch, we first generate a record (with 3 keys) as before and then set the third key to 0. Then, generate another 31 records with the first two keys identical to the first record's and the third keys set to 1, 2, ..., 31 in order.

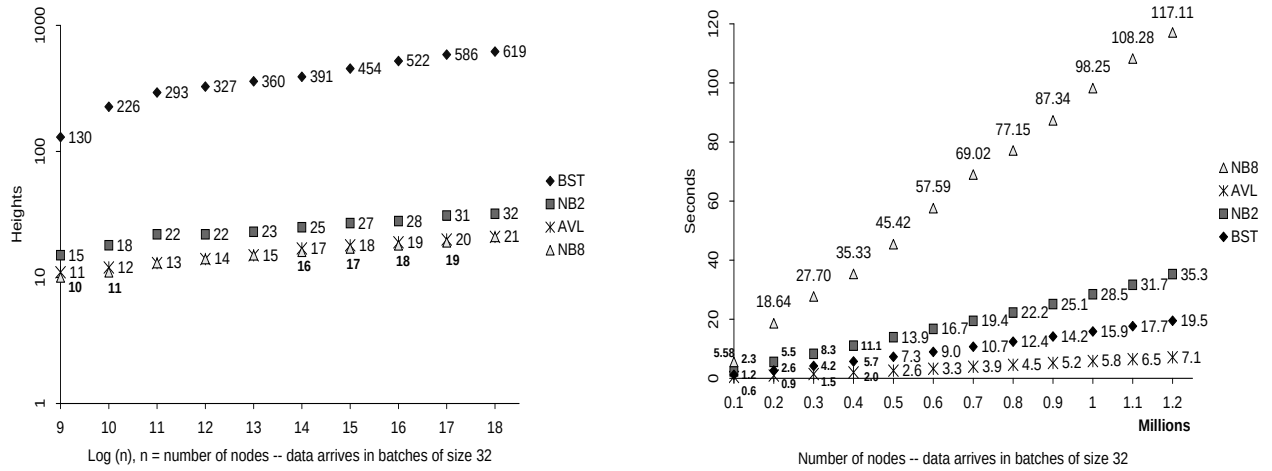

Figure 4: Using `Data_SB`

Figure 4 shows the results. The speed of constructing BST is acceptable but the heights of the resulting trees are out of control. As we expect, NB_2 and NB_8 can perfectly manage the heights, but their constructing times become slow; only NB_2 runs within an acceptable range.

4 Conclusion

Based on our experimental results, NB_k can be used under some controlled conditions (like data in `Data_RU` and `Data_RD`), but in general it is no replacement for those classic balancing techniques such as AVL trees. Our main purpose of introducing NB_k is pedagogical. Compared to any other balancing methods found in the literature, our node-balanced binary search tree is the easiest one. Thus, we believe that our approach is an immediate step to move after students have learned binary search trees, and we also believe that our approach should be considered as a missing intermediate step towards a mature yet complicate technique for balancing binary search trees.

References

[1] G. M. Barnes, J. Noga, P. D. Smith, and J. Wiegley. Experiments with balanced-sample binary trees. *SIGCSE 2005*, 37(1):166–170, March 2005.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1989.

[3] L. Devroye and B. Reed. On the variance of the height of random binary search trees. *SIAM J. Comput.*, 24(60):1157–1162, 1995.

[4] M. Drmota. An analytic approach to the height of binary search trees. *Algorithmica*, 29(1):89–119, 2001.

[5] A. Drozdek. *Data Structures and Algorithms in C++*. Thomson Course Technology, third edition, 2005.

[6] E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal*. Computer Science Press, Inc., 1984.

[7] B. Manthey and R. Reischuk. Smoothed analysis of the height of binary search trees. *Electronic Colloquium on Computational complexity*, (60), 2005.

[8] C. Okasaki. Alternatives to two classic data structures. *SIGCSE 2005*, 37(1):162–165, March 2005.

[9] M. A. Weiss. *Data Structures and Problems Solving Using JAVA*. Addison Wesley, third edition, 2006.

[10] N. Wirth. *Algorithm + Data Structures = Programs*. Prentice-Hall, Inc, 1976.