A Simple Balanced Search Tree with No Balance Criterion

Tae Woo Kim

School of Computing
Korea Advanced Institute of Science and Technology
travis1829@kaist.ac.kr

Abstract

This paper shows that we can maintain a balanced binary search tree by using a simple and intuitive method that does not use any balance criterion. Ultimately, we aimed for maximum simplicity, and we only need to add a simple rebuilding algorithm and minimal extra code to a naive binary search tree. Our method will be suitable as a simple and short solution when amortized logarithmic costs are enough.

1 Introduction

The balanced binary search tree (or balanced BST) is one of the most fundamental data structures. Though there are many ways to implement the dictionary, we use the balanced BST when we need logarithmic costs. Logarithmic cost is the key feature of balanced BSTs, and it distinguishes a balanced BST from a naive BST, where the former uses a tree balancing method to guarantee logarithmic costs. Common examples of a balanced BST include AVL trees [1], binary B-trees [4] or its simplified variant [2] (also known as the AA-tree), weight-balanced trees [6] or its variant [7], and general balanced trees [3]. Note that the splay tree [8] is not a balanced BST since its height can become linear.

However, it is a pity that such a fundamental data structure is infamous for being cumbersome to implement. To cite Munro, Papadakis, and Sedgewick [5], balanced BSTs use a notorious balancing algorithm that needs to consider numerous cases, and hence, implementations are said to be too complicated for average programmers. Not to mention that balanced BSTs are often neglected, or to cite Andersson [2], they are often replaced by poor methods instead. Andersson [3] also said in another paper that many of the commonly used balanced BSTs use a complicated balance criterion to restrict trees and detect imbalance. Also, note that especially nowadays, computer science is needed and used by so many people from all kinds of fields. In many cases, people may not need cutting-edge performance but rather just want to quickly and simply implement

their idea. These strongly emphasize the importance of a simple and explicit implementation.

The intrinsic question is how can we make it easy to implement a balanced BST. If we can assume that most people know about a naive BST, it would be simple if we only need to add a small amount of code to a naive BST to convert it into a balanced one. Only adding one line to each of the "insert" and "delete" procedures would also make it highly abstract. Sen and Tarjan said that rebalances after deletions are generally more complex than after insertions, so using the identical rebalancing method for both insertions and deletions would also help since we then only need to learn one. Also, it would be cumbersome if one needs to constantly check a tree's complicated tree restrictions every time we apply even a subtle change to the tree. Not to mention that understanding such restrictions and their various cases is also cumbersome. Then, using no tree restrictions at all would be highly helpful if it is even possible. Moreover, it would be highly convenient and abstract if we even do not need to care about a node's surroundings at all when actually changing the tree's structure, if this is even possible.

In this paper, we present a method that offers all the above by using an intuitive method. In Section 3, we explain our method and see how we can implement our method by only adding minimal code and a simple partial-rebuilding algorithm to a naive BST. Next, in Section 4, we show that our tree has a logarithmic height and that an insert/delete operation has an amortized logarithmic cost.

2 Notations and Notes

In this paper, we use the following notations.

For a node *node*,

- *node.key* is the key stored at *node*.
- *node.left* is the left child node of *node*.
- *node.right* is the right child node of *node*.
- *node.timer* is the *timer*[1] stored at *node*.

Note that in the following, when we use the word "tree," we refer to the entire tree, not just a part of it.

3 The Balancing Method

Our tree occasionally does a *partial-rebuilding*, which balances a subtree into a perfectly balanced[2] one. In most cases, a subtree is rebuilt when it does

[1] Definition of a *timer* is explained in Section 3.

[2] A perfectly balanced subtree is a subtree whose left and right subtree's size differ by one at most.

not satisfy the balance criterion. However, in our method, we instead schedule rebuilds in advance.

As a start, we briefly explain our method.

Every subtree has a ***timer****, and we decrease it by 1 each time we insert or delete a node in the subtree. If a subtree's* ***timer*** *reaches 0, we rebuild it and reset its* ***timer*** *to* $\lfloor kn \rfloor$ *(or 1 if* $\lfloor kn \rfloor = 0$*), where* $0 < k < 1$ *is a constant and* n *is the subtree's size.*

We will store a subtree's *timer* at its root node. Obviously, the *timer* stores the remaining number of insertions/deletions until the subtree's next rebuild. Additionally, for newly inserted nodes, we set the *timer* to 1. Now, using these *timer* integers, inserts and deletes are done in the following way.

Insert

(1) Insert a leaf node.

(2) Go back up the trail from the leaf node to the root. For each node t, do UPDATE(t).

Delete

(1) Delete a leaf node.[3]

(2) Go back up the trail from the leaf node to the root. For each node t, do UPDATE(t).

UPDATE(t)

(1) Decrease $t.timer$ by 1.

(2) If $t.timer$ is now 0, rebuild the subtree rooted by t.

Rebuild

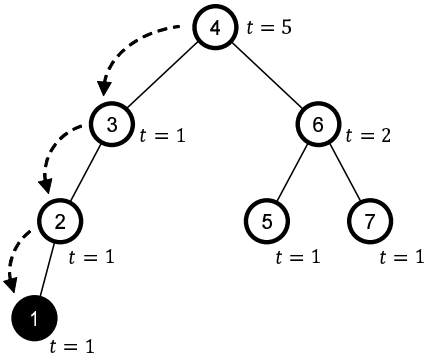
(1) GETCOPY: Using an inorder traversal, copy all nodes of the subtree to an array.

(2) BUILDTREE: Make a perfectly balanced subtree using this array in a divide-and-conquer sense. We also reset each subtree's *timer*.

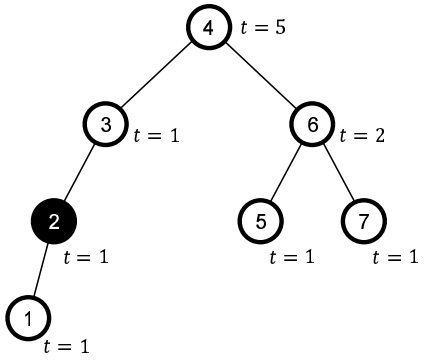
Note that for rebuilds, we used a common method that is simple and easy to implement. It only uses a basic tree traversal and a simple recursive algorithm.

[3] If the node we want to delete is not a leaf node, we use the common method of exchanging the original problem to a problem of deleting a leaf node.

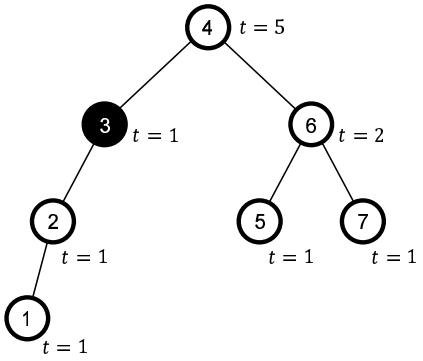
The following Fig. 1 and 2 are each an example of an insert operation and a delete operation on a tree. In both figures, the *timer* is abbreviated to t, and the black node denotes the node we want to do UPDATE.



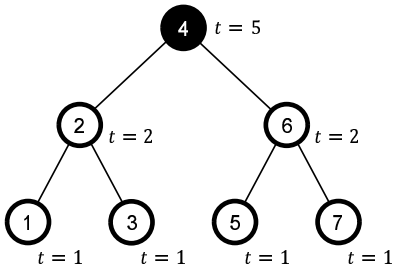
(a) Inserted "1."



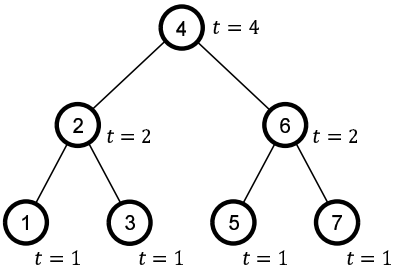
(b) Revisiting "2."



(c) Revisiting "3."

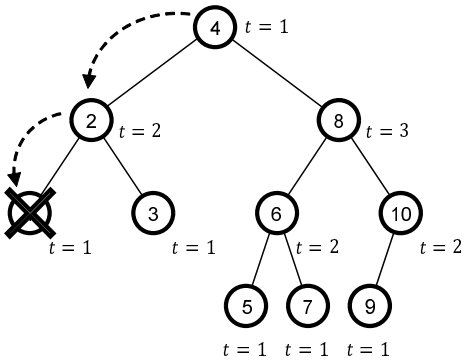


(d) Revisiting "4."

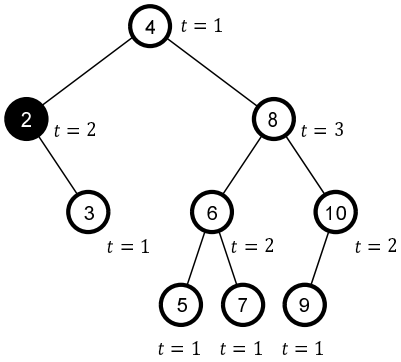


(e) Insertion completed.

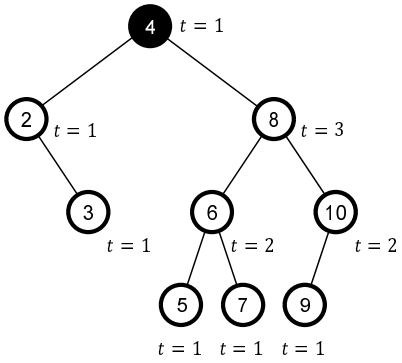
Figure 1: Example of inserting "1" to a tree.



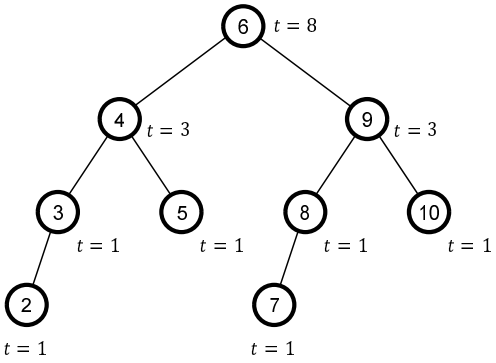
(a) Deleting "1."



(b) Revisiting "2."



(c) Revisiting "4."



(d) Deletion completed.

Figure 2: Example of deleting "1" in a tree.

Implementation

Starting from a common implementation of a naive BST, we can implement our method by adding a rebuilding algorithm and a simple, minimal code. Following is an example of an implementation. We only add Update at the end of Insert and Delete.

```
INSERT(t, key, res)
  if t = null then
    t ← new node
    res ← true
  else if key < t.key then INSERT(t.left, key, res)
  else if key > t.key then INSERT(t.right, key, res)
  else res ← false
  UPDATE(t, res)

DELETE(t, key, res)
  if t = null then res ← false
  else if key < t.key then DELETE(t.left, key, res)
  else if key > t.key then DELETE(t.right, key, res)
  else
    if t.left ≠ null and t.right ≠ null then
      t.key ← GETMAX(t.left).key
      DELETE(t.left, t.key, res)
    else
      if t.left ≠ null then t ← t.left
      else t ← t.right
      res ← true
      return
  UPDATE(t, res)

GETMAX(t)
  while t.right ≠ null do t ← t.right
  return t

UPDATE(t, res)
  if res = true then
    t.timer ← t.timer − 1
    if t.timer = 0 then
      arr ← empty array
      GETCOPY(t, arr)
      t ← BUILDTREE(arr, 0, arr.size − 1)
```

```
GetCopy(t, arr)
  if t ≠ null then
    GetCopy(t.left, arr)
    arr.push(t)
    GetCopy(t.right, arr)

BuildTree(arr, s, f)
  if s > f then return null
  m ← ⌊(s + f)/2⌋
  arr[m].left ← BuildTree(arr, s, m − 1)
  arr[m].right ← BuildTree(arr, m + 1, f)
  arr[m].timer ← max(1, ⌊k · (f − s + 1)⌋)
  return arr[m]
```

4 Proof of Logarithmic Height and Amortized Logarithmic Costs

In this section, we will show that the tree has logarithmic height, which means that a search operation has a logarithmic worst-case time complexity. Also, we will show that an insert/delete operation has an amortized logarithmic cost.

4.1 Logarithmic height

We first start with the following definition.

Definition 1. *Define $T.size$ as the size of T.*

Many trees that use partial-rebuilding maintain logarithmic height by maintaining a constant lower bound (or upper bound) on the quotient between a subtree's weight and its own left/right subtree's weight. However, this kind of analysis does not work in our tree because our tree is too flexible in shape. It is not hard to see that $\frac{T.left.size+1}{T.size+1}$ or $\frac{T.right.size+1}{T.size+1}$ is larger than $\frac{1-2k}{2-2k}$ when $0 < k < 0.5$, but when $k \geq 0.5$, such quotient is not bounded by a constant in $(0, 1)$. Therefore, we will instead show that the quotient between "upper bounds on sizes" is upper bounded by a constant.

First, lets say that when we create a new subtree (which happens when we insert a new node), the *timer* gets initially "set", and when we rebuild a subtree, the *timer* gets "reset" for all subtrees of it. Based on a subtree's last *timer* (re)set, if we refer to an insertion or deletion of a node as an "update," we can divide a subtree into two cases.

Lemma 1. *All subtrees are always in one of the following two cases.*

1. *No updates happened to it since its last timer (re)set.*

2. At least one update happened to it since its last timer (re)set.

Based on this, we will often divide subtrees into two cases. The following definition will come in handy when we do so.

Definition 2. *Define $T.n_0$ as the size of T after its last timer (re)set.*

Also, note the following.

Lemma 2. *If a subtree of size less than $\frac{2}{k}$ went through no updates since its last timer (re)set, then its timer gets reset after a single update in it, assuming it is still non-empty.*

Lemma 3. *For a perfectly balanced subtree of size n, the size of its subtree is no more than $\frac{n}{2}$, if exists.*

Using these, we can see the following.

Theorem 4. *Suppose we can do any combination of x or less updates to a subtree T whose timer was (re)set. If $T.size \leq n$ in all cases, then T's subtree's size is no more than $\left(\frac{1+2k}{2+2k}\right) n$ in all cases.*

Proof. Suppose not. That is, T's size cannot exceed n in any case, but we can have a case where T's subtree's size is bigger than $\left(\frac{1+2k}{2+2k}\right) n$. Suppose that this happened after doing u updates since T's last *timer* (re)set.

Then, $0 \leq u < \max(1, \lfloor kT.n_0 \rfloor)$. Also, because of Lemma 3, T's subtree's size should be no more than $\frac{T.n_0}{2} + u$, so $\frac{T.n_0}{2} + u > \left(\frac{1+2k}{2+2k}\right) n$ must also be true. Finally, note that $n \geq T.n_0 + u$ because of our assumption.

However, note the following.

(i) If $T.n_0 < \frac{2}{k}$, then $u = 0$, and we can see that

$$\frac{T.n_0}{2} + u \leq \left(\frac{1+2k}{2+2k}\right) T.n_0 \leq \left(\frac{1+2k}{2+2k}\right) n.$$

(ii) If $T.n_0 \geq \frac{2}{k}$, then $u \leq \lfloor kT.n_0 \rfloor - 1 \leq kT.n_0$.
However, we said that

$$\frac{T.n_0}{2} + u > \left(\frac{1+2k}{2+2k}\right) n \geq \left(\frac{1+2k}{2+2k}\right) (T.n_0 + u),$$

and this means $u > kT.n_0$, which contradicts.

Therefore, we conclude that it is impossible. □

Also, it is not hard to see that $\left(\frac{1+2k}{2+2k}\right)$ is the smallest constant that we can guarantee in Theorem 4.

Now, we finally have the following.

Theorem 5. *The height of a non-empty tree with size n is no more than $\left\lfloor \log_{\frac{2+2k}{1+2k}} \frac{1+2k}{2-2k} n \right\rfloor + 1$. That is, the height of the tree is $O(\log n)$.*

Proof. We divide the tree into two cases using Lemma 1.

A tree in the first case of Lemma 1 is perfectly balanced. That is, its height is no more than $\lfloor \log_2 n \rfloor$.

Now, we check for a non-empty tree T in the second case of Lemma 1. Note that $T.n_0 \geq \frac{2}{k}$ by Lemma 2. Also, note that we are considering trees that went through no more than $\lfloor kT.n_0 \rfloor - 1$ any updates since its last *timer* reset.

Then, for both T's left or right subtree, its size cannot exceed $\frac{T.n_0}{2} + \lfloor kT.n_0 \rfloor - 1$ for any combination of no more than $\lfloor kT.n_0 \rfloor - 1$ updates. Also, note that $n \geq T.n_0 - \lfloor kT.n_0 \rfloor + 1$. It is not hard to see that the quotient between the two is less than $\frac{1+2k}{2-2k}$. Finally, using Theorem 4, we can easily see that the size of a subtree rooted by a node at depth d is always less than

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

Therefore, we conclude that the height is no more than $\left\lfloor \log_{\frac{2+2k}{1+2k}} \frac{1+2k}{2-2k} n \right\rfloor + 1$. □

Note that a different k value leads to a different upper bound on height. In our tree, we can say that a smaller k leads to a smaller height but more frequent rebuilds.

4.2 The amortized cost of an insert/delete operation

Theorem 6. *Assume that rebuilding a subtree T costs no more than $T.size$. If a subtree's timer reached 0, then the quotient between its rebuild cost and u, the number of updates it went through since its last timer (re)set, is less than $\left(\frac{2}{k} + 1\right)$. That is, the rebuild cost is less than $\left(\frac{2}{k} + 1\right) u$.*

Proof. Call $n = T.size$ and $n_0 = T.n_0$. Then, $n \leq n_0 + u$, and $u = \max(1, \lfloor kn_0 \rfloor)$. Now, we check two cases.

(i) If $n_0 < \frac{1}{k}$, then $u = 1$ and $n < \frac{1}{k} + 1$.

(ii) If $n_0 \geq \frac{1}{k}$, then $u = \lfloor kn_0 \rfloor$ and $n \leq n_0 + \lfloor kn_0 \rfloor$.
Then, for the quotient between the rebuild cost and u,

$$\frac{n}{u} \leq \frac{n_0 + \lfloor kn_0 \rfloor}{\lfloor kn_0 \rfloor} = \frac{n_0}{\lfloor kn_0 \rfloor} + 1 < \frac{2}{k} + 1.$$

Therefore, in both cases, the quotient is less than $\left(\frac{2}{k} + 1\right)$. □

This means that if we add $\left(\frac{2}{k}+1\right)$ extra credit at all ancestor nodes of the inserted/deleted node after each insertion/deletion of a node, a subtree's root will always have enough credit to pay the subtree's rebuild cost by the time its *timer* reached 0. Since our tree's height is $O(\log n)$, the number of ancestor nodes of the inserted/deleted node is also $O(\log n)$, and therefore, we conclude with the following.

Theorem 7. *Our tree can be maintained with $O(\log n)$ amortized cost per insert/delete operation, where n is the size of the tree the operation occurred.*

5 Conclusion

It shows that we can maintain a balanced BST with a simple and intuitive method that does not require much detail or consideration of various cases. Starting from a naive BST, we only need to add a simple rebuilding algorithm and minimal code. For novice or when amortized logarithmic costs are enough, our method will be suitable as a simple solution.

References

[1] G. Adelson-Velsky and E. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):1259–1262, 1962.

[2] A. Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, WADS '93, pages 60–71, Berlin, Heidelberg, 1993. Springer-Verlag.

[3] A. Andersson. General balanced trees. *J. Algorithms*, 30(1):1–18, Jan. 1999.

[4] R. Bayer. Binary b-trees for virtual memory. In *Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*, SIGFIDET '71, pages 219–235, New York, NY, USA, 1971. ACM.

[5] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[6] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 137–142, New York, NY, USA, 1972. ACM.

[7] M. H. Overmars. *Design of Dynamic Data Structures*. Springer-Verlag, Berlin, Heidelberg, 1987.

[8] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.