Leafy Tree: A Weight-Balanced Tree

Translation compiled by Kirito Feng

Wang Siqi

CTSC 2018

Contents

Abstract

Binary trees and balanced trees are important data structures in informatics competitions. They have a great many uses when it comes to maintaining collections or sequences.

This article introduces the **leafy tree**, a data structure that can implement most binary-tree-based operations. This paper describes its implementation as both a binary search tree and a weight-balanced tree, and compares them with other, similar data structures.

§1 Foreword

In recent years, the number of data structure problems in informatics competitions has gradually increased. Many problems need to be maintained using an n-ary tree (e.g. segment tree, balanced binary search tree tree, leftist tree, etc.), with balanced binary search trees being particularly important.

Common balanced binary search trees include the splay tree, treap, etc, however, they all have problems, such as a large constant factor, long code, and a difficulty in making persistent. In response to these problems, this paper introduces a new binary tree structure - leafy tree, and a new balancing strategy - weighted balance. Combining these two, most of these problems can be overcome.

In the second section of this article, the leafy tree data structure is introduced.

In the third section, the method of implementing a binary search tree using the leafy tree is described.

In the fourth section, a the concept of weight-balanced tree is introduced, and an implementation with leafy trees is described.

§2 Description of Leafy Tree

The leafy tree is a binary tree where each node is either a leaf or has two children. The values are stored entirely on the leaf nodes, and the information stored by each non-leaf node is the aggregate of its children. Thus its structure is similar to the that of a segment tree.

> **Remark.** The author compares this to the process of Kruskal's, specifically what I presume is the merging of the DSU data structure. I have omitted this part as it makes little sense to me, and does not impact the understanding of the data structure.

This is a purely functional data structure that can be easily and efficiently implemented with other functional data structures, thus lending itself easily to persistence.

§3 Leafy Tree and Binary Search Trees

§3.1 Definitions

Consider maintaining a collection of elements with a leafy tree. Each leaf node corresponds to an element in the collection, and each non-leaf node maintains the value of its right child (i.e. the subtree maximum).

As the left sub tree's maximum value is less than the minimum value of the right subtree, the tree satisfies the binary search tree property.

§3.2 Basic Operations

For the sake of convenience, we denote the left and right children of A as $A.left$, and $A.right$, respectively, and the value of A as $A.value$. For a leaf node X, we define $X.left = X.right = null$. Additionally, we define the functions `newNode(x)` and `deleteNode(x)` to create a new node with value x, and delete node x, respectively.

§3.2.1 Find

The process of finding a value x in T's subtree is as follows:

If T is a leaf node, return true if $T.value$ equals x, and false otherwise.

Otherwise, we compare $T.left.value$ and x. If $x \leq T.left.value$, we recursively search in T's left child, otherwise we search in T's right child.

The pseudocode is shown in Algorithm 1 below.

Algorithm 1: The leafy tree's find operation.

```
function Find(T, x)
    # T is the root node, and x is the value to find.
    # The function returns true iff x is found.
    if T.left = null then
        if T.value = x then
            return True
        else
            return False
        end if
    else
        if x ≤ T.left.value then
            return Find(T.left, x)
        else
            return Find(T.right, x)
        end if
    end if
end function
```

§3.2.2 Insert

The process of inserting a value x into T's subtree is as follows:

If T is a leaf node, create nodes A and B such that $A.value = \min(T.value, x)$ and $B.value = \max(T.value, x)$. Set A to be the left child of T, and B to be its right child, and update $T.value$ to reflect this change.

Otherwise, we determine if x goes in T's left or right subtree, respectively, and recursively call insert on it.

The pseudocode is shown in Algorithm 2 below.

Algorithm 2: The leafy tree's insert operation.

```
function Insert(T, x)
    # T is the root node, and x is the value to insert.
    if T.left = null then
        A ← newNode(T.value)
        B ← newNode(x)
        if A.value > B.value then
```

```
            swap(A, B)
        end if
        T.left ← A
        T.right ← B
        T.value ← B.value
    else
        if x ≤ T.left.value then
            Insert(T.left, x)
        else
            Insert(T.right, x)
        end if
            T.value ← T.right.value
    end if
end function
```

§3.2.3 Delete

The process of deleting a value x in T's subtree is as follows:

If T is a leaf node, and $T.value$ equals x, then we set T's parent to T's sibling, and then remove both T and its sibling. Otherwise, if T's value does not equal x, then the deletion fails.

Otherwise, we determine if x belongs in T's left or right subtree, respectively, and recursively call delete on it.

The pseudocode is shown in Algorithm 3 below.

Algorithm 3: The leafy tree's delete operation.

```
function Remove(T, x)
    # T is the root node, and x is the value to delete.
    # The function returns true iff x was deleted.
    if x ≤ T.left.value then
        if T.left.size = 1 then
            if T.left.value = x then
                temp ← T.right
                T = T.right
                deleteNode(temp)
                deleteNode(T.left)
                return True
            else
                return False
            end if
        else
            return Remove(T.left, x)
        end if
    else
        if T.right.size = 1 then
            if T.right.value = x then
                temp ← T.left
                T = T.left
                deleteNode(temp)
                deleteNode(T.right)
```

```
25                  return True
26              else
27                  return False
28              end if
29          else
30              temp ← Remove(T.right, x)
31              T.value ← T.right.value
32              return temp
33          end if
34      end if
35  end function
```

Although the pseudocode is quite long, when actually implementing, if you ensure the deleted number exists, and you maintain the children of each node with an array, you can significantly cut down on code length.

§3.3 Comparison with the Binary Search Tree

The insertion and deletion operations are quite easy to implement due to how the information is maintained. Every node that is inserted or deleted is guaranteed to be a leaf node, which removes the casework that the regular binary search tree has to consider when deleting nodes. At the same time, each non-leaf node maintains interval information, so the position is determined by the interval of the left child. To simplify the implementation, we can think of a leafy tree as a binary search tree rooted at its left child's value.

§4 Leafy Tree and Weight-Balanced Search Trees

A weight-balanced tree (also known as a $BB[\alpha]$ tree) is a binary search tree that is augmented with subtree size.

The weight of a subtree equals to the sum of its two children. Additionally, if a node x satisfies $\min(weight[x.left], weight[x.right]) = \alpha \cdot weight[x]$, then this node is said to be α weighted. Obviously, $0 < \alpha \leq \frac{1}{2}$. The height h of such a tree satisfies $h \leq \log_{\frac{1}{1-\alpha}} n = \mathcal{O}(\log n)$

§4.1 Definitions

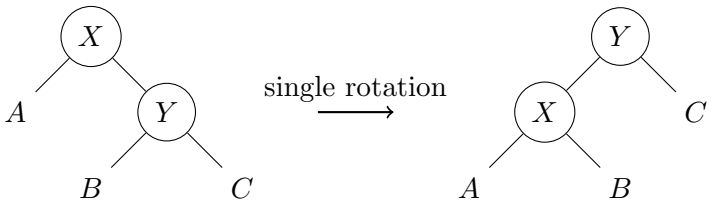
In the weight-balanced tree implemented with a leafy tree, the subtree size equals to the number of leaf nodes in the subtree, and the value is equal to the value of the right child. That is, for a leaf node x, $x.size = 1$, and for a non-leaf node y, $y.size = y.left.size + y.right.size$, and $y.value = y.right.value$. The weight of a node x, equals it subtree size, that is $weight[x] = x.size$.

§4.2 Implementation

There are two methods of implementing a weighted balanced tree, rebuilding and rotating. The only method we will discuss here is rebalancing with tree rotations. For a weight-balanced tree with a constant α, the nodes that do not satisfy the given property will form a chain. We consider what happens when we process this chain.

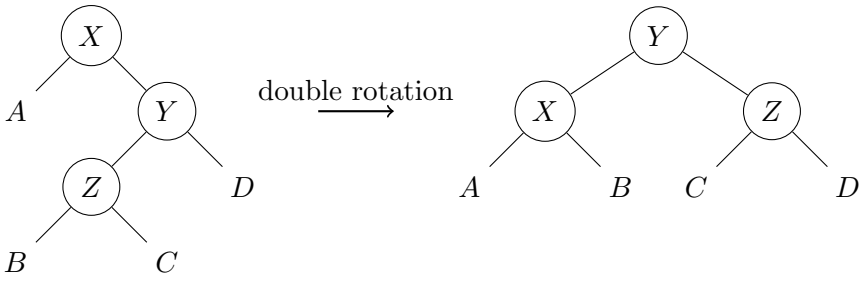
Let us assume that there is a subtree T where all nodes but the root satisfy the balance condition. We can use a single or double rotation to make the root also satisfy the given condition.

The **balance factor** of a node x is defined as $\frac{weight[x.left]}{weight[x]}$.



Let us consider a single rotation, as illustrated by the above diagram. Let X and Y have balance factors ρ_1, ρ_2 before rotation, and γ_1, γ_2 after rotation, respectively.

After rotation, we obtain $\gamma_1 = \frac{\rho_1}{\rho_1+(1-\rho_1)\rho_2}$ and $\gamma_2 = \rho_1 + (1-\rho_1)\rho_2$.



Let us consider a double rotation, as illustrated by the above diagram. Let X, Y, and Z have valance factors ρ_1, ρ_2, ρ_3 before rotation, and γ_1, γ_2, γ_3 after rotation, respectively.

After rotation, we obtain $\gamma_1 = \frac{\rho_1}{\rho_1+(1-\rho_1)\rho_2\rho_3}$, $\gamma_2 = \rho_1 + (1-\rho_1)\rho_2\rho_3$, and $\gamma_3 = \frac{\rho_2(1-\rho_3)}{1-\rho_2\rho_3}$.

If we assume that $\rho_1 < \alpha$, then in the case where a node is either inserted or deleted, ρ_1 is at least $\frac{\alpha}{2-\alpha}$, which can be achieved when we delete a node from the subtree of A.

With these constraints, it can be proven that when $\alpha \leq 1 - \frac{\sqrt{2}}{2}$, the new balance factor will end up in the range $[\alpha, 1-\alpha]$. The exact proof is not provided in the source document, and the translator is currently too tired to attempt a proof...

The value of ρ_2 determines whether a single or double rotation is performed. If $\rho_2 < \frac{1-2\alpha}{1-\alpha}$, we perform a single rotation, otherwise a double rotation is performed.

§4.3 Basic Operations

For convenience, we let $A.child[0]$ and $A.child[1]$ denote the left and right child of A, respectively. As with before, `newNode(x)` creates a node with value x, and `deleteNode(x)` deletes node x.

§4.3.1 Subtree Aggregates

After a node has been manipulated, its subtree size and value need to be recomputed.

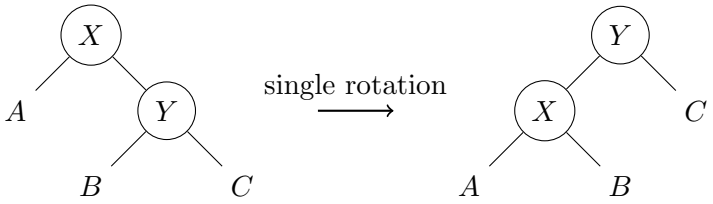
The pseudocode for this is shown in Algorithm 4 below.

Algorithm 4: The leafy tree's aggregate update operation.

```
1 function Pushup(T)
2     # T is the node whose aggregate needs updating.
3     if not T.left = null then
```

```
4           T.size ← T.left.size + T.right.size
5           T.value ← T.right.value
6       end if
7   end function
```

§4.3.2 Tree Rotations



As shown by the diagram above, the goal of a tree rotation is to rotate Y to X's old position. In order to maintain the validity of all pointers, X and Y have their positions exchanged after the rotation, rather than the simple swapping of values.

The pseudocode for this is shown below in Algorithm 5. Note that $\oplus$ represents the bitwise XOR operation.

Algorithm 5: The leafy tree's tree rotation operation.

```
1  function Rotate(T,d)
2      # T is the node whose position we are taking.
3      # d represents which child we are rotating.
4      temp ← T.child[d ⊕ 1]
5      T.child[d ⊕ 1] ← T.child[d]
6      T.child[d] ← T.child[d ⊕ 1].child[d]
7      T.child[d ⊕ 1].child[d] ← T.child[d ⊕ 1].child[d ⊕ 1]
8      T.child[d ⊕ 1].child[d ⊕ 1] ← temp
9      Pushup(T.child[d ⊕ 1])
10     Pushup(T)
11 end function
```

§4.3.3 Maintaining the Balance

After a node has been modified, we need to maintain the tree's structure using a series of double and single rotations.

The pseudocode for this is shown below in Algorithm 6.

Algorithm 6: The leafy tree's tree maintain operation.

```
1  function Maintain(T)
2      # T is the node to maintain the balance of.
3      if not T.left = null then
4          if T.left.size < T.size · α then
5              d ← 1
6          else if T.right.size < T.size · α then
7              d ← 0
8          else
9              return
10         end if
```

```
11          if T.child[d].child[d ⊕ 1].size ≤ T.child[d].size ·
               (1 - 2 α) / (1 - α) then
12              Rotate(T.child[d],d ⊕ 1)
13          end if
14          Rotate(T,d)
15      end if
16  end function
```

§4.3.4 Insert

This is quite similar to the algorithm we showed in section 3.2.2.

The pseudocode for this is shown below in Algorithm 7.

Algorithm 7: The leafy tree's tree insert operation.

```
1  function Insert(T, x)
2      # T is the root of the subtree to insert into.
3      # x is the value to insert.
4      if T.size = 1 then
5          T.left ← newNode(x)
6          T.right ← newNode(T.value)
7          if x > T.value then
8              swap(T.left, T.right)
9          end if
10     else
11         Insert(T.child[x > T.left.value], x)
12     end if
13     Pushup(T)
14     Maintain(T)
15 end function
```

§4.3.5 Delete

This is quite similar to the algorithm we showed in section 3.2.3.

The pseudocode for this is shown below in Algorithm 8.

Algorithm 8: The leafy tree's tree delete operation.

```
1  function Remove(T, x)
2      # T is the root node , and x is the value to delete.
3      d ← x > T.left.value
4      if T.child[d].size = 1 then
5          if T.child[d].value = x then
6              deleteNode(T.child[d])
7              temp ← T.child[d ⊕ 1]
8              T.left ← T.child[d ⊕ 1].left
9              T.right ← T.child[d ⊕ 1].right
10             T.value ← T.child[d ⊕ 1].value
11             deleteNode(temp)
12         else
13             return
14         end if
```

```
15      else
16          Remove(T.child[d])
17      end if
18      Pushup(T)
19      Maintain(T)
20  end function
```

§4.3.6 Find

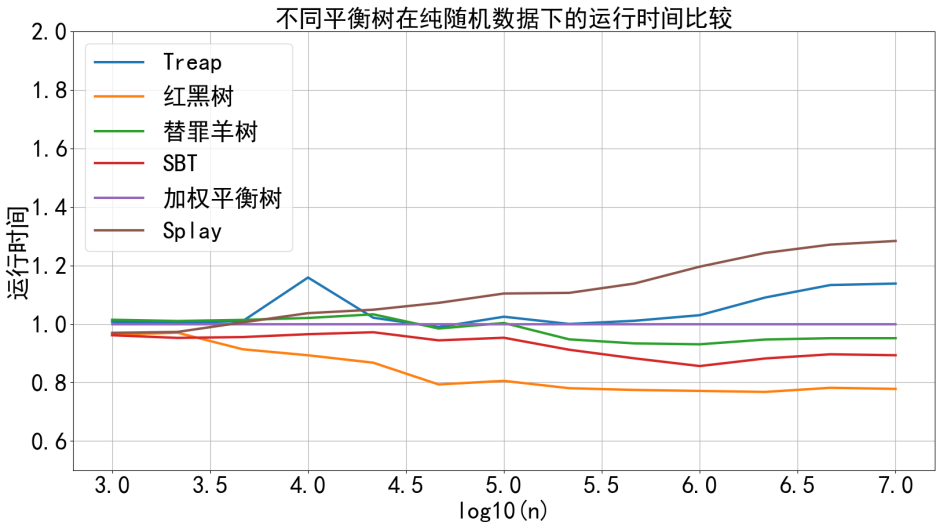
This is exactly the same as the code shown in Algorithm 1, and as such, will not be shown again.

§4.4 Runtime

Several implementations of binary search trees were tested using LOJ #104, a template BBST problem.

Since the results of using several different data generation methods were quite similar, we will only look at the results on purely random test data.

The results may have some errors, and the implementations might not be the most optimized results, but it can roughly reflect the speed of these balanced binary search trees.



Remark. The balanced binary search trees compared are treap, red-black tree, scapegoat tree, size-balanced tree, leafy tree, and splay tree.

As shown in the graph, the runtime is the ratio of the actual runtime of the BBST to the actual runtime of the leafy tree.

It can be seen that although the leafy tree is slower than the red-black tree and SBT, it is faster than the treap and splay tree, and the scapegoat tree runs at a similar speed.

§4.5 Other Operations

In addition to the basic operations, the leafy tree can support splitting and merging, as well as persistence.

§4.5.1 Merging

The merge operation takes two trees, A and B, with the constraint that $\max(A) \leq \min(B)$, and merges them into a single tree. The time complexity of this operation is $\mathcal{O}\left(\log \frac{\max(A.size, B.size)}{\min(A.size, B.size)}\right)$.

Firstly, assume WLOG that $A.size \geq B.size$. If $B.size \geq \frac{\alpha}{1-\alpha} \cdot A.size$, then you can directly create a new node with A as the left child and B as the right child.

Otherwise if $A.size \geq \alpha \cdot (A.size + B.size)$, we can recursively merge A's right child with B, and then recursively merge this result with A's left child.

The final case is where we recursively merge A's left child with A's right child's left child, then recursively merge A's right child's right child with B, and finish by recursively merging the two resultant nodes.

Once again, the author does not provide a proof of time complexity, and the translator does not wish to do so either.

§4.5.2 Splitting

The split operation takes a tree T, and a constant k, and returns two trees T_1, T_2, such that $\max(T_1) \leq k$ and $\min(T_2) > k$.

The implementation of this is a straightforward recursive function, and can be thought of as the inverse of the merge operation. The time complexity of this operation is $\mathcal{O}(\log T.size)$.

Once again, the author does not provide a proof of time complexity, and the translator does not wish to do so either.

§4.5.3 Persistence

For each operation, all modified nodes are duplicated. A single insertion or deletion operation should still have a time complexity of $\mathcal{O}(\log size)$.

In the implementation of a leafy tree, the number of nodes that need to be duplicated are relatively small, and there is no need to create redundant nodes, like in the case of the split-merge treap.

> **Joke.** As someone who consistently uses the split-merge treap, I feel personally attacked.

Thus the leafy tree is able to implement persistence with a relatively small constant factor, and small memory usage, make it very advantageous when compared with several other BBSTs.

§5 Conclusion

This article introduces the **leafy tree**: a weight-balanced binary search tree, and its implementation.

Compared to a standard binary search tree, the leafy tree has simpler insertion and deletion functions, but uses twice as many nodes.

The leafy tree has a rather small constant factor, and is simpler to implement than many other BBSTS, and can be easily made persistent. Compared with the faster red-black tree, the leafy tree does not need to store additional information to maintain its balanced structure, and is easier to implement. Compared to a splay tree, the leafy tree's has about the same length of code, with the added advantage that it can be made

persistent. Compared to treap, there is no need to store an extra field for the random priority, as well has having a smaller constant factor. Compared to the scapegoat tree, the leafy tree's time complexity is not amortized, and it can be made persistent.

Thus the leafy tree is a data structure that can achieve virtually all the operations of most BBSTs, with less code, a better constant factor, and persistence.

Additionally, the leafy tree can be used to support other binary tree-based data structures, such as the segment tree, heap, etc, which, due to space limitations, cannot be introduced in this article.