Efficient Top-Down Updates in AVL Trees

Vincent Jugé
LIGM, CNRS, Univ Gustave Eiffel, Marne-la-vallée, France
IRIF, Université Paris-Cité & CNRS, France

Abstract

Since AVL trees were invented in 1962, two major open questions about rebalancing operations, which found positive answers in other balanced binary search trees, were left open: can these operations be performed top-down (with a fixed look-ahead), and can they use an amortised constant number of write operations per update? We propose an algorithm that solves both questions positively.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases AVL trees, data structures, amortised complexity

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.49

1 Introduction

Balanced search trees are among the most fundamental and basic data structures in computer science. Their formidable story started with the invention of AVL trees by Adel'son-Vel'skii and Landis [1] in 1962, who proposed a data structure and a maintenance algorithm thanks to which the insertion, deletion and research of an element in a linearly ordered set of size n was feasible in time $\mathcal{O}(\log(n))$.

AVL trees are binary search trees in which each node x maintains some form of *balance*: the heights of its children differ from each other by at most 1. Satisfying this property at each node ensures that the tree is of height $\mathcal{O}(\log(n))$. The number of comparisons required to search an element in a binary search tree being linear in the height of the tree, it is thus logarithmic in n.

However, for instance, inserting a node into the sub-tree rooted at the left child of our node x may increase the height of that sub-tree, damaging the balance of x. Thus, the challenge of maintenance algorithms consists in efficiently modifying the parenthood relations inside the tree in order to ensure that each node remains balanced.

Unfortunately, in spite of their excellent worst-case height, AVL trees suffer from updating algorithms that are significantly less efficient than those of other balanced binary search tree structures such as weight-balanced trees [9], red-black trees [4], half-balanced trees [11], splay trees [12] or, more recently, weak AVL trees [5]. Indeed, as illustrated in Table 1, they have two shortcomings, from which almost all other algorithms are exempt:

- starting from an empty tree, performing n updates in a row may trigger $\Theta(n\log(n))$ rotations [2] or, more generally, write operations, which may also consist in modifying the balance of a node without changing the tree structure;
- updates following an insertion or deletion must be performed in a bottom-up fashion; the inventors of weak AVL trees even said, in [5, p. 23], that "Top-down insertion or deletion with fixed look-ahead in an AVL tree is problematic. (We do not know of an algorithm; we think there is none.)"

Requiring only $\Theta(1)$ amortised write operations per update instead of $\Theta(\log(n))$ would clearly improve the efficiency of AVL trees. As mentioned in [4, 5], being able to use top-down algorithms is also important, because requiring a bottom-up pass is costly: either it forces each node to maintain a link toward its parent, or it must be performed *via* recursive calls or

Table 1 Amortised number of rewrite operations per update with n nodes, and existence of top-down update algorithms. Next to each piece of information are indicated references where it is proved.

Tree family	Amortised write cost/update	Top-down update algorithm
AVL (state of the art)	$\Theta(\log(n))$ [2]	no
AVL (this article)	$\Theta(1)$	yes
Weight-balanced	$\Theta(1)$ [3]	yes [7, 8]
Red-black	$\Theta(1)$ [4]	yes [4]
Half-balanced	$\Theta(1)$ [11]	for insertions [10]
Splay	$\Theta(\log(n))$ [12]	yes [12]
Weak AVL	$\Theta(1)$ [5]	yes [5]

storing visited nodes in a stack. This second approach is even more expensive in distributed settings, because each process should lock the entire branch it is visiting, preventing any update from starting before the previous update has ended.

A way of circumventing these deficiencies of AVL trees is to use weak AVL trees instead [5]. By having less demanding balance criteria, this variant of AVL offers update algorithms that require $\mathcal{O}(1)$ rotations in the worst case, and $\mathcal{O}(1)$ amortised write operations (which may include updating information on the balance of nodes) per update. Furthermore, these algorithms can be made top-down, with the following exception, which is common to all binary search tree data structures: instead of deleting an internal node x, we must delete a node y whose key immediately precedes or follows the key of x, an approach called Hibbard deletion [6], and then replace the key of x by that of y. In AVL trees, y must be a leaf or the parent of a leaf, which reduces the problem of node deletion to that of leaf deletion, up to to keeping a pointer on an internal node we may wish to delete.

In this article, we prove that top-down insertion and deletion with fixed look-ahead in an AVL tree *is* feasible; having a look-ahead of d means that the algorithm, when it reads or writes information at a node x, is forbidden to later read or write informations at nodes that do not descend from the d^{th} ancestor of x. At the same time, this can also be write-efficient, thus bridging a gap of theoretical importance: it was somewhat unsettling to observe that this seminal balanced tree structure was also the one for which no efficient algorithms had been found, thus leading specialists in this field to conjecture that such algorithms did not exist.

We proceed step-by-step: in Section 2, we briefly present textbook algorithms for updating AVL trees; in Section 3, we propose a bottom-up algorithm that requires $\mathcal{O}(1)$ amortised write operations per update; finally, in Section 4, we propose a top-down algorithm that requires $\mathcal{O}(1)$ amortised write operations per update. Note that, unlike algorithms for AVL trees or some other structures (e.g., red-black trees), a given update may require these algorithms to perform $\Theta(\log(n))$ rotations: our efficiency criterion is based on the amortised number of write operations (and, in particular, rotations) only, not on the maximal number of such operations that a single update may require.

Both algorithms have flavours similar to those used when designing weak AVL trees [5]: they rely on making sure that "transient" rebalancing operations decrease some integer-valued potential and, for the top-down algorithm, on being able to start and stop long enough yet finite sequences of such transient operations. This illustrates the usefulness of the approach used to design these algorithms, and also suggests that additional results similar

to [5, Theorem 7.4] should also be valid here. The main difficulties, however, reside first in designing a suitable notion of potential, and then in managing to start and stop these sequences of transient operations; this task is easy when dealing with insertions only, and quite challenging when taking deletions into account.

2 State-of-the-art

2.1 Preliminaries

A *binary search tree* is an ordered rooted tree $\mathcal{T}$ in which each tree node x has one *key* $\mathsf{key}(x)$, one *left child* x_1 and one *right child* x_2, which can be *tree nodes* or *null nodes*; null nodes represent the empty tree and have no key nor children, and tree nodes whose two children are null nodes are called *leaves*. By definition, the *descendants* of x consist of x, its children x_1 and x_2, and all their descendants (if they are tree nodes); these descendants form a sub-tree of $\mathcal{T}$ rooted at x and denoted by $\mathcal{T}(x)$, and x is one of their *ancestors*.

Node keys are pairwise distinct, and must belong to a linearly ordered set. For each tree node y that descends from x, we have $\mathsf{key}(y) < \mathsf{key}(x)$ if y belongs to $\mathcal{T}(x_1)$, and $\mathsf{key}(y) > \mathsf{key}(x)$ if y belongs to $\mathcal{T}(x_2)$. Moreover, the *height* of x is the length (i.e., the number of parent-child edges) of the longest downward path starting from x and ending in a leaf; it is denoted by $\mathsf{h}(x)$. By convention, each null node has height -1.

A binary search tree is an AVL tree when the height difference $\mathsf{h}(x_1) - \mathsf{h}(x_2)$ belongs to the set $\{-1, 0, 1\}$ for all tree nodes x. This ensures that an AVL tree with n nodes is of height $\mathcal{O}(\log(n))$.

Here, we follow the presentation of [5] and assume that AVL tree structures are implemented by assigning a *rank* $\mathsf{r}(x)$ to each tree node x. This rank is our current estimation of the height $\mathsf{h}(x)$, which may be slightly invalid if, say, some leaf was inserted way below x, and we did not yet have the time to update information about the height of x. Hence, we should control the rank differences $\mathsf{r}(x) - \mathsf{r}(x_i)$ for each node x. Focusing on these differences, we say that x_i is an ℓ-child if $\mathsf{r}(x) - \mathsf{r}(x_i) = \ell$, and that x is an $\{\ell, \ell'\}$-node if its children are ℓ- and ℓ'-children.

In practice, these ranks are rarely stored directly in AVL implementations: instead, each node x contains either (i) the difference $\mathsf{r}(x_2) - \mathsf{r}(x_1)$ between the ranks of its children x_1 and x_2, or (ii) the difference $\mathsf{r}(p) - \mathsf{r}(x)$ between the rank of its parent p and its own rank. In what follows, we will disregard these implementation details, and pretend that we stored node ranks in clear; adapting our algorithm to let it fit the frameworks (i) or (ii) is easy.

Since binary search trees represented ordered sets, the three main operations they must support are searching, inserting, and deleting an element. Searching an element is easy: when searching a key k in a non-empty tree $\mathcal{T}(x)$, we compare k with $\mathsf{key}(x)$; then, we keep searching k in $\mathcal{T}(x_1)$ if $k < \mathsf{key}(x)$, or in $\mathcal{T}(x_2)$ if $k > \mathsf{key}(x)$, or we declare that k was found if $k = \mathsf{key}(x)$; whereas, when searching k in an empty tree, we must declare that k was nowhere to be found.

Inserting or deleting an element is substantially more difficult, because it requires altering the structure of the tree: this may change its height and even damage the balance of some tree nodes. As a result, the relation $\mathsf{r}(x) = \max\{\mathsf{r}(x_1), \mathsf{r}(x_2)\} + 1$ might fail for one node x, or one rank difference $\mathsf{r}(x_1) - \mathsf{r}(x_2)$ might no longer belong to the set $\{-1, 0, 1\}$. The purpose of bottom-up rebalancing algorithms is precisely to eliminate this anomaly, either directly, or by "propagating it upward", i.e., making it concern a strict ancestor of x, with rank larger than $\mathsf{r}(x)$.

2.2 Tree cuts and cut rebalancing

A *cut* of a tree $\mathcal{T}$ is a list $(x^1, x^2, \dots, x^\ell)$ of nodes of $\mathcal{T}$ that are pairwise incomparable for the ancestorship relation (i.e., the sub-trees $\mathcal{T}(x^i)$ are pairwise disjoint), ordered from left to right, and which is maximal for inclusion.

Let x denote the root of $\mathcal{T}$. When the nodes $x^1, x^2, \dots, x^\ell$ have ranks $\mathsf{r}(x) - \delta^1$, $\mathsf{r}(x) - \delta^2, \dots,$ $\mathsf{r}(x) - \delta^\ell$, the *profile* of the associated cut is defined as the integer tuple $\boldsymbol{\delta} = (\delta^1, \delta^2, \dots, \delta^\ell)$. Accordingly, we say that a node x has profile $\boldsymbol{\delta}$ when $\mathcal{T}(x)$ contains a tree cut of profile $\boldsymbol{\delta}$. When the order of the integers $\delta^1, \delta^2, \dots, \delta^\ell$ is irrelevant, we may also say that x has profile $\{\delta^1, \delta^2, \dots, \delta^\ell\}$, just writing a multiset of integers: this just means that there exists a permutation σ of $\{1, 2, \dots, \ell\}$ such that x has profile $(\delta^{\sigma(1)}, \delta^{\sigma(2)}, \dots, \delta^{\sigma(\ell)})$. In particular, an $\{\ell, \ell'\}$-node is simply a node with profile $\{\ell, \ell'\}$.

Below, we may modify the structure of a tree $\mathcal{T}$, thus generalising so-called *rotations*: starting from a cut $(x^1, x^2, \dots, x^\ell)$ of $\mathcal{T}$, let $a^1, a^2, \dots, a^{\ell-1}$ be the strict ancestors of the cut nodes x^i, listed from left to right. We assign a new pair of children to each node a^i, and decide that the new root should be some node a^j, as long as the graph we obtain is a binary search tree rooted at a^j; nodes a^j are said to be *affected* by the cut rebalancing. While doing so, (i) we must never modify the structure or rank of any of the sub-trees $\mathcal{T}(x^i)$, and (ii) node ranks of nodes a^i are reassigned to make sure that each node a^i is a $\{1, 1\}$- or a $\{1, 2\}$-node in the resulting tree. Such an operation is called a *cut rebalancing*, and is easy to describe diagrammatically, like in Figure 1 below; cut nodes and their ancestors will typically be denoted with capital letters, from left to right.

2.3 Textbook bottom-up algorithm

Inserting a key may create a *zero-edge*, i.e., an edge between a parent x and its child x_i such that $\mathsf{r}(x) = \mathsf{r}(x_i)$; in that case, the integer $\mathsf{r}(x) = \mathsf{r}(x_i)$ is called the *rank* of the zero-edge. A zero-edge is created when x is a leaf below which we wish to insert the key k, because some child x_i of x, which used to be null, becomes a leaf.

Similarly, deleting a key may create a $\{2, 2\}$- or $\{1, 3\}$-node, which we call a *four-node*; this happens if x is a $\{1, 2\}$-node one of whose children is a leaf containing the key k we wish to delete, because that leaf will be transformed into a null node.

Zero-edges are dealt with as follows, and as illustrated in Figure 1. Up to using a left-right symmetry, and without loss of generality, we assume that $\mathsf{r}(x_1) \geqslant \mathsf{r}(x_2)$, i.e., that both $\mathsf{r}(x)$ and $\mathsf{r}(x_1)$ are equal to the rank r of the zero-edge:

1. If $\mathsf{r}(x_{11}) = \mathsf{r}(x_{12}) = \mathsf{r} - 1$ and $\mathsf{r}(x_2) = \mathsf{r} - 2$ (row 1, left), the cut (x_{11}, x_{12}, x_2) has profile $(1, 1, 2)$. Let A, C and E denote the cut nodes x_{11}, x_{12} and x_2, and let B and D denote their ancestors x_1 and x. We rebalance $\mathcal{T}(x)$ by letting D become the parent of C and E, and B become the parent of A and D, thus obtaining a tree with root B. This cut rebalancing is a *simple rotation*.
2. If $\mathsf{r}(x_{11}) = \mathsf{r} - 1$ and $\mathsf{r}(x_{12}) = \mathsf{r}(x_2) = \mathsf{r} - 2$ (row 1, right), we perform the same rotation; however, the resulting ranks differ.
3. If $\mathsf{r}(x_{12}) = \mathsf{r} - 1$ and $\mathsf{r}(x_{11}) = \mathsf{r}(x_2) = \mathsf{r} - 2$ (row 2, left), the cut $(x_{11}, x_{121}, x_{122}, x_2)$ is to be rebalanced. Let A, C, E and G denote the cut nodes x_{11}, x_{121}, x_{122} and x_2, and let B, D and F denote their ancestors x_1, x_{12} and x. We rebalance $\mathcal{T}(x)$ by giving these seven nodes the shape of a complete binary tree: B becomes the parent of A and C, F becomes the parent of E and G, and D becomes the parent of B and F, as well as the root of the resulting tree. This is a *double rotation*.
4. If $\mathsf{r}(x_2) = 1$ (row 2, right), we rebalance $\mathcal{T}(x)$ by not changing its structure, but simply increasing the rank of its root x. This is a *promotion*.

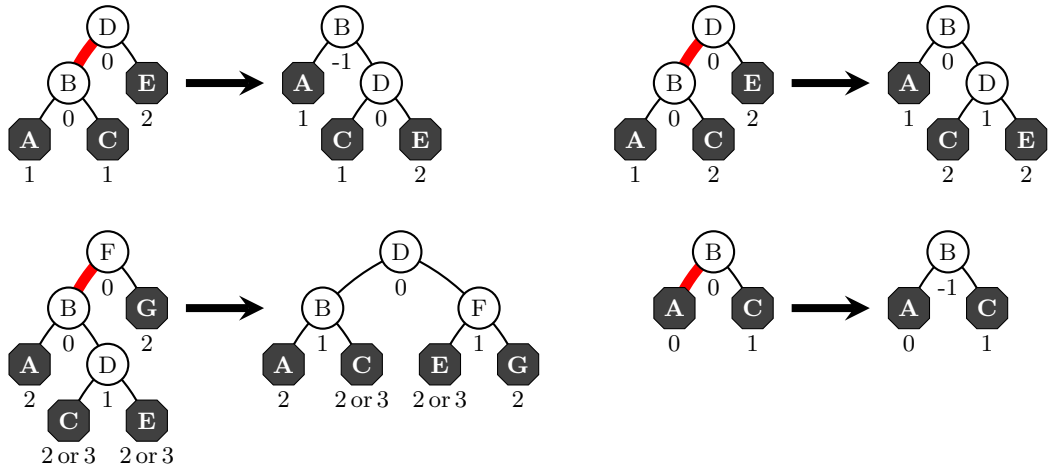

Figure 1 Eliminating or propagating upward a (thick, red-painted) zero-edge of rank r through cut rebalancings. Black octagons represent cut nodes, at which are rooted sub-trees whose structure and ranks will not change. White circles represent their ancestors. Once the rebalancing operation has taken place, the rank of a node is defined as the smallest integer larger than the ranks of its children. Below each node z is written the rank difference $\mathsf{r} - \mathsf{r}(z)$; when two rank differences δ and δ' are possible, we just write "δ or δ'". These notations will be used in all subsequent diagrams.

Doing so, whenever we face a zero-edge e of rank r, we will either eliminate it at once, or replace it by a new zero-edge e' of rank $\mathsf{r}+1$, placed just above where e' used to be. Consequently, if inserting a key creates a zero-edge, we will simply propagate that zero-edge upward until it disappears.

Likewise, four-nodes are dealt with as follows, and as illustrated in Figure 2. Up to using a left-right symmetry, we assume that $\mathsf{r}(x_1) \geqslant \mathsf{r}(x_2)$:

5. If $\mathsf{r}(x_{11}) = \mathsf{r}(x_{12}) = \mathsf{r}-2$ and $\mathsf{r}(x_2) = \mathsf{r}-3$ (row 1, left), we perform a simple rotation.
6. If $\mathsf{r}(x_{11}) = \mathsf{r}-2$ and $\mathsf{r}(x_{12}) = \mathsf{r}(x_2) = \mathsf{r}-3$ (row 1, right), we perform the same rotation; the resulting ranks differ.
7. If $\mathsf{r}(x_{12}) = \mathsf{r}-2$ and $\mathsf{r}(x_{11}) = \mathsf{r}(x_2) = \mathsf{r}-3$ (row 2, left), we perform a double rotation.
8. If $\mathsf{r}(x_2) = \mathsf{r}-2$ (row 2, right), we decrease the rank of its root x; this is a *demotion*.

Cut rebalancing operations 5 to 7 are the same as operations 1 to 3 for eliminating zero-edges, except that the ranks of x_1, x_2 and their descendants were all shifted down by one. In all eight cases, these operations result either in eliminating the four-node x or in making the (former) parent of x our new four-node.

3 Efficient bottom-up updates

3.1 Overview and complexity analysis

The textbook algorithm presented in Section 2.3 aims at eliminating or propagating upward zero-edges and four-nodes. Thus, each insertion or deletion update consists in (i) finding the location at which we want to create or delete a tree leaf, (ii) observe that we may have created a zero-edge or four-node, (iii) moving this anomaly upward, until (iv) we eliminate it once and for all. Step (i) requires no write operation on the tree structure, and steps (ii) and (iv) occur once per update. However, step (iii) may require arbitrarily many write operations per update.

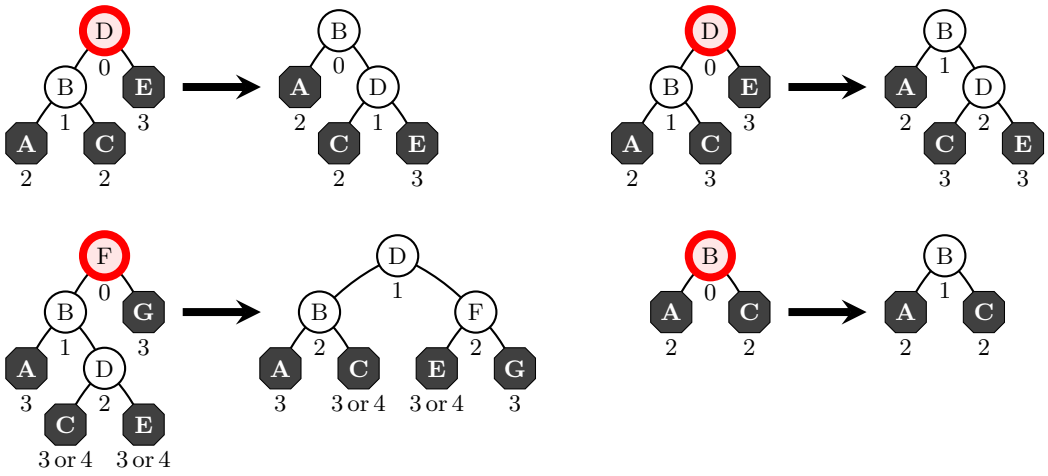

Figure 2 Eliminating or increasing the rank of a (thick, red-painted) four-node.

Hence, cut rebalancing operations of cases 1 to 8 are called *terminal* if they result in eliminating the anomaly (thus being a part of step (iv)), and *transient* if they just propagate it upward (thus being a part of step (iii)): transient operations are operations 1 and 4 when x used to be a 1-child, and operations 6 to 8 when x had a parent that used to be a $\{1,2\}$-node. We should prove that updating n times an initially empty AVL tree triggers $\mathcal{O}(n)$ transient operations.

With the textbook algorithm, the statement we wish to prove is invalid, as was shown in [2]. Consequently, we will tweak the operations performed in cases 1 to 8, aiming at making them terminal whenever possible.

The complexity proof of the resulting algorithm is based on *potential* techniques. In our case, we define the *potential* of a a tree $\mathcal{T}$ as the the number of $\{1,2\}$-nodes, plus twice the number of *full* nodes it contains, where full nodes are defined as follows.

In absence of zero-edges, a node x is called *full* when x and its two children x_1 and x_2 are $\{1,1\}$-nodes, and at least one of its grand-children x_{12} and x_{21} is a $\{1,1\}$-node. In other words, a node is full when it has profile $(2,3,3,2,2)$ or, symmetrically, $(2,2,3,3,2)$. When zero-edges are (temporarily) allowed, a node x may also be considered full if x is a $\{0,1\}$-node whose 1-child is a $\{1,1\}$-node; or if x is a $\{1,1\}$-node, x_1 and x_2 are $\{0,1\}$- and/or $\{1,1\}$-nodes, and one of x_{12} and x_{21} is a 0-child or a null, a $\{0,1\}$- or a $\{1,1\}$-node.

As we will prove:

1. each update or write operation modifies the neighbourhood (parenthood relations and/or rank) of a constant number of nodes, thus increasing the tree potential by at most a constant;
2. each transient operation aimed at eliminating a zero-edge of rank $\mathsf{r} \geqslant 4$ destroys one full node and creates one $\{1,2\}$-node, thus decreasing the tree potential;
3. each transient operation triggered by a deletion decreases the number of $\{1,2\}$-nodes, and does not increase the number of full nodes, also decreasing the tree potential.

Statement 1 will be immediate, and thus we will focus on proving statement 2 in Section 3.3 (this will be Lemma 3), and statement 3 in Section 3.4 (this will be Lemma 4). Once all three statements are proved, we will finally be able to derive the following result.

▶ **Theorem 1.** *The updating algorithm that we present in Sections 3.3 (for insertions) and 3.4 (for deletions) performs $\mathcal{O}(1)$ amortised write operations per update.*

Proof. Starting from an empty tree, let n insertion or deletion updates be performed. Let us keep track of the variations of tree potential while these updates are performed. There exists a constant K such that each update requires:

1. inserting or deleting a leaf, an operation that increases the potential by at most K;
2. eliminating at most four zero-edges of rank $\mathsf{r} \leqslant 3$, each of which increases the potential by at most K;
3. performing a certain number of transient operations, each of which decreases the potential by at least one;
4. performing at most one terminal operation, which increases the potential by at most K.

Hence, if T transient operations were performed in total, the potential increased by at most $6n\mathsf{K} - \mathsf{T}$. It started with the value 0, and ended with a non-negative value, which shows that $\mathsf{T} \leqslant 6n\mathsf{K}$. Therefore, in total, these n updates required performing no more than $\mathcal{O}(n) + \mathcal{O}(\mathsf{T}) \subseteq \mathcal{O}(n)$ write operations. ◀

3.2 Cut promotions and demotions

Some cut profiles, such as the ones that charaterise full nodes, allow rebalancing trees to either increase or decrease their rank. More precisely, if we are lucky enough, a tree $\mathcal{T}$ with rank r and profile $\boldsymbol{\delta}$ will always admit a cut, whose profile does not need to be $\boldsymbol{\delta}$, that can be rebalanced to obtain a tree $\mathcal{T}'$ of rank $\mathsf{r}' = \mathsf{r} + \varepsilon$, where $\varepsilon = \pm 1$ is an integer of our choice. This cut rebalancing operation is called a *cut promotion* when $\varepsilon = 1$, and a *cut demotion* when $\varepsilon = -1$.

Some profiles ensure that a cut promotion or demotion can be performed, as outlined by the following result.

▶ **Lemma 2.** *Trees with a full root can always be subject to a cut promotion; the root of the resulting tree is a $\{1,2\}$-node. By contrast, trees with profile $\{3,3,3,4,4\}$ can always be subject to a cut demotion; the root of the resulting tree is a $\{1,1\}$-node.*

Proof. The recipe given in Figure 3 (row 1) shows how to apply a cut promotion to a tree whose root is full. Hence, let us focus on applying a cut demotion on a tree $\mathcal{T}$ that admits a cut $(x^1, x^2, x^3, x^4, x^5)$ with profile $\{3,3,3,4,4\}$.

Replacing the three nodes x^i with rank $\mathsf{r}-3$ by their children gives an 8-node cut, whose nodes all have rank $\mathsf{r}-4$ or $\mathsf{r}-5$, at least five of these nodes having rank $\mathsf{r}-4$. These 8 nodes and their 7 ancestors can be reshaped to form a 15-node complete binary tree $\mathcal{T}'$ of height 3, whose nodes at height h must have rank $\mathsf{r}-5+h$ or $\mathsf{r}-4+h$. Moreover, among the four leaves (y^1, y^2, y^3, y^4), at least one has rank $\mathsf{r}-4$, which proves that the root x' of $\mathcal{T}'$ has a left child x'_1 of rank $\mathsf{r}-1$; similarly, its right child has rank $\mathsf{r}-2$, and x' is a $\{1,1\}$-node of rank $\mathsf{r}-1$. This is illustrated in Figure 3 (row 2). ◀

In general, when cut-rebalancing a k-node cut transforms a sub-tree $\mathcal{T}(x)$ into a tree $\mathcal{T}'$ with root x' and rank $\mathsf{r}(x)-1$, $\mathsf{r}(x)$ or $\mathsf{r}(x)+1$, the tree potential increases by no more than $2k+4$. Indeed, $k-1$ cut ancestors were affected, possibly becoming $\{1,2\}$-nodes or full nodes; the parent of x, whose child is now x', might also become a $\{1,2\}$-node or a full node; and the grand-parent and great-grand-parent of x might become full nodes. Hence, cut promotions increase the tree potential by 14 or less, and cut demotions increase this potential by 20 or less.

These two kinds of cut promotions and demotions will be used in Sections 3.3 and 3.4. Another kind of cut rebalancing operation, called *saving*, will be introduced in Section 4.3, only being useful for designing efficient top-down updating algorithms.

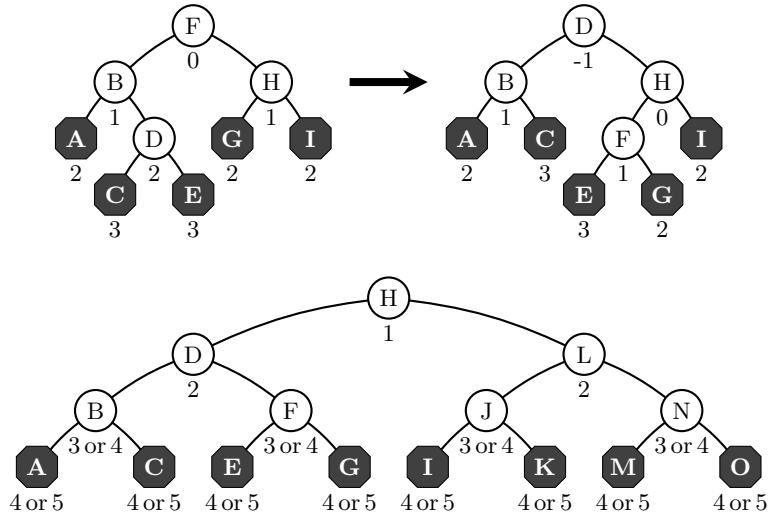

Figure 3 Row 1: Applying a cut promotion to a tree whose root is full. Row 2: 15-node complete binary tree whose root has profile $\{1, 1\}$ and rank $\mathsf{r} - 1$, which results from applying a cut demotion to a tree with profile $\{3, 3, 3, 4, 4\}$.

3.3 Insertion

Let us modify the textbook insertion algorithm to make some transient operations, which should occur in cases 1 and 4 of page 4, terminal operations. In particular, all transient operations in our variant are already transient in the textbook algorithm.

Case 1 is easy to take care of. Indeed, let us say that a node x is *weak* when x is either a leaf or a $\{1, 2\}$-node whose 1-child is also weak. An immediate induction proves that, when using the textbook algorithm, the lower end of a zero-edge is always weak. This proves, in particular, the well-known fact that case 1 is actually spurious, since it features a zero-edge whose lower end is a $\{1, 1\}$-node.

It remains to take care of case 4, which we subdivide into two sub-cases. First, we promote x like in the textbook algorithm, thus obtaining a tree $\mathcal{T}'$ of rank $\mathsf{r} + 1$. Then,

4. a) if x_2 is a $\{1, 1\}$-node, we do nothing more: we just performed a node promotion;

b) if x_2 is a $\{1, 2\}$-node, let us recall that x_1 is weak, which gives it the profile $\{2, 2, 3\}$, whereas x_2 has profile $\{1, 2\}$; hence, $\mathcal{T}'$ has profile $\{3, 3, 3, 4, 4\}$, and a cut demotion transforms $\mathcal{T}'$ into a tree $\mathcal{T}''$ of rank r again: we just performed a terminal operation.

▶ **Lemma 3.** *Each transient operation aimed at eliminating a zero-edge of rank* $\mathsf{r} \geqslant 4$ *destroys one full node and creates one* $\{1, 2\}$*-node, thus decreasing the tree potential.*

Proof. We just saw that the only transient case is case 4a, on the condition that x was a 1-child.

If the parent $x^{(1)}$ of x is full after x was promoted, it must now be a $\{0, 1\}$-node whose other child is a $\{1, 1\}$-child. Hence, before promoting x, its parent $x^{(1)}$ was already a $\{1, 1\}$-node whose children were a $\{0, 1\}$-and a $\{1, 1\}$-node, its child x having two children x_1 and x_2 that were a 0-child and a $\{1, 1\}$-node. This means that $x^{(1)}$ was already full.

Similarly, if the grand-parent $x^{(2)}$ of x is full after x was promoted, it must now be a $\{1, 1\}$-node whose children are a $\{1, 0\}$-node (the node $x^{(1)}$) and a $\{1, 1\}$-node (the sibling of $x^{(1)}$), one of it grand-children $x^{(2)}_{12}$ or $x^{(2)}_{21}$, say y, being a 0-child or a $\{1, 1\}$-node. Before promoting x, the node $x^{(1)}$ was already a $\{1, 1\}$-node, and either $y = x$, in which case y

was already a $\{0,1\}$-node, or $y \neq x$, in which case y was already a $\{1,1\}$-node. This means that $x^{(2)}$ was already full.

Finally, promoting x, in case it used to be a 1-child, does not make any other ancestor of x a full node. Hence, our transient operation transformed a full node into a $\{1,2\}$-node, and did not create any other full node or $\{1,2\}$-node: as a result, it decreased the tree potential by one. ◀

3.4 Deletion

Similarly, we wish to transform as many transient operations, which occur in cases 6 to 8 of the textbook textbook algorithm, into terminal operations.

The rotations and demotion performed in these cases transform a tree $\mathcal{T}(x)$ of rank r rooted at a four-node x into a tree $\mathcal{T}'$ of rank $\mathsf{r}-1$ whose root x' is a $\{1,1\}$-node. If x' is full, performing a cut promotion on $\mathcal{T}'$ results in another tree $\mathcal{T}''$ of rank r; similarly, if some child x'_i of x' is full, performing a cut promotion on $\mathcal{T}'(x'_i)$ and increasing the rank of x' by 1 also transforms $\mathcal{T}'$ into another tree $\mathcal{T}''$ of rank r; and if both children of x' are full, x' is also full, a case we already took care of. In all cases, this two-step transformation of $\mathcal{T}(x)$ into $\mathcal{T}''$ is a terminal operation.

If, however, neither x' nor its children is full, the rotation or demotion we performed in case 6, 7 or 8 did not create any full node. Indeed, replacing $\mathcal{T}(x)$ by a tree of smaller rank did not make any ancestor of x a full node. The only other nodes whose rank or induced sub-tree was changed are x' and its children, which we precisely assumed are not full either. In that case, our operation is transient if the parent of x, say y, was a $\{1,2\}$-node, and became a four-node.

▶ **Lemma 4.** *Each transient operation triggered by a deletion decreases the number of* $\{1,2\}$*-nodes, and does not increase the number of full nodes, also decreasing the tree potential.*

Proof. We just proved that transient operations are those where y was a $\{1,2\}$-node and neither its new child x' nor the children of x' became full. Hence, we did not create any full node, and it suffices to check, on a case-by-case basis, that the number of $\{1,2\}$-nodes decreased:

- **i)** in case 6, we destroyed two $\{1,2\}$-nodes ($\mathsf{B} = x_1$ and y);
- **ii)** in case 7, we destroyed at least two $\{1,2\}$-nodes ($\mathsf{B} = x_1$ and y) to create at most one $\{1,2\}$-node ($\mathsf{B} = x'_1$ or $\mathsf{F} = x'_2$, in case D used to be a $\{1,2\}$-node);
- **iii)** in case 8, we destroyed one $\{1,2\}$-node (y). ◀

4 Top-down updates

4.1 Overview and complexity analysis

Our top-down algorithms for inserting a key k in a tall enough AVL tree $\mathcal{T}$ and for deleting k from $\mathcal{T}$ follow the same general ideas, which were already successfully applied to weak AVL trees [5]. We will discover, in a top-down manner, the branch $x^0, x^1, \ldots$ below which k should be inserted or deleted, where x^0 is the root of $\mathcal{T}$ and each node x^{i+1} is a child of the previous node x^i. Using a $\mathcal{O}(1)$ look-ahead, and after discovering the first $\mathcal{O}(1)$ nodes on that branch, we must already decide what to do.

If we are lucky enough, we found a node x^i, for some integer $i \geqslant 1$, that is deemed *safe*: propagating a zero-edge (during insertions) or four-node (during deletions) that we would have planted along our branch far enough below x^i will stop when meeting x^i. When such a

node x^i exists among the top few nodes of the branch, we say that $\mathcal{T}$ was *friendly*; indeed, we should simply focus on updating the tree $\mathcal{T}(x^i)$, whose rank is smaller than $\mathcal{T}$ itself, and which will be transformed into a new tree of the same rank.

This fortunate case suggests focusing only on updating trees whose root is safe. Hence, if $\mathcal{T}$ has an unsafe root, a preliminary step will consist in transforming $\mathcal{T}$ into a tree whose root is safe, which we will then update; all subsequent (recursively called) updates will be performed on trees whose root is safe.

If, however, $\mathcal{T}$ has a safe root but is unfriendly, we will choose a deep enough node, say x^ℓ, and transform the sub-tree $\mathcal{T}(x^\ell)$ into a tree $\mathcal{T}'$ whose root is safe, while promoting it (during insertions) or demoting it (during deletions). This transformation will create a zero-edge or a four-node that will be propagated upward and whose propagation will stop when meeting the safe root x; afterward, it just remains to update $\mathcal{T}'$.

Once implemented, these ideas lead to a bottom-up updating algorithm, whose amortised complexity we study now.

▶ **Theorem 5.** *The updating algorithm whose skeleton we just presented, and whose details are presented in Sections 4.2 (for insertions) and 4.3 (for deletions) performs $\mathcal{O}(1)$ amortised write operations per update.*

Proof. Like for Theorem 1, we start from an empty tree, perform n insertion or deletion updates, and keep track of the variations of tree potential while these updates are performed. There is a large enough constant K such that each update requires:

1. optionally, making the tree root safe, which increases the tree potential by at most K;
2. identifying some safe nodes; this requires no write operation and does not change the tree potential;
3. operating on trees small height, i.e., whose height is smaller than a constant $h_{\min}$; this requires performing a constant number of write operations, and increases the tree potential by a at most K;
4. performing a certain number of batches, each of which consists in (i) cut-demoting a node to make it safe and make its ancestor a four-node; (ii) optionally, performing a cut rebalancing for ensuring that some node will stop propagating four-nodes; (iii) $\ell - \mathcal{O}(1)$ transient operations; and (iv) one "terminal" operation where the four-node stop being propagated.

In each batch, steps (i), (ii) and (iv) increase the tree potential by no more than a constant, and each transient operation decreases the tree potential. Thus, each batch decreases the tree potential by $\ell - \mathcal{O}(1)$, i.e., by no more than $\mathsf{L} - \ell$, where L is a constant; it also requires performing $\mathcal{O}(\ell+1)$ write operations. In particular, if $\ell \geqslant \mathsf{L}+1$, and if T batches of operations were performed in total, the potential increased by at most $n\mathsf{K} - \mathsf{T}$; of course, making sure that $\ell \geqslant \mathsf{L}+1$ means that we will need to determine a suitable constant L.

The tree potential started with the value 0, and ended with a non-negative value, which shows that $\mathsf{T} \leqslant 2n\mathsf{K}$. Therefore, in total, these n updates required performing no more than $\mathcal{O}(n) + \mathsf{T} \times \mathcal{O}(\mathsf{L}+1) \subseteq \mathcal{O}(n)$ write operations. ◀

Hence, it remains to:

1. define our notions of safe nodes, both for insertions and for deletions;
2. prove that trees rooted at an unsafe node can be cut promoted to make their root safe, during insertions;
3. prove that trees rooted at an unsafe node can be cut demoted to make their root safe, during deletions;

4. compute the constants hidden in the $\mathcal{O}(1)$ notations above, in order to effectively compute a constant ℓ for which batches decrease the tree potential.

Below, we will prove that choosing $\ell = 48$ during insertions, and $\ell = 126$ during deletions is enough for our purposes: each batch will decrease the tree potential by at least 1, and other operations will increase the tree potential by $\mathcal{O}(\ell)$ per update. These values of ℓ are grossly sub-optimal: for example, choosing $\ell = 15$ during insertions, and $\ell = 20$ during deletions would be enough; computing these better values, although not intrinsically difficult, involves lengthy case-by-case studies that we chose to avoid here.

4.2 Insertion

Below, we say that a node x is *insertion-safe* (or simply *safe*, when the context is clear) when (i) $\mathsf{r}(x) = 1$ and x is a $\{1, 2\}$-node, or (ii) $\mathsf{r}(x) \geqslant 2$ and x is not full. We showed in Section 3.2 that each tree rooted at a full node, i.e., each tree $\mathcal{T}$ of rank $\mathsf{r} \geqslant 2$ whose root is unsafe, can be subject to a cut promotion, resulting in a tree $\mathcal{T}'$ of rank $\mathsf{r} + 1$ whose root is a $\{1, 2\}$-node, thus being safe.

This makes following the recipe of Section 4.1 easy once the desired integer ℓ is chosen. First, if $\mathsf{r}(\mathcal{T}) \leqslant \ell + 2$, just use the bottom-up algorithm of Section 3.3, which may result in a tree $\mathcal{T}'$ of rank $\mathsf{r}(\mathcal{T}') = \mathsf{r}(\mathcal{T}) + 1$ only in case $\mathcal{T}$ was rooted at an unsafe node. Otherwise, $\mathsf{r}(\mathcal{T}) \geqslant \ell + 3$, and:

1. if $\mathcal{T}$ is rooted at an unsafe node, cut-promote it, thus obtaining a tree $\mathcal{T}'$ of rank $\mathsf{r}(\mathcal{T}') \geqslant \ell + 4$ whose root is safe;
2. if $\mathcal{T}$ is friendly, let $i \leqslant \ell$ be the least positive integer for which x^i is safe: we simply focus on inserting the key k in $\mathcal{T}(x^i)$;
3. if $\mathcal{T}$ is unfriendly, the first nodes $x^1, \dots$ on our branch are all unsafe; thus, they are $\{1, 1\}$-nodes, and the branch contains at least ℓ nodes, with the guarantee that $\mathsf{r}(x^\ell) \geqslant \mathsf{r}(\mathcal{T}) - (\ell + 1) \geqslant 2$, so that all nodes $x^1, \dots, x^\ell$ are full; hence we cut-promote $\mathcal{T}(x^\ell)$ and promote all nodes $x^1, \dots, x^{\ell-1}$, thereby creating a zero-edge between x^0 and x^1, which we eliminate by using the cut rebalancing prescribed in Section 3.3.

▶ **Lemma 6.** *Step 3 consists in a batch operation that decreases the tree potential by $\ell - 47$ or more.*

Proof. It suffices to decompose step 3 in atomic operations:

- **i)** a cut promotion, which increases the tree potential by 14 or less;
- **ii)** replacing each of the $\ell - 1$ full nodes $x^1, \dots, x^{\ell-1}$ by a $\{1, 2\}$-node, which decreases the tree potential by $\ell - 1$;
- **iii)** eliminating a zero-edge through a terminal cut rebalancing performed on a cut with 2 to 4 nodes, and optionally followed by a cut demotion; this increases the tree potential by 32 or less. ◀

4.3 Deletion

We will now characterise *deletion-safe* (or simply *safe*, when the context is clear) nodes: a node x should be safe when they can stop propagating four-nodes. There will be three families of safe nodes: (i) *surely safe* nodes, which stop propagating four-nodes even when using the textbook deletion algorithm, (ii) *barely safe* nodes, which are some of the nodes that can be transformed to create surely safe nodes in their close vicinity, and (iii) *simply safe* nodes, which stop propagating four-nodes when using the algorithm of Section 3.4.

Surely safe nodes are the nodes x of rank $\mathsf{r}(x) \geqslant 1$ with profile $(1,1)$, or rank $\mathsf{r}(x) \geqslant 2$ and profile $(2,2,2)$. Indeed, if some child x_i of x is to be replaced by a node of smaller rank, either (i) x had profile $(1,1)$, in which case its new profile is simply $\{1,2\}$, or (ii) x had profiles $\{1,2\}$ and $(2,2,2)$, in which case x_i was the 2-child of x (because the 1-child of x had profile $(1,1)$), and we fall in case 5 of the bottom-up algorithm.

Detecting other kinds of safe nodes is more challenging; in particular, whether a node x is safe may depend on which descendants of x are to be replaced by nodes of smaller rank. More precisely, let $x = x^\ell$ be a node that belongs to the branch $x^0, x^1, \ldots$ below which the key k will be deleted, and let $z = x^{\ell+4}$.

▶ **Lemma 7.** *Each tree $\mathcal{T}$ can be subject to a cut rebalancing, called* cut saving*, that yields a tree $\mathcal{T}'$ of rank $\mathsf{r}(\mathcal{T})$ or $\mathsf{r}(\mathcal{T})-1$ in which z has a surely safe ancestor. This cut saving increases the tree potential by no more than* 38*.*

Proof. Let x be the root of $\mathcal{T}$. If x is surely safe already, there is nothing to do. Hence, we assume that x has profiles $\{1,2\}$ and $\{2,2,3\}$. Replacing its descendants of rank $\mathsf{r}(x)-2$ by their children gives a cut $(\mathsf{A},\mathsf{C},\mathsf{E},\mathsf{G},\mathsf{I})$ with profile $\{3,3,3,3,3\}$, $\{3,3,3,3,4\}$ or $\{3,3,3,4,4\}$.

In the first case, the children of x must have profiles $(1,1)$ and $(2,2,2)$, and once again, there is nothing to do. In the third case, cut-demoting $\mathcal{T}(x)$ transforms it into a tree $\mathcal{T}'$ of rank $\mathsf{r}(x)-1$ whose root is a $\{1,1\}$-node.

It remains to treat the second case. Let y be the node of rank $\mathsf{r}(x)-4$ in the cut $(\mathsf{A},\mathsf{C},\mathsf{E},\mathsf{G},\mathsf{I})$, and let B, D, F and H be the cut ancestors. By symmetry, we assume that y coincides with A, C or E; if $y = \mathsf{E}$, that z descends from E, G or I; and if $y = \mathsf{E}$ and z descends from E, there are no more $\{1,2\}$-nodes among A and C than among G and I. Then, we distinguish ten cases; the tree $\mathcal{T}'$ will be of rank $\mathsf{r}(x)$ in cases 1 to 8, and $\mathsf{r}(x)-1$ in cases 9 and 10:

1. If (i) $y = \mathsf{A}$ or $y = \mathsf{C}$ and (iii) z descends from E, G or I, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_1$); this makes F a surely safe ancestor of z.
2. If (i) $y = \mathsf{A}$, (ii) C is a $\{1,1\}$-node and (iii) z descends from A or C, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_1$); this makes B a surely safe ancestor of z.
3. If (i) $y = \mathsf{A}$, (ii) E is a $\{1,1\}$-node (whereas C is not), and (iii) z descends from A or C, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_2$); this makes C a surely safe ancestor of z.
4. If (i) $y = \mathsf{C}$, (ii) A is a $\{1,1\}$-node, and (iii) z descends from A or C, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_1$); this makes B a surely safe ancestor of z.
5. If (i) $y = \mathsf{C}$, (ii) E is a $\{1,1\}$-node (whereas A is not), and (iii) z descends from A or C, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_4$); this makes B a surely safe ancestor of z.
6. If (i) $y = \mathsf{E}$ and (iii) z descends from G or I, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_5$); this makes H a surely safe ancestor of z.
7. If (i) $y = \mathsf{E}$, (ii) C is a $\{1,1\}$-node, and (iii) z descends from E, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_5$); this makes D a surely safe ancestor of z.
8. If (i) $y = \mathsf{E}$, (ii) A is a $\{1,1\}$-node (whereas C is not), and (iii) z descends from E, we rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_6$); this makes F a surely safe ancestor of z.
9. If (i) $y = \mathsf{A}$ or $y = \mathsf{C}$, (ii) all of A, C and E coincide with y or are $\{1,2\}$-nodes, and (iii) z descends from A or C, we first rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_3$). The sub-tree $\mathcal{T}_3(\mathsf{D})$ has profile $\{3,3,3,4,4\}$ and we cut-demote it, before demoting the node F itself; this yields a tree of rank $\mathsf{r}(x)$-1 whose root F is a $\{1,1\}$-node.
10. If (i) $y = \mathsf{E}$, (ii) all of A, C, G and I are $\{1,2\}$-nodes, and (iii) z descends from E, we first rebalance $\mathcal{T}(x)$ as shown in Figure 4 (tree $\mathcal{T}_5$). The sub-tree $\mathcal{T}_5(\mathsf{B})$ has profile $\{3,3,3,4,4\}$ and we cut-demote it, before demoting the node F itself; this yields a tree of rank $\mathsf{r}(x)$-1 whose root F is a $\{1,1\}$-node.

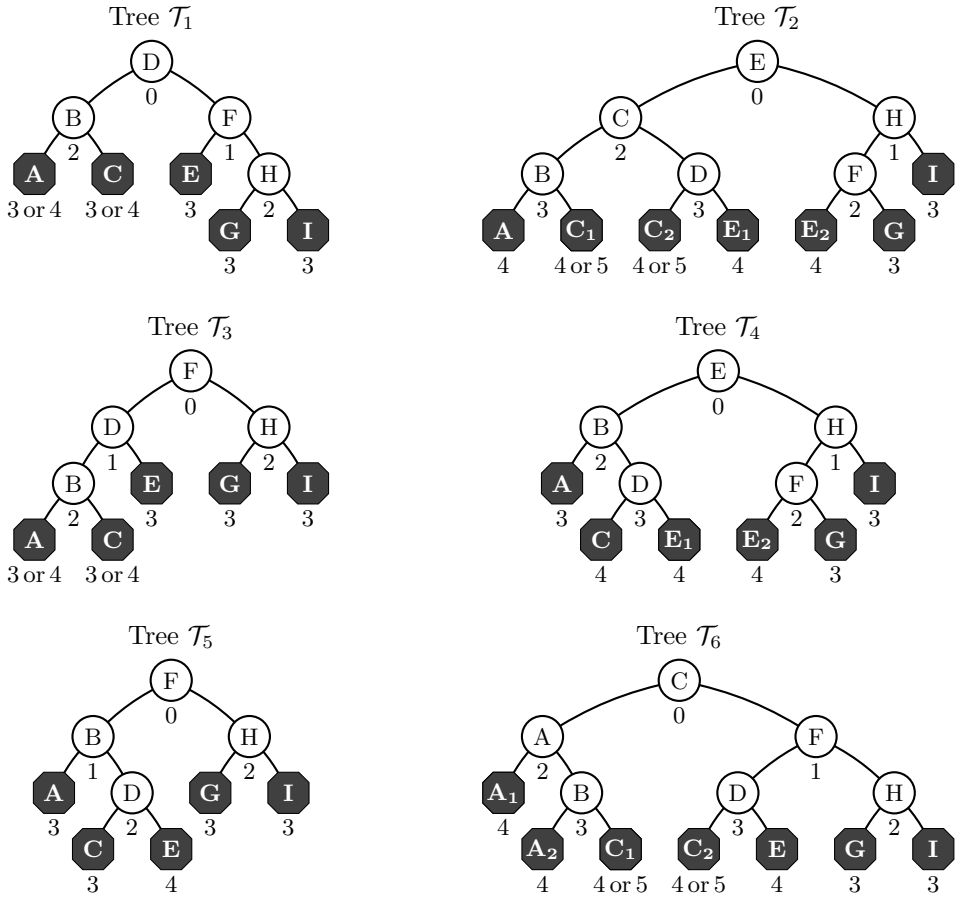

Figure 4 Trees the saving procedure may result in, or use as intermediate step (in cases 9 and 10).

All cases consist in performing a cut rebalancing operation on a cut with at most 7 nodes, optionally followed by a cut demotion. Hence, the saving operation increases the tree potential by no more than 38. ◀

We can now present two additional notions of safety. First, a node x is said to be *barely safe* when the cut saving transforms $\mathcal{T}(x)$ into a tree $\mathcal{T}'(x')$ of rank $\mathsf{r}(x)$. Second, as already mentioned, a node is said to be *simply safe* if its stops propagating four-nodes. Fortunately, deciding whether a node is simply safe is easy, as outlined by the following result.

▶ **Lemma 8.** *Being simply safe depends on local criteria only.*

Proof. Provided that the node $x = x^\ell$ is not surely safe, and if the node $t = x^{\ell+1}$ has just been replaced by a node of smaller rank, the textbook algorithm must now replace $\mathcal{T}(x)$ by a tree $\mathcal{T}'$ whose root x' is a $\{1,1\}$-node with rank $\mathsf{r}(x') = \mathsf{r}(x) - 1$. Furthermore, as can be seen in Figure 2, at least one of its children x'_1 and x'_2 will also be a $\{1,1\}$-node. Hence, we say that x is *m-strong* when the child x'_m is a $\{1,1\}$-node.

Being 1- or 2-strong depends on local criteria only. Indeed, the node x is 1-strong when:

- t coincides with x_1, and x has profile $(1,2)$, $(2,3,2)$, $(2,3,3,3)$ or $(2,3,4,3)$; or
- t coincides with x_2, and x has profile $(3,3,1)$, $(3,3,3,2)$ or $(3,3,4,2)$.

Hence, being 1-strong is a purely local criterion and, symmetrically, so is being 2-strong.

Then, knowing whether t is 1-strong and/or 2-strong allows checking whether x itself is safe, based on other local criteria. For instance, when t coincides with x_2, the node x is safe when it has profile $(1,1)$ or $(2,2,2)$, or in the following cases:

6. $\mathsf{r}(x_{11}) = \mathsf{r}(x_2) = \mathsf{r}-2$ and $\mathsf{r}(x_{12}) = \mathsf{r}-3$, and
  a) x has profile $(3,4,4,3,2)$, $(3,3,4,4,2)$ or $(2,4,5,5,2)$, or
  b) x has profile $(2,4,4,2)$ and x_2 is 1-strong;

7. $\mathsf{r}(x_{11}) = \mathsf{r}-3$ and $\mathsf{r}(x_{12}) = \mathsf{r}(x_2) = \mathsf{r}-2$, and
  a) x has profile $(3,4,4,3,2)$, $(3,3,4,4,2)$, $(4,5,5,4,4,3,2)$, $(4,5,5,4,4,4,2)$, $(4,4,5,5,4,3,2)$, $(4,4,5,5,4,4,2)$, $(3,3,4,5,5,2)$ or $(3,4,4,5,5,2)$, or
  b) x has profile $(3,3,4,4,2)$ or $(3,4,4,4,2)$ and x_2 is 1-strong;

8. $\mathsf{r}(x_1) = \mathsf{r}-2$ and $\mathsf{r}(x_2) = \mathsf{r}-1$, and
  a) x has profile $(3,4,4,1)$, or
  b) x has profile $(3,3,1)$ and x_2 is 1-strong.

These cases were listed by simply enumerating all conditions that could leaf to creating a full node while cut-rebalancing a tree in cases 6 to 8. Hence, some of them are redundant; for instance, in case 7a, if x has profile $(3,3,4,5,5,2)$, it must also have had profile $(3,3,4,4,2)$, which was sufficient for flagging it as safe. Symmetric characterisations exist when t coincides with the left child x_1 of the node x. ◀

Safe nodes and the above saving procedure allow us to implement the recipe succinctly described in Section 4.1. If $\mathcal{T}$ is of rank $\mathsf{r}(\mathcal{T}) \leqslant 2\ell+4$ and its root is barely safe, first use the saving procedure to let a surely safe node appear on the branch below which k will be deleted, and then use the bottom-up algorithm of Section 3.4; this will result in a tree $\mathcal{T}'$ of rank $\mathsf{r}(\mathcal{T}') = \mathsf{r}(\mathcal{T})$. Otherwise, if $\mathcal{T}$ is of rank $\mathsf{r}(\mathcal{T}) \leqslant 2\ell+4$ and its root is *not* barely safe, directly use the bottom-up algorithm of Section 3.4. It may result in a tree $\mathcal{T}'$ of rank $\mathsf{r}(\mathcal{T}') = \mathsf{r}(\mathcal{T}) - 1$ only when $\mathcal{T}$ was rooted at an unsafe node.

We focus now on trees $\mathcal{T}$ of rank $\mathsf{r}(\mathcal{T}) \geqslant 2\ell+5$:

1. If $\mathcal{T}$ is rooted at an unsafe node, the saving procedure transforms it into a tree $\mathcal{T}'$ of rank $\mathsf{r}(\mathcal{T}') \geqslant 2\ell+4$ whose root is safe.

2. If $\mathcal{T}$ is friendly, let $i \leqslant \ell$ be the least positive integer for which x^i is safe: we simply focus on inserting the key k in $\mathcal{T}(x^i)$.

3. If $\mathcal{T}$ is unfriendly, we first consider two sub-cases: if the root x^0 of $\mathcal{T}$ is surely or simply safe, we set $\alpha = x^0$; otherwise, applying the saving algorithm to $\mathcal{T}$ results, without altering the sub-tree $\mathcal{T}(x^4)$, in a tree $\mathcal{T}'$ of rank $\mathsf{r}(\mathcal{T})$ in which x^4 has a surely safe ancestor, the least of which we call α. Then, observing that $\mathsf{r}(x^\ell) \geqslant \mathsf{r}(x) - 2\ell \geqslant 4$, we apply the saving procedure to the sub-tree $\mathcal{T}(x^\ell)$, which makes $x^{\ell-1}$ a four-node. We propagate once this four-node by using the textbook deletion algorithm, i.e., we do not cut-promote full-nodes that might result from using the textbook algorithm; then, we keep propagating our four-node using the algorithm from Section 3.4, a propagation that will stop only when reaching the node α, at which point the four-node is eliminated.

▶ **Lemma 9.** *Step 3 consists in a batch operation that decreases the tree potential by $\ell - 125$ or more.*

Proof. Like for Lemma 4, it suffices to decompose step 3 in atomic operations:

i) optionally, using the saving procedure (cases 1 to 8) on $\mathcal{T}$, which increases the tree potential by no more than 38;

ii) cut-demoting $\mathcal{T}(x^\ell)$ or using the saving procedure (case 9 or 10) on $\mathcal{T}(x^\ell)$, which increases the tree potential by no more than 38;

iii) propagating once the four-node by using the textbook deletion algorithm on $\mathcal{T}(x^{\ell-1})$: this consists in rebalancing a cut with at most 4 nodes, thus increasing the tree potential by no more than 12;

iv) propagating the four-node by using the algorithm of Section 3.4 on nodes $x^{\ell-2}, \dots, x^4$, and stopping just before one meets the surely safe ancestor α of x^4; there are at least $\ell-5$ such propagations, hence the tree potential decreases by at least $\ell - 5$;

v) eliminating the four-node by using the algorithm of Section 3.4 on α; this consists in rebalancing a cut with at most 4 nodes, then possibly performing a cut promotion, and even a node promotion afterward; these three phases, two of which are optional, increase the tree potential by at most 12, 14, and 6, respectively. ◀

References

1 Georgii Adel'son-Velskii and Evgenii Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146(2), pages 263–266. Russian Academy of Sciences, 1962.

2 Mahdi Amani, Kevin Lai, and Robert Tarjan. Amortized rotation cost in AVL trees. *Information Processing Letters*, 116(5):327–330, 2016. doi:10.1016/J.IPL.2015.12.009.

3 Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980. doi:10.1016/0304-3975(80)90018-3.

4 Leonidas Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.

5 Bernhard Haeupler, Siddhartha Sen, and Robert E Tarjan. Rank-balanced trees. *ACM Transactions on Algorithms (TALG)*, 11(4):1–26, 2015. doi:10.1145/2689412.

6 Thomas Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM*, 9(1):13–28, 1962. doi:10.1145/321105.321108.

7 Vincent Jugé. Grand-children weight-balanced binary search trees. In *19th International Symposium on Algorithms and Data Structures (WADS)*, volume 349 of *LIPIcs*, pages 14:1–14:20, 2025. doi:10.4230/LIPICS.WADS.2025.14.

8 Tony Lai and Derick Wood. A top-down updating algorithm for weight-balanced trees. *International Journal of Foundations of Computer Science*, 4(4):309–324, 1993. doi:10.1142/S0129054193000201.

9 Jürg Nievergelt and Edward Reingold. Binary search trees of bounded balance. In *4th Annual ACM Symposium on Theory of Computing (STOC)*, pages 137–142. ACM, 1972. doi:10.1145/800152.804906.

10 Hendrik Olivié. *A study of balanced binary trees and balanced one-two trees*. Universitaire Instelling Antwerpen (Belgium), 1980.

11 Hendrik Olivié. A new class of balanced search trees: half-balanced binary search tress. *RAIRO. Informatique théorique*, 16(1):51–71, 1982. doi:10.1051/ita/1982160100511.

12 Daniel Sleator and Robert Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.