Crumple Trees

Coleman Nugent
Department of Computer Science
California State University, Fullerton
Fullerton, California, 92831, USA
Email: colemannugent@csu.fullerton.edu

Kevin A. Wortman
Department of Computer Science
California State University, Fullerton
Fullerton, California, 92831, USA
Email: kwortman@fullerton.edu

***Abstract*—We introduce the *crumple tree,* an alternative derivation of the wavl binary search tree that facilitates discovery learning. The crumple tree is distinguished by features that aid teaching and learning: its balance invariant corresponds to the intuitive notion of stretching or crumpling yarn; it is straightforward to draw, digitally or by hand, in both balanced and imbalanced states; the rebalancing operations derive organically from the invariant and drawings, and could be discovered by students themselves; both insertion and deletion are straightforward enough to cover in an undergraduate course; and implementing a crumple tree is an achievable undertaking for undergraduate students.**

***Index Terms*—data structures, tree data structures, computer science education**

I. Introduction

Self-balancing binary search trees (BSTs) are a fixture in CS2 courses, but can be the most challenging fundamental data structure for instructors and students alike. This is unsurprising, as classical BSTs such as AVL trees [1] and red-black trees [2] were not designed with pedagogy in mind, but rather to optmize asymptotic efficiency, constant factors, and elegance (as understood by algorithm researchers). Other kinds of BSTs include 2-3 trees [3], AA trees [4], B-trees [5], skip lists [6], splay trees [7], treaps [8], and ravl trees [9]. All BSTs are search trees with a *balance invariant* that ensures $O(\log n)$ height, and a *rebalancing algorithm* that restores the balance invariant after an insertion or deletion operation. Covering these established BSTs in CS2 invariably runs afoul of one or more pitfalls: the invariant may be abstract and unintuitive; it may involve coloring nodes, which is difficult to draw on paper and is inaccessible to students with visual impairments; the analysis may involve mathematics beyond the scope of CS2; imbalance conditions may be difficult to draw convincingly; the rebalancing algorithms may seem arbitrary, or in the case of deletion, too complicated to discuss at all; and the structure may be too complex for students to implement. Broaching a topic and then immediately pronouncing that it is too advanced for students is counterproductive, as that may exacerbate fixed mindset [10] [11] [12] or stereotype threat [13] [14] [15] [16].

We introduce *crumple trees,* a BST variant that is designed with the express purpose of supporting pedagogy by sidestepping the aforementioned pitfalls. Crumple trees are an alternative way of defining, describing, and deriving *weak AVL (wavl) trees* [17], with novel terminology and visual notation. The wavl tree boasts both the shallow height of the AVL tree and infrequent rotations of the red-black tree, and represents progress toward a pedagogical BST, as its rebalancing algorithms are comparatively simple. However, wavl trees are presented as a special case of the *rank-balanced tree* taxonomy [17], which would be a challenging and time-intensive digression for a typical CS2 course. By contrast, crumple trees are defined by intuitive invariants, and their rebalancing algorithms can be derived from a straightforward deductive reasoning exercise using those invariants as initial premises. Therefore the entire insertion *and deletion* algorithms could be presented as an accessible lecture in CS2; or better yet, an instructor could explain how to derive the rebalancing cases and then students could derive the rest themselves as an active learning [18] [19] or discovery learning [20] activity.

978-1-7281-3783-4/20/$31.00 ©2020 IEEE

II. Crumple Trees

We begin with a formal definition for a crumple tree that is coherent enough to be drawn, but may or may not be balanced.

Definition 1: A *drawable crumple tree* t is either

1) an *external (null) node* $t = \square$, devoid of elements; or
2) an *internal (parent) node* $t = (x, l, L, R)$ where x is an element from an ordered universe, l is a positive integer *level,* and L and R are each drawable crumple trees.

The *height* of t is

$$h(t) = \begin{cases} 0 & \text{if } t = \square \\ \max(1 + h(L), 1 + h(R)) & \text{if } t = (x, l, L, R), \end{cases}$$

and the *level* of t is

$$\ell(t) = \begin{cases} 0 & \text{if } t = \square \\ l & \text{if } t = (x, l, L, R). \end{cases}$$

Every drawable crumple tree obeys three invariants:

1) *order invariant:* any element w in L obeys $w < x$, and any element y in R obeys $y > x$;
2) *upwards invariant:* every internal node $t = (x, l, L, R)$ satisfies $\ell(t) \geq \ell(L)$ and $\ell(t) \geq \ell(R)$; and
3) *bottom invariant:* if t is a *terminal node* with $L = R = \square$, then $\ell(t) = 1$.

A crumple tree is balanced when the length of every edge, in terms of levels, is either one or two.

Definition 2: A *balanced crumple tree* is a drawable crumple tree t that also satisfies the *balance invariant:* for every edge

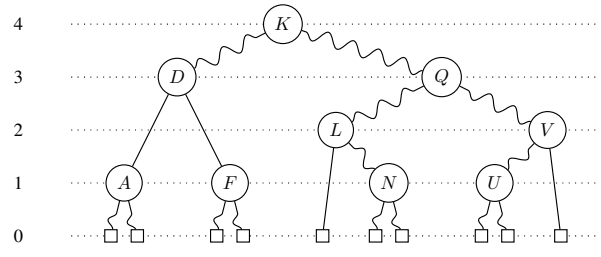

Fig. 1. A balanced crumple tree storing nine letters. Level numbers are written to the left.

e from parent p to child c in t, $|e| \in \{1, 2\}$, where $|e| \equiv \ell(p) - \ell(c)$.

Observe that the upwards invariant implies that every such $|e| \geq 0$.

An edge of length 0, 1, 2, and 3 is termed *crumpled, bent*, *straight,* and *stretched* respectively. So every edge in a balanced crumple tree is either bent or straight. If parent node p has left and right edges e_L, e_R, we summarize the *shape* of p as the pair $\big(|e_L|, |e_R|\big)$. So in a balanced tree, every parent node's shape is one of $(1,1), (1,2), (2,1),$ or $(2,2)$. A tree that contains a crumpled or stretched edge is *imbalanced.*

We visualize balancing a crumple tree as akin to rearranging tacks that are connected by stretchable string. The rebalancing process proceeds in bottom-up order, so at each level some higher nodes may be rearranged, while their descendants are already rebalanced and their levels and positions are *fixed*. We can move the non-fixed nodes vertically to different levels, and add or remove edges, but are constrained by the four invariants above. This sets the stage for discovering the rebalancing operations as a closed-ended problem-solving exercise.

A crumple tree is drawn with a faint horizontal line for each visible level; nodes are aligned vertically on these lines; a balanced internal node is a circle; a balanced external node is a small square; an imbalanced node is a regular polygon or star; and crumpled, bent, straight, and stretched edges are zigzag, wavy, straight, and dashed lines, respectively. Figure 1 shows a balanced crumple tree.

The algorithm for searching in a plain binary search tree carries over to a crumple tree; the levels are irrelevant.

III. Insertion

As with other BSTs, inserting a new element q into a balanced crumple tree involves three conceptual phases.

1) *Search phase:* Starting from the root, search downward until an external node $z = \square$ is reached on level 0.
2) *Node creation phase:* Replace z with a new internal node $z' = (q, 1, \square, \square)$, i.e. a node containing q on level 1 with bent edges to external nodes on level 0. We say z' is *rising* because its level is one more than that of the subtree it replaces; this implicitly decrements the length of the edge above z' (if any). We have incorporated q, consistent with the order invariant; but if z' was not the root and the edge above z' used to be bent, that edge is now crumpled and the tree is imbalanced.
3) *Rebalance phase:* As long as some subtree f is imbalanced, perform a *local restructuring* operation that balances f. If a restructuring does not raise f then this phase ends, otherwise it continues at a higher level.

The search and node creation phases are conventional, so we turn to the rebalancing algorithm. Rather than dictate, we derive together. We employ divide-and-conquer to break this process into separate cases based on the shape of f before the node creation phase. Descendants of f may be involved: $a, b, c, d, e, g, h,$ and i. Restructuring progresses upward, so the deepest relevant nodes are fixed and cannot move; we must deduce what to do with the non-fixed nodes, and should stop rising as soon as possible. Figure 4 illustrates all five cases, showing f before rebalancing (i.e. immediately after node creation) and then after successful rebalancing. (Though we invite the reader to follow our exploration before studying the solutions.) In that figure, an arc labeled x or y denotes both an edge and the node below it, which are moved as one unit. We discuss only cases where f's left child b is rising, but the symmetric cases are also depicted in Figure 4.

1) f's left edge was straight, so f was $(2,1)$ or $(2,2)$. We simply allow b to rise, making f's left edge bent.

If we are not in case 1, then we know that f's left edge was bent.

2) f was $(1,1)$. As shown in Figure 2, nodes b and j are fixed, so our only recourse is to change f. We look to the invariants for guidance. Per the order invariant, f is between b and j horizontally; per the upwards invariant, f must be on level $\ell(j)+2$ or above; the bottom invariant is irrelevant because b exists so f is non-terminal; the balance invariant on the b–f edge says f that can only be on level $\ell(j)+2$ or $\ell(j)+3$; and the balance invariant on the j–f edge says that f can only be on level $\ell(j)+1$ or $\ell(j)+2$. The only way to satisfy all these constraints is to place f on level $\ell(j)+2$, as the parent of b and j, with shape $(1,2)$, as shown in Figure 4.

If we are neither in case 1 nor case 2, we know f was $(1,2)$ and is now $(0,2)$. Applying invariants to b, f, j as in case 2 exposes a contradiction: f must be above b but the f–j edge may not be stretched, so we cannot rebalance solely by moving f. Intuitively, the problem is that the left subtree of f has too many nodes relative to the right subtree. This problem becomes solvable if we expose b's children a and d; the remaining cases are categorized by the shape of b.

3) b was $(1,2)$. As shown in Figure 3, a, d, j are fixed and b, f are non-fixed. Per the order invariant, b must be between a and d and f must be between d and j horizontally; per the upwards invariant b, f must be on or above level $\ell(j)+2, \ell(j)+1$ respectively; per the balance invariant f may not be raised as long as it is j's parent; so we have no choice but to rearrange edges so that b becomes f's parent. We could put b on level $\ell(j)+3$ or $\ell(j)+2$, and choose $\ell(j)+2$ to stop the rise.

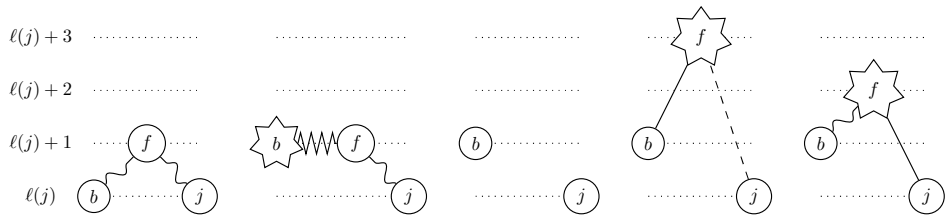

Fig. 2. Insertion case 2 before node creation; after node creation; before relocating f; and the two alternative shapes.

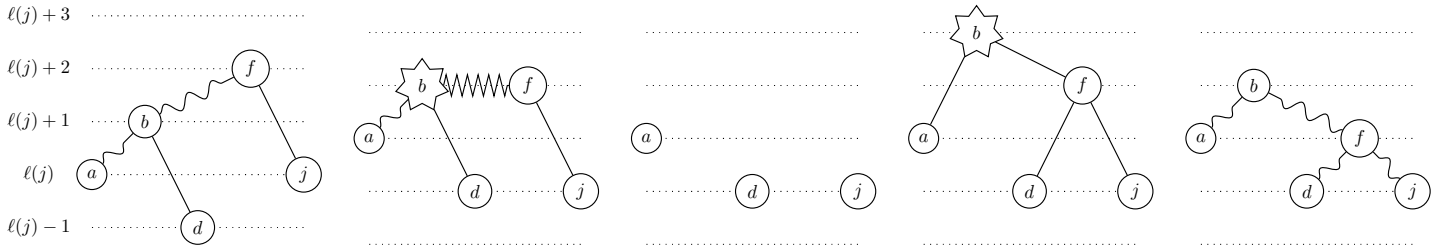

Fig. 3. Insertion case 3 before node creation; after node creation; before relocating b and f; and the two alternative shapes.

An astute reader will recognize that we rotated clockwise around f; we have derived rotation from first principles! The only way to solve this puzzle is to discover rotation.

The remaining cases are abridged due to space limitations.

4) b is $(1,1)$. Nodes $a, d,$ and j are fixed; we must rotate clockwise around f again; the only way to avoid crumpling the d–f or f–b edge is to raise b.
5) b is $(2,1)$. Nodes a and j are fixed, as are the left and right subtrees of d; we double-rotate around f; and can absorb the rise by making all the new edges bent.

These five cases are exhaustive. Perhaps surprisingly, there is no sixth case when b is $(2,2)$ because $(2,2)$ nodes never rise: observe that neither the node creation phase nor any case 1–5 raises a $(2,2)$ node.

IV. Deletion

Deletion also involves three conceptual phases.

1) *Downward phase:* Search for the node z to be deleted; when that node is not terminal, it is swapped with its inorder predecessor or successor, so in any event z is terminal after this phase.
2) *Node destruction phase:* Destroy z and replace it with an external node, which is considered *falling* since its level is one less than the corresponding subtree before node destruction.
3) *Rebalancing phase:* Analogous to insertion, we perform local restructurings bottom-to-top until the falling condition is either eliminated or propagates past the root.

As with insertion, we break the restructuring operation into discrete cases, and the solution to each case can be deduced from the invariants and a sketch of the fixed nodes. Deletion is marginally more complicated than insertion because the bottom invariant becomes relevant. If, after node destruction, we are left with a terminal $(2,2)$ node, that node must fall to become $(1,1)$. It turns out that only cases 1 and 4 need subcases "1B" and "4B" to handle the bottom invariant this way. Figure 5 shows all the cases, which are summarized below.

1) f was $(1,1)$ or $(1,2)$. In this case, straightening f's left edge absorbs the fall. If f is not a $(2,2)$ terminal after this step, we are done (case 1A). Otherwise f falls and becomes $(1,1)$ (case 1B).
2) f was $(2,2)$. When b fell, it stretched f's left edge so f is $(3,2)$; f falls to become $(2,1)$.
3) j is $(1,1)$. Single-rotate j counter-clockwise into root position.
4) j is $(2,1)$. This is the same as case 3, except that h is one level lower so its parent edge is straight instead of bent (case 4A). Further, when nodes b and h are external, j must fall to prevent f from becoming a terminal $(2,2)$ node. In this case the b–f, h–f, and k–j edges bend (case 4B).
5) j is $(1,2)$. Double-rotate h counter-clockwise into root position.
6) j is $(2,2)$. We lower f and bend both of j's edges, but do not rotate.

V. Analysis

Crumple trees are isomorphic to wavl trees; a crumple tree t on level $\ell(t)$ corresponds to a wavl tree of rank $r = \ell(t)$. Therefore, regardless of the name, such a tree has height at most $\log_\varphi n \approx 1.44 \log_2 n \in O(\log n)$, where φ is the golden ratio [17]. We can use a rudimentary argument suitable for CS2 to obtain a looser, yet asymptotically equivalent, logarithmic bound.

Lemma 1: For any balanced crumple tree t containing n elements, the height of t is $O(\log n)$.

Proof 1: The lemma is trivially true when $n = 0$, so we discuss only the case when $n > 0$ and the root t is an internal node. We first bound the maximum level $\ell(t)$ of a crumple

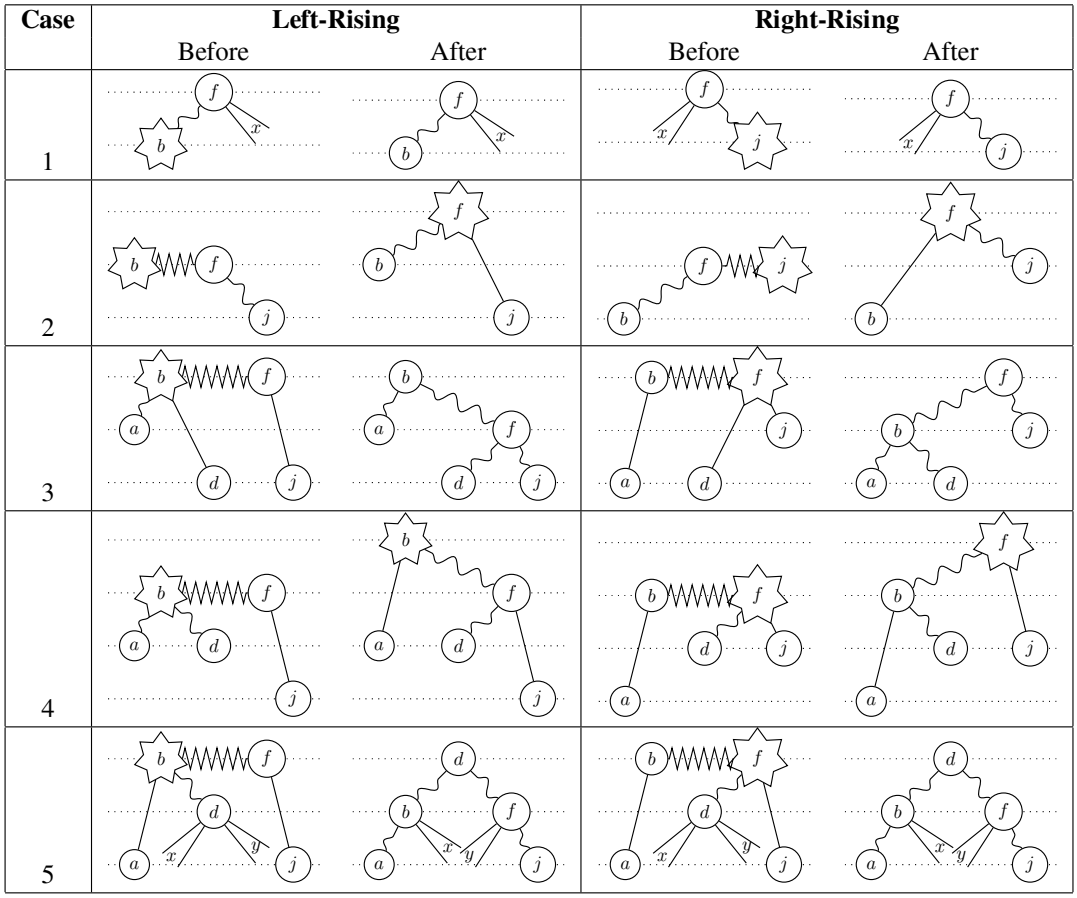

Fig. 4. Rebalancing after insertion.

tree with respect to n, and then relate the level of an arbitrary tree to its height.

In a balanced crumple tree each edge is bent or straight, and straight edges add more to the level of the root. Suppose without loss of generality that n is one less than a power of two, and consider a tree in which every edge is straight. Level $\ell(t)$ has exactly one node; level $\ell(t)-2$ has exactly two nodes; level $\ell(t)-4$ has exactly four nodes; and so on until we reach the terminal nodes on level 1. This tree has as many nodes as a perfect binary tree of height $(\ell(t)+1)/2$, which is $n = 2^{(\ell(t)+1)/2}-1$. Solving for $\ell(t)$ gives $\ell(t) = 2\cdot\log_2(n+1)-1$.

Now let t be an arbitrary balanced crumple tree; we will see that $h(t) \leq \ell(t)$. Let $p = (x, l, L, R)$ be some internal node, and e_L, e_R be the edges between p and L, R respectively. Recall that the height of p is $h(p) = \max(1+h(L), 1+h(r))$ and the level of p satisfies $\ell(p) \geq \ell(L) + |e_L|$ and $\ell(p) \geq \ell(R)+|e_R|$. By assumption t is balanced so $|e_L|, |e_R| \in \{1, 2\}$. So every edge in the deepest path of t contributes 1 to t's height, and either 1 or 2 to t's level, and $h(t) \leq \ell(t)$.

Therefore $h(t) \leq \ell(t) \leq 2 \cdot \log_2(n+1) - 1$, and t's height is $O(\log n)$.

Finally we summarize the efficiency of crumple trees.

Corollary 1: Crumple trees have $O(n)$ space complexity, and support search, insertion, and deletion operations in $O(\log n)$ worst-case time each.

Proof 2: A crumple tree of n elements stores exactly n internal nodes and at most $n+1$ external nodes, each of which takes $O(1)$ space, so the total space complexity of the tree is $O(n)$. The search, insertion, and deletion operations each involve $O(1)$ phases; each phase spends $O(1)$ time on each height layer; and there are $O(\log n)$ height layers; so these operations take $O(\log n)$ worst-case time each.

VI. Implementation

To substantiate that students can implement crumple trees, an undergraduate student (and coauthor) developed two implementations, one in Python and the other in C++. We found this exercise to be straightforward. The process reinforces and motivates programming language constructs taught in CS1: the edge types (e.g. crumpled) can be represented by an enumerated type; insertion and deletion can be implemented either with recursion or loops; and local restructuring is a straightforward outer if statement to decide between left and right, and an inner if/else chain to discern between the numbered cases. The canonical numbering for the cases helps make that if/else chain readable; the terminology for the rising/falling conditions, edge types, and node shapes (e.g. $(1, 2)$) encourages clear identifier names; and naming the invariants (e.g. order invariant) facilitates writing expressive error checks

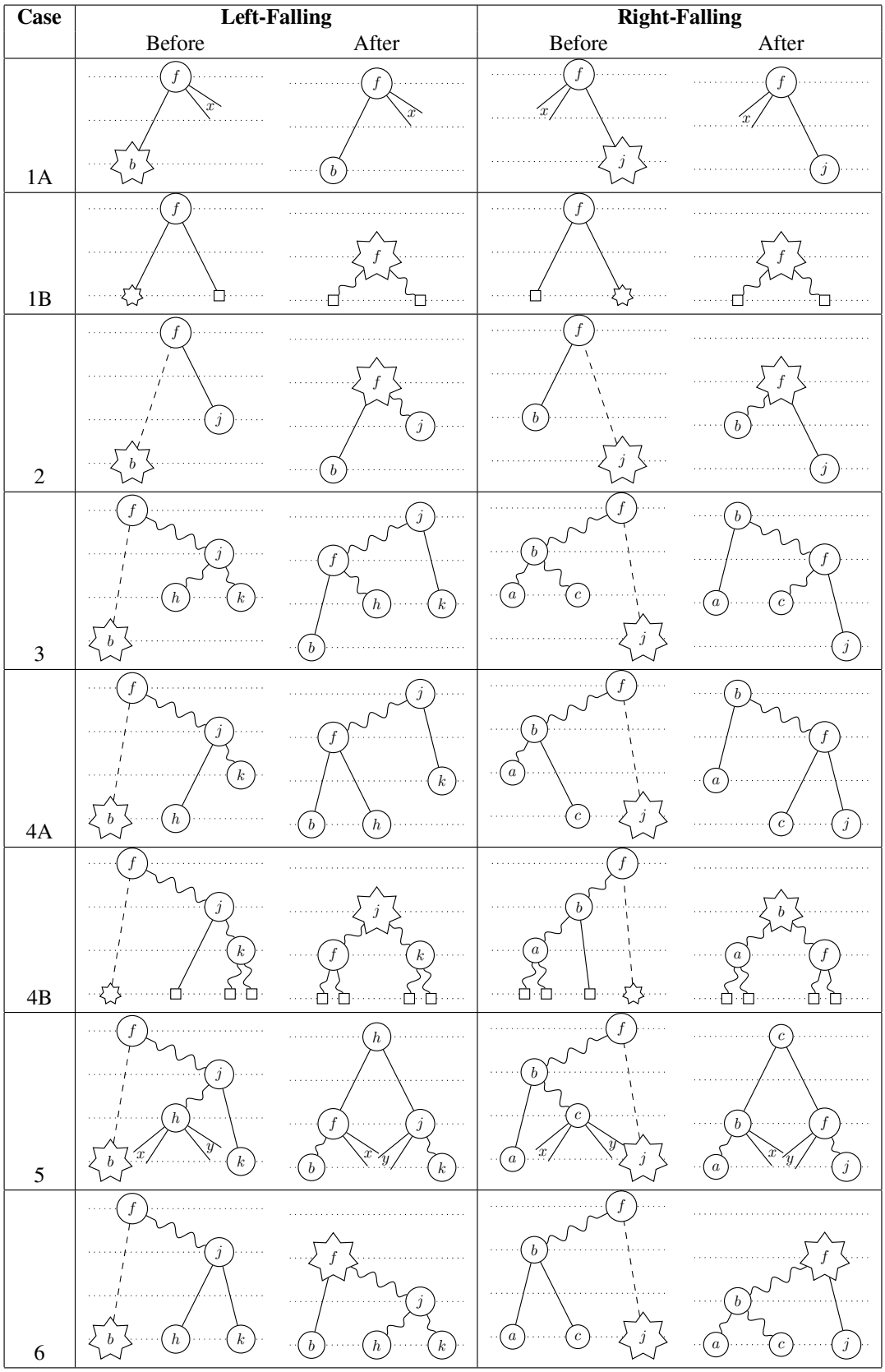

Fig. 5. Rebalancing after deletion.

and unit tests. Figures 4 and 5 aid in debugging; each of our bugs turned out to be a discrepancy between the figure and the corresponding code. Experimental results, shown in Figure 6, confirm that insertion and deletion operations do indeed take logarithmic time.

While an implementation could store each node's level explicitly in an integer, it is possible to represent balance information more compactly. The shape of each internal node of a crumple tree may be in one of four states: $(1,1), (1,2), (2,1)$ or $(2,2)$. Distinguishing between these states requires two bits of storage overhead per internal node. This is not too much more than red-black trees which store one bit per node for the red/black state. Nor is it much more than AVL tree nodes which have three possible states, so in theory need 1.5 bits per node, which in practice usually rounds up to two bits per node. It is actually more natural to make crumple tree balance information a property of edges instead of nodes; each edge in a balanced tree has length one or two. An implementation that prioritizes space efficiency could pack one bit of edge-length information into the least significant bit of each child pointer. This tactic is common to Lisp implementations [21] [22] since allocators conventionally assign only even-numbered memory addresses, leaving the least-significant bit of object addresses available for other uses. A crumple tree implementation that uses this tactic effectively uses zero space overhead for balance information; each internal node only needs to store one data element and two bit-packed child pointers.

VII. Conclusion and Future Work

Crumple trees meet CS2 students where they are. Their structure, rebalancing algorithms, analysis, and implementation are all within reach. Consequently crumple trees support various intructional modalities. They could be presented in full in a conventional, accessible lecture; derived through an instructor-facilitated interactive discussion; with some guidance, discovered by students themselves; and implemented as a hands-on project. Crumple trees avoid logistical impediments that are prosaic, yet still legitimate obstacles. They are convenient to draw on slides, whiteboard, and paper, and do not depend upon color. They are competitive with other BSTs asymptotically, and crumple trees' constant factors are reasonable.

Avenues for future work include empirical analysis of an optimized industrial-grade implementation, and larger-scale study of student learning outcomes using crumple trees versus alternatives. This method of using codified sketches to facilitate discovery learning could be applied to other linked data structures such as linked lists and heaps.

References

[1] G. Adel'son-Vel'skii and E. M. Landis, "An algorithm for the organization of information," *Sov. Math. Dokl.*, vol. 3, pp. 1259—1262, 1962.

[2] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, Oct 1978, pp. 8–21.

[3] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1974.

[4] A. Andersson, "Balanced search trees made simple," *Algorithms and Data Structures*, pp. 60–71, 1993.

[5] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, Dec 1972. [Online]. Available: http://dx.doi.org/10.1007/BF00289509

[6] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990. [Online]. Available: http://doi.acm.org/10.1145/78973.78977

[7] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3828.3835

[8] C. R. Aragon and R. G. Seidel, "Randomized search trees," in *Foundations of Computer Science, 1989., 30th Annual Symposium on*. IEEE, 1989, pp. 540–545.

[9] S. Sen and R. E. Tarjan, "Deletion without rebalancing in balanced binary trees," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010, pp. 1490–1499.

[10] C. M. Lewis, K. Yasuhara, and R. E. Anderson, "Deciding to major in computer science: a grounded theory of students' self-assessment of ability," in *Proceedings of the seventh international workshop on Computing education research*. ACM, 2011, pp. 3–10.

[11] L. Murphy and L. Thomas, "Dangers of a fixed mindset: implications of self-theories research for computer science education," in *ACM SIGCSE Bulletin*, vol. 40, no. 3. ACM, 2008, pp. 271–275.

[12] B. Simon, B. Hanks, L. Murphy, S. Fitzgerald, R. McCauley, L. Thomas, and C. Zander, "Saying isn't necessarily believing: Influencing self-theories in computing," in *Proceedings of the Fourth International Workshop on Computing Education Research*, ser. ICER '08. New York, NY, USA: ACM, 2008, pp. 173–184. [Online]. Available: http://doi.acm.org/10.1145/1404520.1404537

[13] A. E. Bell, S. J. Spencer, E. Iserman, and C. E. Logel, "Stereotype threat and women's performance in engineering," *Journal of Engineering Education*, vol. 92, no. 4, pp. 307–312, 2003.

[14] S. Cheryan, V. C. Plaut, P. G. Davies, and C. M. Steele, "Ambient belonging: how stereotypical cues impact gender participation in computer science." *Journal of personality and social psychology*, vol. 97, no. 6, p. 1045, 2009.

[15] J. Cooper and K. D. Weaver, *Gender and computers: Understanding the digital divide*. Psychology Press, 2003.

[16] J. Margolis and A. Fisher, *Unlocking the clubhouse: Women in computing*. MIT press, 2003.

[17] B. Haeupler, S. Sen, and R. E. Tarjan, "Rank-balanced trees," *ACM Trans. Algorithms*, vol. 11, no. 4, pp. 30:1–30:26, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2689412

[18] S. Freeman, S. L. Eddy, M. McDonough, M. K. Smith, N. Okoroafor, H. Jordt, and M. P. Wenderoth, "Active learning increases student performance in science, engineering, and mathematics," *Proceedings of the National Academy of Sciences*, vol. 111, no. 23, pp. 8410–8415, 2014.

[19] J. J. McConnell, "Active learning and its use in computer science," *ACM SIGCSE Bulletin*, vol. 28, no. SI, pp. 52–54, 1996.

[20] M. Ben-Ari, "Constructivism in computer science education," *Journal of Computers in Mathematics and Science Teaching*, vol. 20, no. 1, pp. 45–73, 2001.

[21] F. Winkelmann et al. (2019) CHICKEN scheme data representation. [Online]. Available: https://wiki.call-cc.org/man/4/Data%20representation

[22] Free Software Foundation. (2019) The Guile Reference Manual. [Online]. Available: https://www.gnu.org/software/guile/manual/html_node/Data-Representation.html

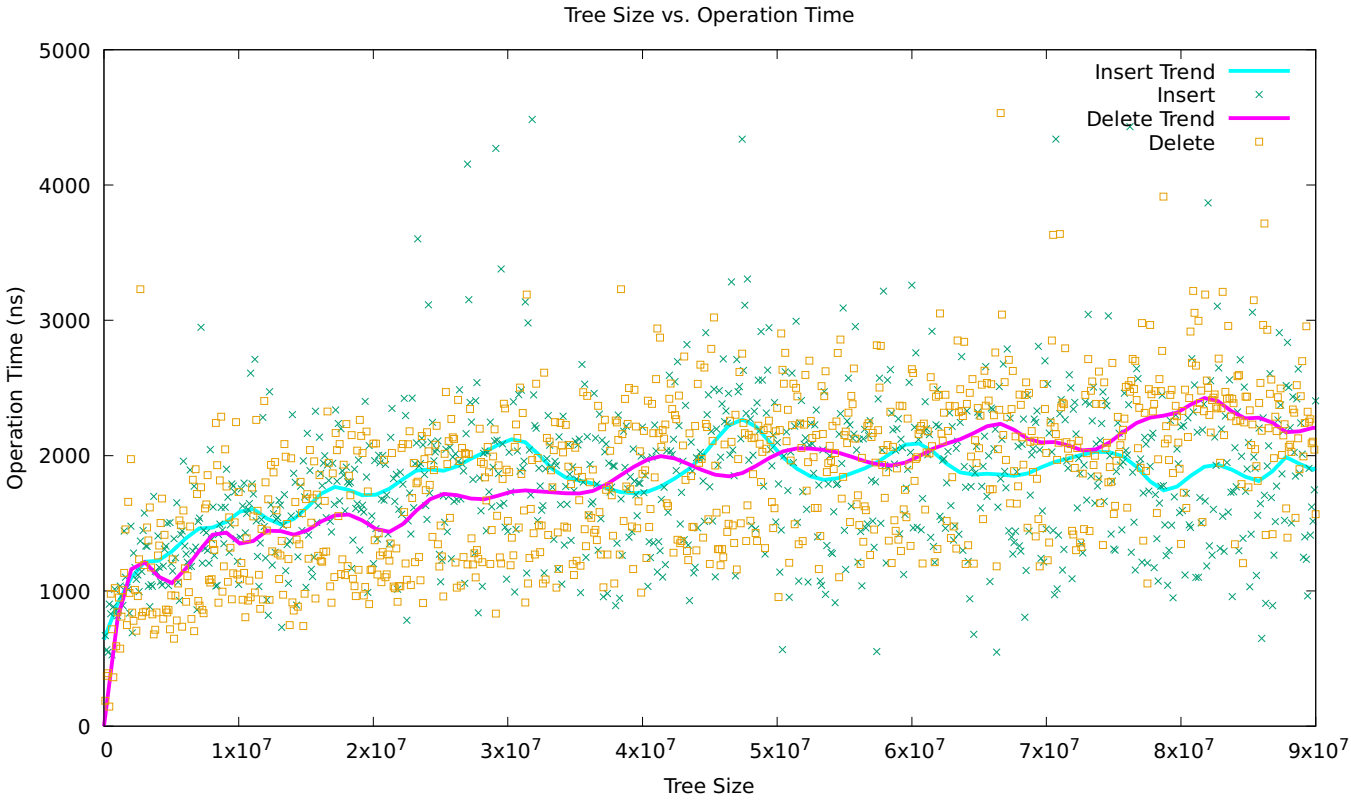

Fig. 6. Empirical performance of our C++ implementation.