Teaching the Analysis of Algorithms with Visual Proofs

Michael T. Goodrich*
Dept. of Comp. Sci.
Johns Hopkins Univ.
Baltimore, MD 21218
goodrich@cs.jhu.edu

Roberto Tamassia†
Dept. of Comp. Sci.
Brown Univ.
Providence, RI 02912
rt@cs.brown.edu

Abstract

We describe an approach for visually teaching important proofs in the Junior-Senior level course on the design and analysis of data structures and algorithms (CS7/DS&A). The main idea of this educational paradigm is to justify important claims about data structures and algorithms by using pictures that visualize proofs so clearly that the pictures can qualify as proofs themselves. The advantage of using this approach for DS&A is that it augments or even replaces inductive arguments that many students find difficult. Moreover, this paradigm communicates important algorithmic facts in a compelling way for students who are more visually-oriented. We illustrate this technique by giving examples of visual proofs of several key concepts in DS&A.

1 Introduction

In this era of real-time video games and *MTV*, students these days seem more visually-oriented than ever. They learn most naturally by seeing a concept described with a picture, and they remember that concept by recalling the picture that goes with it. This visual orientation is actually quite natural, for we humans devote an immense amount of brain power to the processing of visual information. We feel that we can realize great educational benefits by finding visual ways of presenting the key ideas of important computer science concepts.

In this paper we address the communication of key concepts in the design and analysis of data structures and algorithms, which are topics taught in a course known by the acronyms CS7 and DS&A (we will use DS&A). This course is full of powerful ideas that have many applications, yet key concepts in DS&A are not fully comprehended by many students. We feel that this lack of comprehension is due to the fact that these concepts are often presented and justified by invoking sophisticated mathematical arguments. We argue in this paper that this mathematical sophistication is often unnecessary, because key ideas of DS&A can be presented visually.

As a justification of the potential of the visual alternative to teaching DS&A, we describe simple visual proofs of several core topics in DS&A, including the following:

- summing linear terms,
- counting nodes in a binary tree,
- analyzing binary tree traversal,
- analyzing bottom-up heap construction,
- rebalancing AVL trees via rotations.

Some of the visual proofs we present are new, to the best of our knowledge, while others are known but possibly under-utilized.

1.1 Related Work

The trend towards visual ways to presenting important topics of DS&A finds its inspiration in the work of Brown and Sedgwick on algorithm animation and visualization [4, 5, 11, 12], as well as that of Stasko [13] and others [2, 3]. This work illustrates the power of visualization for communicating how algorithms work and how they transform their inputs. In addition, the authors include several additional visual ways of presenting ideas in DS&A in their recent book [7].

Many of the visual proofs we present in this paper augment proofs that use mathematical induction (e.g., see Manber [10]). We feel that induction is a beautiful and powerful mathematical tool, but it nevertheless is something that many students find mysterious. One of the motivations for our use of visual proofs is to reduce our reliance on mathematical induction as the only way of justifying important concepts in DS&A, and thereby effectively educate students that seem to never comprehend this proof technique.

We describe several visual proofs in the remainder of this paper, beginning with a well-known summation identity that is usually justified using mathematical induction.

*The work of this author is supported by the U.S. Army Research Office under grant DAAH04–96–1–0013, and by the National Science Foundation under grant CCR–9625289.

†The work of this author is supported by the U.S. Army Research Office under grant DAAH04–96–1–0013, and by the National Science Foundation under grant CCR–9423847.

2 Combinatorial Arguments

One of the first analyses that students see in DS&A is an analysis of the worst-case running time an algorithm such as bubble-sort, insertion-sort, selection-sort, or quick-sort. Each of these analyses use the following summation:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$$

This summation arises in the analyses because of an iteration performed by each of the algorithms where the number of operations performed inside the loop increases by a fixed, constant amount with each iteration. This summation has the following identity:

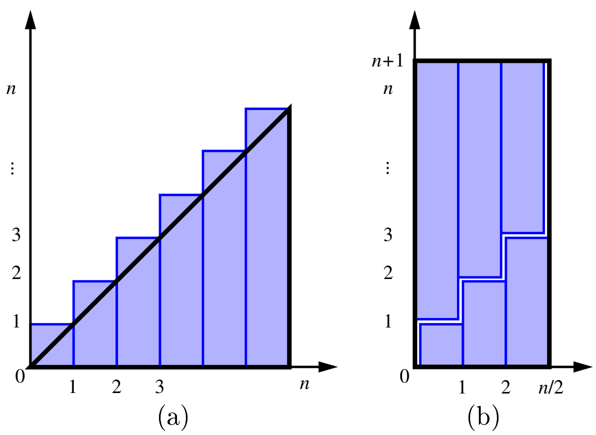

Figure 1: Visual justifications of Proposition 2.1. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1, 2, \ldots, n$. In (a) the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles of area $1/2$ each (base 1 and height 1). In (b), which applies only when n is even, the rectangles are shown to cover a big rectangle of base $n/2$ and height $n+1$.

Proposition 2.1: *For any integer $n \geq 1$, we have*

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

We give two visual proofs of this fact in Figure 1. The illustration in Figure 1.a is less well known as the one in Figure 1.b, but it applies for all values of n whereas the illustration in Figure 1.b only applies when n is even (although it is fun exercise to ask students to provide an analogous visual proof for the case when n is odd). Both of these visual proofs augment a well-known proof by induction (e.g., see [7]).

3 Binary Tree Algorithms

Almost immediately after giving the above summation identity, the curriculum for DS&A turns to discussions of several topics involving binary trees, including their combinatorial properties and their uses as search structures. We discuss some visual justifications for several facts involving binary trees in this section.

3.1 Counting Nodes in a Binary Tree

Binary trees have several interesting structural properties, which are not shared by general trees. A simple, but important such property is the following:

Proposition 3.1: *In a proper binary tree, where each internal node has two children, the number of external nodes is 1 more than the number of internal nodes.*

Proof: We justify this fact using a simple visual proof, which is actually a proof-by-induction in "disguise." Let T be a proper binary tree. If T has only one node, then this node is external, and the property holds. Otherwise, remove from T an (arbitrary) external node w and its parent v, which is an internal node. If v has a parent u, reconnect u with the former sibling z of w, as shown in Figure 2. This operation removes one internal node and

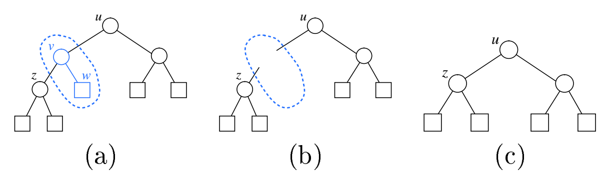

Figure 2: The operation that removes an external node and an internal node in the justification of Proposition 3.1.

one external node, and it leaves the tree being a proper binary tree. By repeating this operation, we shall eventually obtain a binary tree with a single external node. Since the same number of external and internal nodes are removed by this sequence of operations, and we end up with a single external node, we conclude that the number of external nodes of T is 1 plus the number of internal nodes. ■

We next discuss a common algorithm that is performed on binary trees.

3.2 Analyzing Binary Tree Traversal

One of the prime uses of binary trees is to store objects, and these objects are often enumerated by using binary tree traversal algorithms, such as the preorder, inorder, and postorder traversal algorithms. Viewed in an object-oriented framework, these tree-traversal algorithms are all forms of iterators (or enumerations in Java). Each traversal visits the nodes of a tree in a certain order, guaranteed to visit each node exactly once. However, we can unify these tree-traversal algorithms into a single framework, by relaxing the requirement that each node is visited exactly once. The resulting traversal is called the *Euler tour traversal* [8,9]. The advantage of the Euler tour traversal is that it allows for more general kinds of tree traversals to be easily expressed.

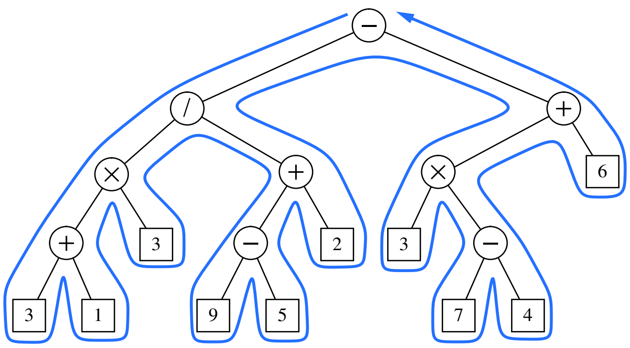

Figure 3: Euler tour of a binary tree.

Intuitively, the Euler tour traversal of a binary tree T can be informally defined as a walk around T, where we start by going from the root towards its left child, viewing the edges of T as being "walls" that we always keep to our left. (See Figure 3.) Each node v of T is encountered three times by the Euler tour:

- "On the left" (before the Euler tour of v's left subtree)
- "From below" (between the Euler tours of v's two subtrees)
- "On the right" (after the Euler tour of v's right subtree).

If v is external, then these three "visits" actually happen at the same time.

The preorder, inorder, and postorder traversals of T are equivalent to an Euler tour, such that each node is visited when encountered on the left, from below, or on the right, respectively. The time complexity of the preorder, postorder, and inorder tour traversals of a binary tree with n nodes are easy to analyze using the Euler tour traversal and its visualization in Figure 3. Suppose that visiting a node takes $O(1)$ time, which is often the case. In this case, we spend a constant amount of time at each node of the tree during the traversal, so that the overall time complexity is $O(n)$.

3.3 Analyzing Bottom-Up Heap Construction

Binary trees are discussed in DS&A again in the heap-sort algorithm, where they are used to implement the priority queue abstract data type in the heap data structure. One way to present the heap-sort algorithm is to show that we can construct a heap storing n keys (or key-element pairs) in $O(n \log n)$ time by means of n successive insertion operations, each taking $O(\log n)$ time, starting from an empty heap [14]. However, if all the keys to be stored in the heap are known in advance, there is an alternative *bottom-up* construction method, which runs in $O(n)$ time [6]. This is a more-efficient construction algorithm that can be included as one of the constructors in a Heap class. Bottom-up heap construction is shown in Code Fragment 1.

Algorithm $\mathsf{BottomUpHeap}(S)$:

Input: a sequence S storing $n = 2^h - 1$ keys

Output: a heap T storing the keys in S.

> **if** S is empty **then**
>   **return** an empty heap (consisting of a single external node).
> **end if**
> Remove the first key, k, from S.
> Split S into two sequences, S_1 and S_2, each of size $(n-1)/2$.
> Let $T_1 = \mathsf{BottomUpHeap}(S_1)$.
> Let $T_2 = \mathsf{BottomUpHeap}(S_2)$.
> Create a binary tree T with root node r storing k, left subtree T_1, and right subtree T_2.
> Perform a down-heap bubbling from the root r of T, if necessary.
> **return** T.

Code Fragment 1: Recursive bottom-up heap construction.

Bottom-up heap construction is asymptotically faster than incrementally inserting n keys into an initially-empty heap, as the following proposition shows.

Proposition 3.2: *The bottom-up construction of a heap with n keys takes $O(n)$ time.*

Proof: Let us use a function $t(n)$ to denote the running time of this algorithm, where n is the number of keys. We claim that $t(n)$ is $O(n)$. Since the algorithm is recursive, one approach to justifying this claim is to characterize the function $t(n)$ by the recurrence relation

$$t(n) \le \begin{cases} b & \text{if } n \le 1 \\ 2t(n/2) + c \log n & \text{otherwise,} \end{cases}$$

where b and c are constants. Analyzing this formula usually involves reducing it to a closed form showing that $t(n)$ is

$$O\left(n + \sum_{i=2}^{\log n} \left(\frac{i}{2^i}(n+1)\right)\right),$$

which is then solved by using induction and some facts from Calculus. We offer instead the visual proof illustrated in Figure 4.

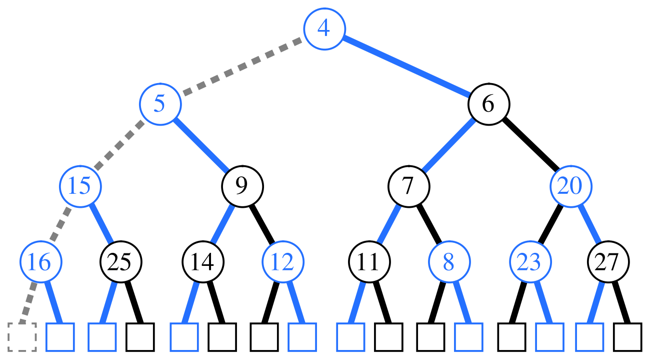

Figure 4: Visual justification of the linear running time of bottom-up heap construction, where the paths associated with the internal nodes have been highlighted with alternating colors. For example, the path associated with the root consists of the internal nodes storing keys 4, 6, 7, and 11, plus an external node. Also, the path associated with the right child of the root consists of the internal nodes storing keys 6, 20, and 23, plus an external node.

Let T be the final heap, and let v be an internal node of T, and let $T(v)$ denote the subtree of T rooted at v. In the worst-case, the time for forming $T(v)$ from the two recursively-formed subtrees rooted at its children is proportional to the height of $T(v)$. The worst-case occurs when down-heap bubbling from v traverses a path from v all the way to a bottommost external node of $T(v)$.

Consider now the path $p(v)$ of T from node v to its inorder successor external node, i.e., the path that starts at v, goes to the right child of v, and then goes down leftward until it reaches an external node. We say that path $p(v)$ is *associated with* node v. Note that $p(v)$ is not necessarily the path followed by down-heap bubbling when forming $T(v)$. Clearly, the length (number of edges) of $p(v)$ is equal to the height of $T(v)$. Hence, forming $T(v)$ takes in the worst case time proportional to the length of $p(v)$. Thus, the total running time of bottom-up heap construction is proportional to the sum of the lengths of the paths associated with the internal nodes of T.

It is easy to see that for any two internal nodes u and v of T, paths $p(u)$ and $p(v)$ do not share edges, although they may share nodes (see Fig. 4). Hence, the sum of the lengths of the paths associated with the internal nodes of T is no more than the number of edges of heap T, i.e., no more than $2n$. We conclude that the bottom-up construction of heap T takes $O(n)$ time. ■

Algorithm rotate(x):

Input: a node x of a binary search tree T that has both a parent y and a grandparent z

Output: tree T restructured

1: Let (a, b, c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (T_0, T_1, T_2, T_3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.
2: Replace the subtree rooted at z with a new subtree rooted at b
3: Let a be the left child of b and give a the roots of T_0 and T_1 as its left and right children, respectively.
4: Let c be the right child of b and give c the roots of T_2 and T_3 as its left and right children, respectively.

Code Fragment 1: Rotation in a binary search tree.

3.4 The Analysis of AVL Trees

One the prime uses of binary trees is to support the binary search tree data structure, and one of the most popular binary search trees is the AVL tree [1,7]. A difficult case analysis is often included in DS&A for rebalancing AVL trees after insertions and deletions. We offer a unifying visual approach.

Let w be a node in an AVL tree that has just been updated because of an insertion (the deletion method is similar). Let x be the first node we encounter in going up from w toward the root of T such that the grandparent z of x is unbalanced. Note that node x could be equal to w. Also, let y denote the parent of x, so that y is a child of z. Since node z became unbalanced because of an insertion in the subtree rooted at its child y, the height of y is equal to 2 plus the height of the sibling of y. We now rebalance the subtree currently rooted at z by performing a *rotation* operation, which is described in Code Fragment 1 and is schematically illustrated in Figure 5. This operation temporarily renames the nodes x, y, and z as a, b, and c, so that a is left of b and b is left of c (in an inorder traversal listing). It then replaces z with the node called b, makes the children of this node be a and c, and makes the children of a and c be the four previous children of x, y, and z (other than x and y), while maintaining the inorder relationships of all the nodes in T.

This rebalancing operation is called a *rotation* because of a geometric way we can visualize the way it restructures T. If $b = y$ (see again Code Fragment 1), the execution of method rotate is called a *single rotation*, for it can be visualized as "rotating" y over z (see Figure 5(a)–(b)). Otherwise, if $b = x$, this operation is called a *double rotation*, for it can be visualized as first "rotating" x over y and then over z (see Figure 5(c)–(d)). Some researchers

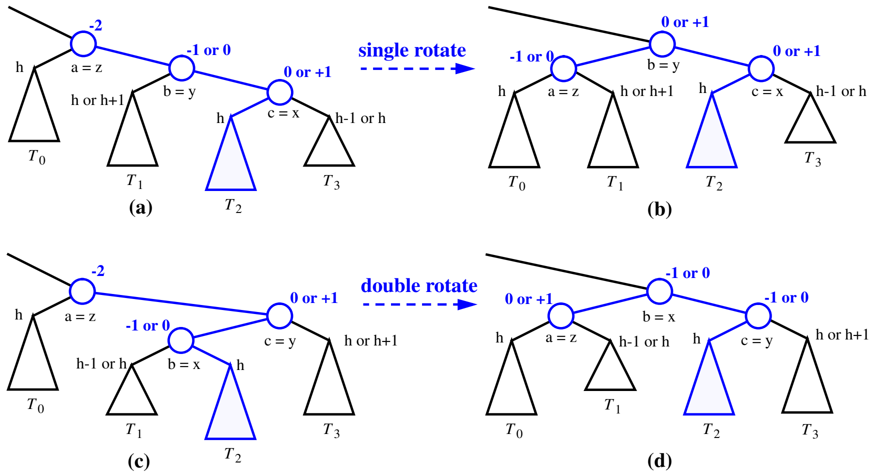

Figure 5: Schematic illustration of method rotate described in Code Fragment 1. We show next to nodes a, b and c the signed difference between the heights of the right and left subtree. Also, we show next to subtrees $T_0, \ldots, T_3$ their height: (a)–(b) single rotation; (c)–(d) double rotation.

separate these two kinds of rotations as separate methods; we have chosen however a rotate method that unifies these two types of rotations.

The prime reason for a rotation is to change the heights of nodes in T so as to restore balance. Recall that we execute a rotation operation because z, the grandparent of x, is unbalanced. Moreover, this unbalance is due to one of the children of x now having to large a height relative to the height of z's other child. As a result of a rotation we move up the "tall" child of x while pushing down the "short" child of z. Thus, after performing a rotation, all the nodes in the subtree now rooted at the node we called b are balanced (see Figure 5). (A similar visual analysis works for deletions.)

4 Conclusion

In this paper we present visual proofs for several key concepts taught in the design and analysis of data structures and algorithms course (CS7/DS&A), and we argue that these proofs are effective ways of teaching powerful ideas of DS&A without resorting to sophisticated mathematics. We have not tried to present an exhaustive repetoire of visual proofs, however, and we encourage the reader to develop visual proofs of his or her own. In addition, we refer the reader interested in further examples of visual ways of presenting important concepts for DS&A (and also the Freshman-Sophomore data structures course (CS2)) to the recent book by the authors [7].

References

[1] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, **3**, 1259–1262.

[2] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. A model for algorithm animation over the WWW. *ACM Comput. Surv.*, 27(4):568–572, 1995.

[3] J. L. Bentley and B. W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1):5–30, Winter 1991.

[4] M. H. Brown. *Algorithm Animation*. MIT Press, Cambridge, 1988.

[5] M. H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, Jan. 1985.

[6] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[7] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley and Sons, New York, 1997.

[8] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Mass., 1992.

[9] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–941. Elsevier/The MIT Press, Amsterdam, 1990.

[10] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, Reading, Mass., 1989.

[11] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 1983.

[12] R. Sedgewick. *Algorithms in C++*. Addison Wesley, Reading, MA, 1992.

[13] J. T. Stasko. Simplifying algorithm animation with tango. In *Proc. IEEE Workshop on Visual Languages*, pages 1–6, 1990.

[14] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.