Teaching Binary Tree Algorithms through Visual Programming

Amir Michail
Department of Computer Science and Engineering
University of Washington, Box 352350
Seattle, Washington 98195, United States of America
amir@cs.washington.edu

Abstract

In this paper, we show how visual programming can be used to teach binary tree algorithms. In our approach, the student implements a binary tree algorithm by manipulating abstract tree fragments (not necessarily just single nodes) in a visual way. This work contributes to visual programming research by combining elements of animation, programming, and proof to produce an educational visual programming tool. In addition, we describe our experiences with Opsis, a system we built to demonstrate the ideas in this paper. (Opsis is a Java applet and can be accessed at http://www.cs.washington.edu/homes/amir/Opsis.html.) Finally, we make the claim that visual programming is an ideal way to teach data structure algorithms.

1 Introduction

The idea of using computers to teach algorithms is not new; computer animations are used to illustrate many algorithms. Although one might gain insight from watching an algorithm animation, it is often the case, as with any passive activity, that important details are glossed over or missed altogether. Indeed, the student may not really understand the algorithm but merely have a pretense of having done so. Empirical evidence suggests that students learn better through *active* rather than passive activities.[9]

A more active method is to have the student implement the algorithm in some textual programming language. In this way, the student must understand the details of the algorithm in order to implement it correctly. However, although the student may now understand *how* the algorithm works, this does not mean the student understands *why* it works. Moreover, the student often has to go through the drudgery of low-level details, such as pointer manipulation, that are not crucial to understanding the algorithm. Worse yet, the algorithm may naturally require the formulation of mental pictures not readily expressible in code — the student must constantly translate back and forth between the mental pictures and textual code.

Another possibility is to have the student present a proof of correctness for the algorithm. This is often done by coming up with various loop invariants and then proving them correct by induction. Now the student has to really think about why the algorithm works. The low-level details of the implementation need not be considered anymore. One problem with this approach is that the program may be quite large and complicated, thus making a rigorous correctness proof laborious and error-prone. Another problem is that the student may not be able to easily experiment with concepts as is possible with programming on a computer. Furthermore, the student does not gain the satisfaction of seeing the algorithm execute after having implemented it.

In this paper, we show that through visual programming, one can combine various elements of animation, programming, and proof so as to teach binary tree algorithms in a more effective manner. In our approach, the student implements a binary tree algorithm by manipulating abstract tree fragments (not necessarily just single nodes) in a visual way. In so doing, the student not only programs the algorithm but also proves some of its properties and can animate it on examples if desired.

The remainder of the paper is organized as follows: Section 2 surveys previous work done on related subjects; Section 3 presents the visual programming model; Sections 4, 5, and 6 describe visual binary tree depiction, navigation, and manipulation, respectively; Section 7 discusses the correctness of the visual binary tree programs; Section 8 describes our current implementation; Section 9 presents preliminary user testing with our system; and Section 10 provides a summary and future research directions.

2 Past Work

Many of the elements of our approach have been considered in earlier research, but not all at the same time, and not within the context of an educational visual programming

tool. Moreover, we believe our approach is enhanced from the synergy of many disparate ideas encountered throughout the literature.

Much work has been done in algorithm animation. As an example, the Zeus [1] system animates many fundamental algorithms and does so in several ways. However, as we are interested in abstract representations of trees (i.e, depicting a class of trees, not just one), we draw upon more abstract tree diagrams found in many algorithms and data structures texts (such as [10, 4]).

Many visual programming systems have been designed for beginning programmers or application end-users. In the former group, we find systems like Glinert's PICT [7], in which a programmer uses icons and flowcharts to program simple arithmetic computations. In the latter group, we find systems like Modugno's Pursuit [11], which allows end-users to visually program simple shell scripts by example. However, in both groups, the systems are not designed to teach algorithms, nor do they allow easy construction of complicated algorithms (such as AVL tree insertions or deletions).

Some visual programming systems have been designed with more advanced programmers in mind. For example, Christensen's AMBIT/G [2] and AMBIT/L [3] languages have been used to manipulate directed graphs and lists, respectively. However, these languages are essentially a visual version of Snobol with static pictures to indicate the pattern matching rules; the resultant programs can be difficult to read and manipulate. Our approach is more dynamic, similar in style to the data structure programming system Think Pad [13], but easier to use and more abstract.

As we are also interested in visually proving various properties of the binary tree algorithms, we borrow some concepts (primarily loop invariants) from the area of programming methodology (i.e., formal methods). Indeed, this work is inspired by research into how one can implement an algorithm and prove it correct at the same time.[8] However, such efforts make use of complicated first-order logic expressions and are not necessarily suited — as is — for the task of teaching algorithms.

3 Visual Programming Model

We use a state-based model for representing a program. A user specifies transitions from one state to another by manipulating abstract objects in a visual manner. The idea is to specify the algorithm in full generality — and not just on a specific example. For this reason, our approach is not *programming by example* [14, 12, 6] but is similar to *programming by abstract demonstration* [5].

3.1 State Types

For our computation model, we use an *abstract state*, which is an abstract visual diagram that represents a set of concrete states (e.g., a set of binary trees). Two abstract states are *identical* if and only if their respective abstract visual diagrams match exactly. It is possible for two abstract states to have different diagrams but still represent the same set of concrete states.

For example, an abstract state may represent all binary search trees with a node containing a certain key K. Such an abstract state is shown visually in Figure 1, (4c). Another abstract state, different from Figure 1, (4c), but that represents the same binary search trees is shown in Figure 2, (4).)

In the remainder of the paper, we shall use the word "state" whenever we mean "abstract state".

3.2 State Graph

Specifically, we model a user-defined function as a *state graph*, which is a directed graph whose nodes represent states and whose directed edges represent *operations* to transform one state to some other(s). Computation starts at the *initial state* and ends at one of the *final states*. The initial state specifies the parameters passed in to the function, while each final state specifies a possible return value for that function. In the state graph, the initial state has no incoming edges and the final states have no outgoing edges. (In Figure 1, the initial state is (1) and the final states are (3b) and (4c).)

(In Figures 1, 2, and 3, the operation shown (textually) below each tree diagram is invoked on the selected fragments in that diagram. Selected fragments are indicated by dashed lines.)

3.3 Sequencing, Iteration, and Conditionals

The state-based model subsumes sequencing, iteration, and conditionals — no additional control flow constructs are required. Sequencing is accomplished though simple transitions from one state to another. Iteration is done by creating cycles in the state graph. Conditionals are done through operations which result in two or more states. (In Figure 1, sequencing occurs from (4b) to (2), iteration occurs in (2), (3a), (4a), (2), and a conditional occurs from (2) to (3a) and (3b). Observe that the loop ends when the subtree being expanded in (2) is empty.)

4 Visual Binary Tree Abstractions

In this paper, we describe a visual formalism for implementing dictionary "search", "insert", and "delete" opera-

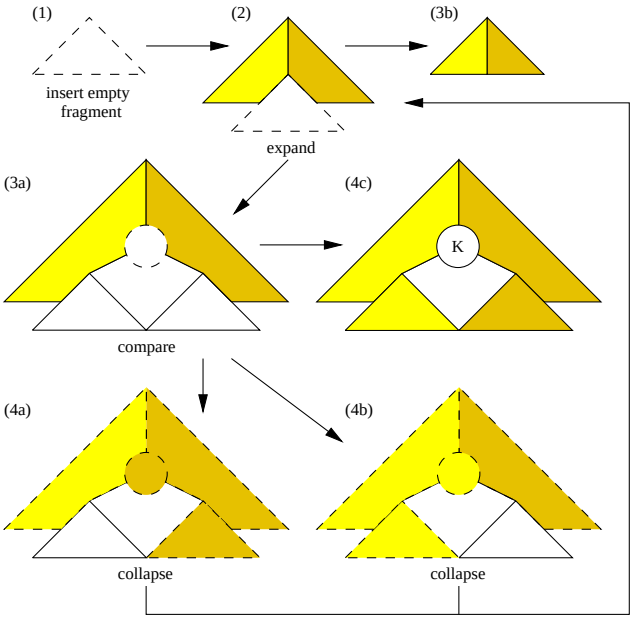

Figure 1. Visual code for binary tree search.

tions via binary search tree algorithms. This will be done by manipulating nodes and fragments in a visual way.

4.1 Fragments and Subtrees

Before we proceed, we need to introduce the notion of a *fragment*: a fragment is a (possibly empty) connected subgraph of a binary tree. A fragment is similar to a subtree in that it consists of a root node and descendents of that node. Unlike a subtree, a fragment need not include all descendents of its root.

4.2 Visual Depiction of Nodes and Fragments

Visually, we use a circle to represent a node and a triangle to depict a (possibly empty) tree fragment. Nodes and fragments may also be shaded light gray or dark gray. Those shaded light gray have keys less than a certain key K; those shaded dark have keys greater than K. (See Figure 1, (4a).)

A fragment may also be shaded light on the left and dark gray on the right, thus indicating that it may have some nodes with keys less than K and some nodes with keys greater than K *but* that it doesn't have a node with key K. (For example, see Figure 1, (3b).) The vertical boundary between the light and dark halves of a fragment indicates a path of nodes, $(x_1, x_2, \ldots, x_j)$, such that:

1. the key at each x_i is not K; and
2. x_i is the left (resp. right) child of its parent x_{i-1} if and only if K is less (greater) than the key at x_{i-1}.

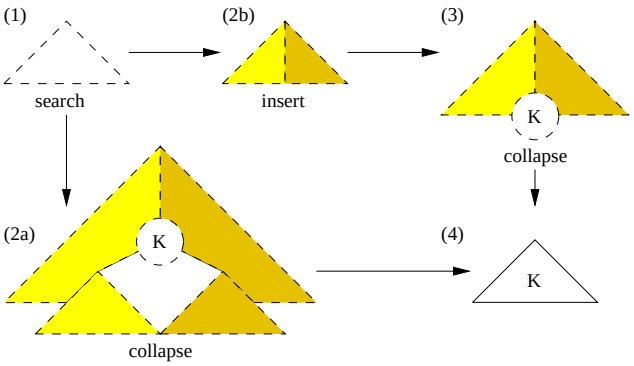

Figure 2. Visual code for binary tree insertion.

(In Figure 5, we show: (1) an abstract tree diagram with the upper fragment consisting of light and dark halves; (2) a sample binary tree from the class of trees represented by the abstract tree diagram. In this case, the nodes represented by the vertical boundary (separating the light and dark halves) have keys: 'X', 'K', and 'S'.)

A node labeled K indicates that the node has the desired key K, while a fragment labeled K indicates that the (non-empty) fragment has a node in it which has key K. (In Figure 1, (4c), a node with key K is explicitly shown in the tree. In Figure 2, (4), we just know that some node in the tree has key K.)

5 Visual Binary Tree Navigation

The user can navigate around a binary tree by changing the view to that tree. We believe this approach is flexible and more natural than using pointer variables (as is done in textual programming languages). There are three main operations for changing views: expand, collapse, and insert empty fragment.

5.1 Expand

The *expand* operation allows one to see an additional node in a binary tree fragment if one exists, or to otherwise determine that the fragment is empty. (In Figure 1, (2), we use the expand operation to show the root node of the selected subtree if the subtree is non-empty (as in state (3a)), or to indicate that the subtree is empty (as in state (3b)).)

If the fragment being expanded is non-empty, then the particular node that will be shown depends on whether the fragment is a subtree or not. If the fragment is a subtree, then the fragment expands into a node (i.e., the root) connected to its left and right subtrees. (This is what happened in Figure 1, (3a).)

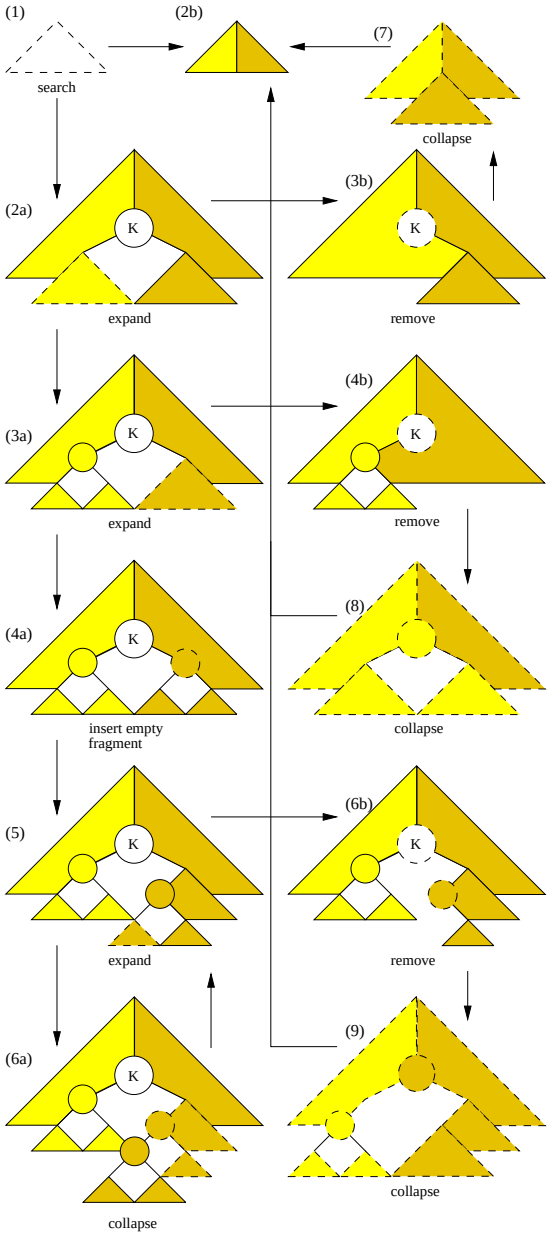

Figure 3. Visual code for binary tree deletion.

If the fragment is not a subtree (because some other node/fragment appears below it), then the fragment expands by showing the parent of the node/fragment just below it. As that parent may have the node/subtree as its left or right child, we have two cases. (In Figure 1, (4c), the top fragment would expand by revealing the parent of the node labeled K.) This latter type of expand is useful for moving up in a tree, as one might do to maintain a balanced binary tree structure.

5.2 Collapse

The *collapse* operation allows one to combine several fragments into one fragment. The fragments to be combined must be adjacent to each other in the tree.

(For example, in Figure 1, (4a), we use collapse to combine three tree fragments into one, thus yielding state (2).)

Of course, it is possible for the fragments to be collapsed to have different properties. Whether the resultant fragment inherits a property from one of its constituent fragments depends on the type of the property and on whether the other fragments have that property:[1]

1. If the property is *disjunctive*, then if any of the fragments have it, then the resulting fragment will have it. (For example, in Figure 2, (2a), the resultant fragment, (4), inherits the property that "it contains a node with key K" because that property is disjunctive.)

2. If the property is *conjunctive*, then all fragments must have it for the resulting fragment to have it. (For example, in Figure 2, (2a), any property signifying nodes with keys less than or greater than K are not preserved in (4) because such properties are conjunctive. However, in Figure 3, (6a), the fragments to be collapsed all have the property that "all nodes have keys greater than K", so the resultant fragment, in (5), retains this property.)

5.3 Insert Empty Fragment

The *insert empty fragment* operation allows one to insert an (initially) empty fragment anywhere in the abstract binary tree. In doing so, we obtain a new state that represents a larger class of binary trees (but that includes all those represented earlier). Inserting empty fragments in this way allows us to create loop invariants, which we use to form loops.

Whenever we insert a fragment, it is initially empty. This means we can make the new fragment have any property we desire — as long as that property doesn't imply the existence of at least one node. (For example, in Figure 1, (1), (2), we

[1] Similar rules can be formulated for "expand", but we shall not do so here.

add an empty fragment with the property that "no node have key K".)

5.4 An Example

We have now described various ways to navigate around a binary tree. These concepts are sufficient to understand the binary tree search algorithm shown in Figure 1.

Initially, we prepare to search for the key K in the tree by adding an initially empty fragment above the currently selected subtree (in state (1)). This new fragment does not have any nodes with key K (so half of the fragment is colored light gray while the other half is colored dark gray). Upon adding the empty fragment, we obtain the the loop invariant for the search (shown in state (2)).

At this point, we wish to determine whether there is a node with key K in the selected subtree (in state (2)). So, we expand the subtree, and depending on whether the subtree is empty or not, we obtain state (3b) or state (3a), respectively. If the subtree is empty, then we have shown that no node in the binary tree has key K.

If the subtree is not empty, then we have revealed its root node. We now compare this node's key with K. The comparison yields one of three states (4a), (4b), (4c) depending on whether the node's key was greater than, less than, or equal to K, respectively.

If the node's key is equal to K, then we have found the node and we are done. Otherwise, we are in state (4a) or (4b). At this point, there is still a subtree which might contain a node with key K. So, we collapse the other fragments (which we know do not contain K) to obtain the loop invariant in state (2). In this way, we have formed a loop by matching the loop invariant.

In Figures 4 and 5, we show the execution trace of this algorithm on a sample binary tree. (This trace was generated by the Opsis system, which we describe in Section 8.)

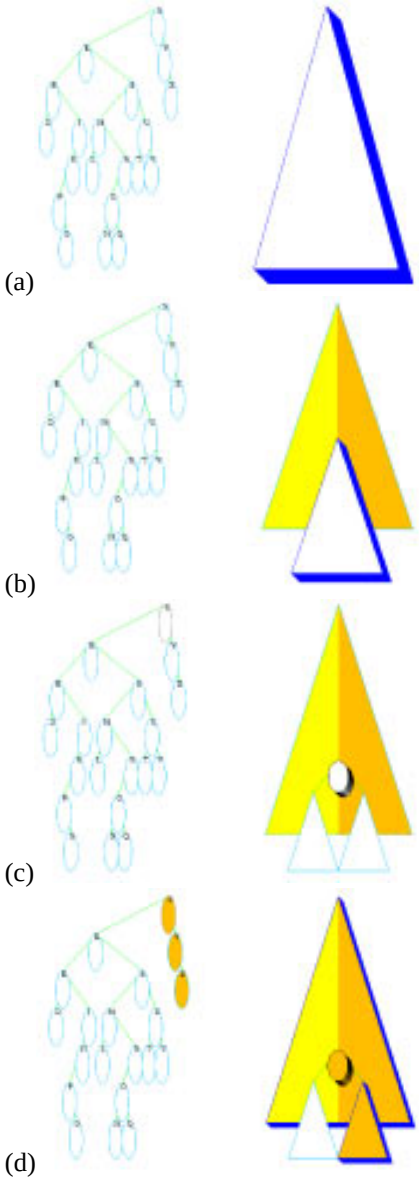

Figure 4. The first four steps in an execution trace of binary tree search. We are looking for a node with an 'M' in the example tree. In part (a), the system is about to add an initially empty fragment above the selected subtree. (Selected fragments are shown with a shadow.) In part (b), the system is about to expand the selected subtree. In part (c), the system is about to compare the key of the selected node with 'M'. In part (d), we see that the node has a key greater than 'M', so the system is about to collapse the node, its right subtree, and the top fragment into a single fragment. In doing so, we will return to the loop invariant state shown in part (b). The algorithm continues in a similar manner. The final step is shown in Figure 5.

6 Visual Binary Tree Manipulation

None of the operations of Section 5 actually change the binary tree in any way. In this section, we present two operations that do make changes.

6.1 Insert Node

The operation *insert* is used to put a new node in a binary tree. The insert operation changes the structure of the tree, so if the user is working with a binary search tree, it may be that the insertion destroys the key ordering property of the tree. To avoid this problem, insertions are only allowed if the user has established that the point of insertion is indeed the right location for the key K. (In Figure 2, (2b), the user has established through assertions about keys in the tree

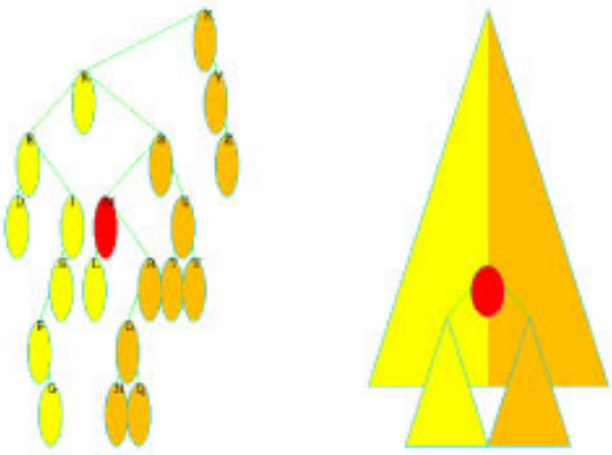

Figure 5. The final state reached in the binary tree search example. (The first four steps of this example are shown in Figure 4.) In the diagram above, the system has found the node with key 'M' and has marked it with very dark shading.

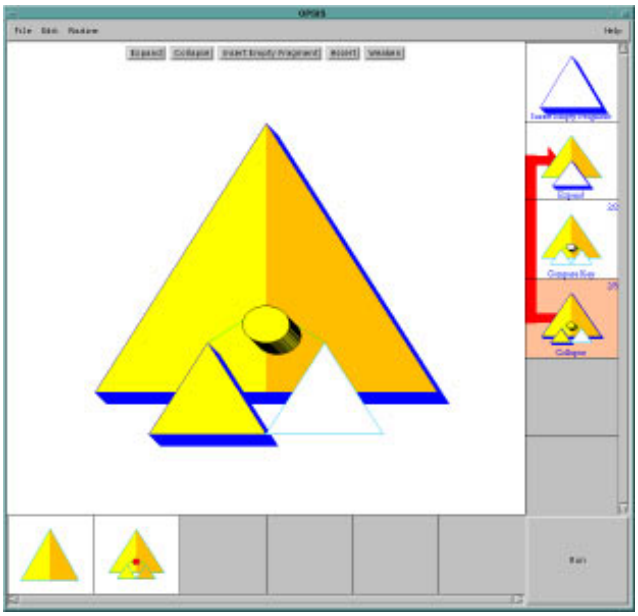

Figure 6. The binary tree search algorithm developed in the Opsis system. Editing occurs on the current state which dominates much of the display. On the right, a sequence of states show a history of the computation that leads to the current state. Observe the arrow in the state history list: this arrow indicates that the operation on the state at its tail of the arrow yields the state at its head (thus indicating a loop). At the bottom, the final states of the computations are shown (i.e., these are states at which the computation terminates).

(shown as light and dark shades) that the insertion point lies on the search path and does not violate the ordering property.)

6.2 Remove Node

The operation *remove* is used to take out a node from a binary tree. If the node to be deleted has at most one child, then the remove operation on that node is sufficient. Observe that taking out a leaf or node with one child cannot destroy the key ordering property of a binary search tree. (In Figure 3, the remove operation is illustrated in states (3b) and (4b).)

If the node to be removed has two children, then the user must activate the remove operation on that node and also on its inorder predecessor (or resp. successor) node. In this case, the operation moves the node information from the inorder predecessor (successor) to the node and removes the inorder predecessor (successor), which has at most one child, from the tree. (As an example of such a case, see Figure 3, (6b).)

6.3 Other Operations

One can add other operations as required that manipulate a binary tree. For example, for implementing balanced binary search trees, we provide "rotate left" and "rotate right" operations. Generally speaking, one should provide operations that are high-level but that do not trivialize the algorithm being taught. Moreover, we provide only operations that maintain the key ordering property of the binary search tree; otherwise, we feel that the operation is too low-level and error prone.

6.4 More Examples

We consider the binary tree insertion and deletion algorithms in Figures 2, and 3, respectively.

In the binary tree insertion visual code, we invoke the search operation (defined in Figure 1) on the subtree. This yields two cases: either a node with key K is found (state (2a)) or no such node exists in the tree (state (2b)). If we found a node with key K, we simply collapse the fragments in the tree (to yield state (4)). If not, we must insert a node with key K. However, in state (2b), we have a subtree with a vertical boundary that separates nodes with keys less than K and nodes with keys greater than K. Thus, we can simply insert the new node since the insertion point lies on the binary tree search path and will not violate the ordering property. Finally, we collapse the resulting fragments to yield state (4).

In the binary tree deletion visual code, we invoke the search operation on the subtree. If no node has key K, then

we are done (as in state (2b)). Otherwise, we have a state with the node with key K explicitly shown (as in (2a)). At this point, we check if the node with key K has less than two children. If so, then a simple "remove" operation suffices. (We check for a left child in states (2a), (3b), and for a right child in states (3a), (4b).) If the node with key K has two children, then we find its inorder successor so that we can delete the node with the more complex form of the operation "remove". We start the search for the inorder successor by adding an initially empty fragment above the node's right child so that the right child is on the leftmost path in that fragment (as shown in states (4a), (5)). Now, using a combination of "expand" and "collapse", we descend left along the nodes until we reach the inorder successor (as is done with the loop in states (5), (6a) which eventually terminates to yield (6b)). Next, we invoke the "remove" operation on the node with key K and its inorder successor to perform the deletion (as is done in states (6b), (9), (2b)).

7 Correctness

The visual programs presented in Figures 1, 2, and 3 are close to also being correctness proofs.

For example, consider the binary tree search program in Figure 1. State (2) at the head of loop (2), (3a), (4a/b), (2) is a visual loop invariant. The proof that this loop invariant holds is also shown visually: in the first iteration, the loop invariant holds vacuously because the fragment inserted is empty; in subsequent iterations, the sequence (2), (3a), (4a/b), (2) preserves the loop invariant (as long as no match is made yet). Thus, the loop invariant in state (2) is proved by induction in a visual way.

From the visual program, we know that if the program terminates, then it will either find the node with key K or assert that no such node exists. However, the program does not show that the algorithm terminates or how long it takes if it does.

Generally speaking, our visual formalism allows the user to prove certain structural properties about the tree structure and also ensures that the key ordering of binary search trees is maintained. However, for complicated algorithms, one may need to prove other properties that are not readily encoded in the tree diagrams. In this case, we recommend that either the student simply not prove such properties, or that he annotate the tree diagrams with English text to fill in the details of the proof.

8 Implementation

We have been working on a system named *Opsis*, which implements the model and domain described in Section 3. (The word "opsis" is Greek for the visual image of an object.) Opsis is a Java applet and can be accessed at http://www.cs.washington.edu/homes/amir/Opsis.html. A snapshot of the system is shown in Figure 6.

8.1 User-Interaction

A user defines a function by starting out with the initial state and by repeatedly invoking operations on one of the final states at that point. (For convenience, the user is always presented with a history of the computation and a list of all final states at every step of the process; see Figure 6.) This process continues until the only remaining final states in the program represent valid return values of the function being defined.

In particular, a user creates a state graph by concentrating on one state at a time. The user selects by mouse some tree fragment(s) in the *current* state and invokes an operation on that selection. Invoking an operation causes the creation of transitions from the current state to one or more other resultant states. One of these resultant states (chosen arbitrarily if there is more than one) replaces the current state on the screen. *Thus, creating a program is akin to designing an "abstract animation" for the algorithm.* Also, if any of the resultant states match one already in the program, then the current state is linked to the one already present; otherwise new states are created as required.

8.2 Mirror Mode

In many balanced binary tree algorithms, such as with AVL trees, one often has to deal with large computations that are simply mirror images of one other. To reduce the programming effort for such computations, the student can go into "mirror mode" in which every state added to the computation graph also leads to addition of its mirror image.

8.3 Annotations

Opsis allows the student to annotate various fragments in the tree diagrams with text. This can be useful in two ways: (1) to fill in additional details of a proof of correctness that are missed by the visual representation; and (2) to prevent false matches of states that should be treated differently (because the visual formalism is insufficient to distinguish them). Annotations represent properties and thus can be disjunctive or conjunctive as explained in Section 5.2.

In addition, the system also allows the user to add information to nodes in the trees. This can be useful for AVL or red black tree algorithms where additional information must be maintained at each node.

9 User Testing

We did some preliminary user testing with four students in a third year data structures class at the University of Washington. Each student was shown a few examples on the system and was then asked to work out the binary search tree deletion algorithm. Afterwards, the student was given a questionnaire to fill out. Here are some typical responses/impressons from the students:

- Although students did not think this system is a replacement for textual programming assignments in a data structures course, they did believe it improved understanding over algorithm animations.
- Similarly, students believed this system can be useful in enhancing written textual proofs by showing key steps in an (abstract) visual manner.
- Several students suggested that the system would be useful in the classroom where the professor could go step-by-step and even retrace, while explaining a written algorithm.
- Students found a non-trivial learning curve at first, but once the concepts were understood, they became very interested in exploring the system further.

We found the user testing to be very useful in making the system more intuitive and understandable. For example, many students were confused about the idea of adding empty fragments to the tree. However, when told that this is done to create an induction hypothesis for a loop invariant, they then understood the concept more readily.

10 Conclusion

In this paper, we have proposed a new way to teach binary tree algorithms through visual programming. We believe that this approach better allows a student to implement an algorithm while concentrating on why it works rather than on low-level implementation details. Moreover, our visual approach not only yields a program but also a proof of some properties maintained or that result from the computation.

Finally, we believe visual programming is not only a very good way to teach binary tree algorithms (and more generally, data structures), but also a good way to teach algorithms in other fields. It would be interesting to explore such possibilities further.

11 Acknowledgments

I would like to thank Steve Tanimoto for careful reading of this paper, helpful suggestions, and encouragement throughout this research. I would also like to thank Rick Hehner for insights obtained from his theory of programming.

Funding for this research was provided by the Natural Sciences and Engineering Research Council of Canada.

References

[1] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *IEEE Workshop on Visual Languages*, pages 4–9, October 1991.

[2] C. Christensen. An example of the manipulation of directed graphs in the AMBIT/G programming language. In M. Klerer and J. Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pages 423–435. Academic Press, 1968.

[3] C. Christensen. An introduction to AMBIT/L, a diagrammatic language for list processing. In *Proceeding of the 2nd Symposium on Symbolic and Algebraic Manipulation*, pages 248–260, 1971.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[5] G. A. Curry. Programming by Abstract Demonstration. Technical Report 78-03-02, University of Washington, 1978.

[6] A. Cypher, editor. *Watch What I Do*. MIT Press, 1993.

[7] E. P. Glinert. PICT: Experiments in the Design of Interactive, Graphical Programming Environments. Technical Report 85-01-01, University of Washington, January 1985.

[8] E. C. R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[9] A. W. Lawrence, A. M. Badre, and J. T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Sympsoium on Visual Languages*. IEEE, October 1994.

[10] H. R. Lewis and L. Denenberg. *Data Structures and Their Algorithms*. Haper Collins, 1991.

[11] F. Modugno and B. Myers. A state-based visual language for a demonstrational visual shell. In *Sympsoium on Visual Languages*. IEEE, October 1994.

[12] B. Myers. Visual programming, programming by example, and program visualization; A taxonomy. In *Proceedings of CHI '86*, pages 59–66. ACM, April 1986.

[13] R. V. Rubin, E. J. Colin, and S. P. Reiss. Think pad: A graphical system for programming by demonstration. *IEEE Software*, 3:73–78, March 1985.

[14] S. D. Smith. Pygmalion: A Creative Programming Environment. Technical Report STAN-CS-75-499, Stanford University, 1975.