Teaching Binary Tree Algorithms through Visual Programming

Amir Michail

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 introduce 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.) We describe our experience with using Opsis in a data structures and algorithms course at the University of Washington. 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* only if their respective abstract visual diagrams match exactly. However, we do allow two states with exactly the same abstract visual diagrams to not be identical if the user so desires. This is allowed because we do not assume that the abstract state completely defines the state of the program at that point. Finally, 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 set of 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*. 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, which are denoted 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.)

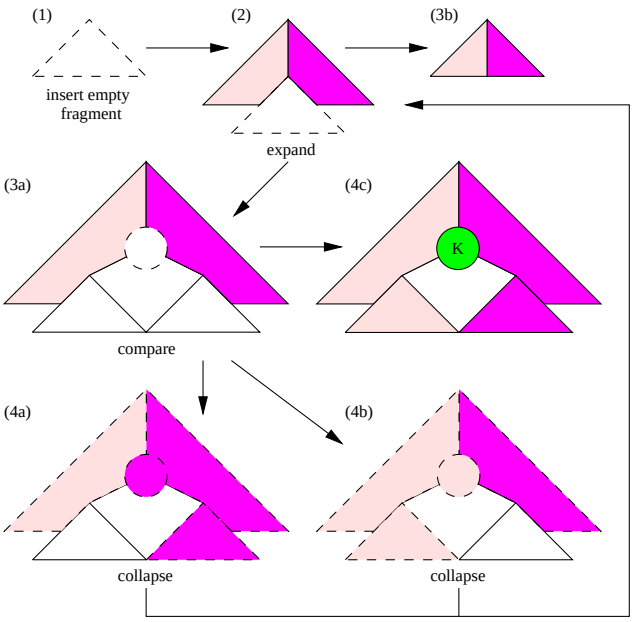


Figure 1. Visual code for binary tree search.

4 Visual Binary Tree Abstractions

In this paper, we describe a visual formalism for implementing dictionary "search", "insert", and "delete" operations 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. We color nodes and fragments to indicate various properties that they may have. For example, pink indicates keys less than a certain key K while the darker magenta indicates keys greater than K. (See Figure 1, (4a).)

A fragment may also be colored pink on the left and magenta on the right. This indicates that the fragment 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

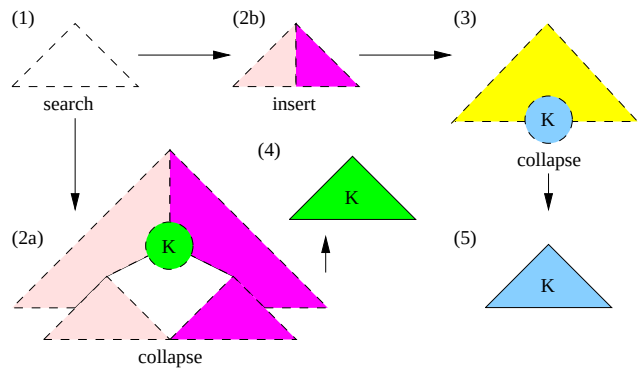


Figure 2. Visual code for binary tree insertion.

K. (For example, see Figure 1, (3b).) The vertical boundary between the pink and magenta 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}.

Intuitively, this path is the path followed if we search the binary tree looking for key K. Consequently, we call it a *search path*.

(In Figure 6, (a), bottom, we show: (1) an abstract tree diagram with the upper fragment consisting of pink and magenta halves; (2) a sample binary tree from the class of trees represented by the abstract tree diagram. In this case, we were searching for the key 'L'. The nodes represented by the vertical boundary (separating the pink and magenta halves) have keys 'O' and 'H'.)

A green node labeled K indicates that the node has the desired key K, while a green 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 nodes and fragments explicitly visible in the tree. This approach is flexible and more natural than using pointer variables (as is done in textual programming languages). There are three main operations for navigation: 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

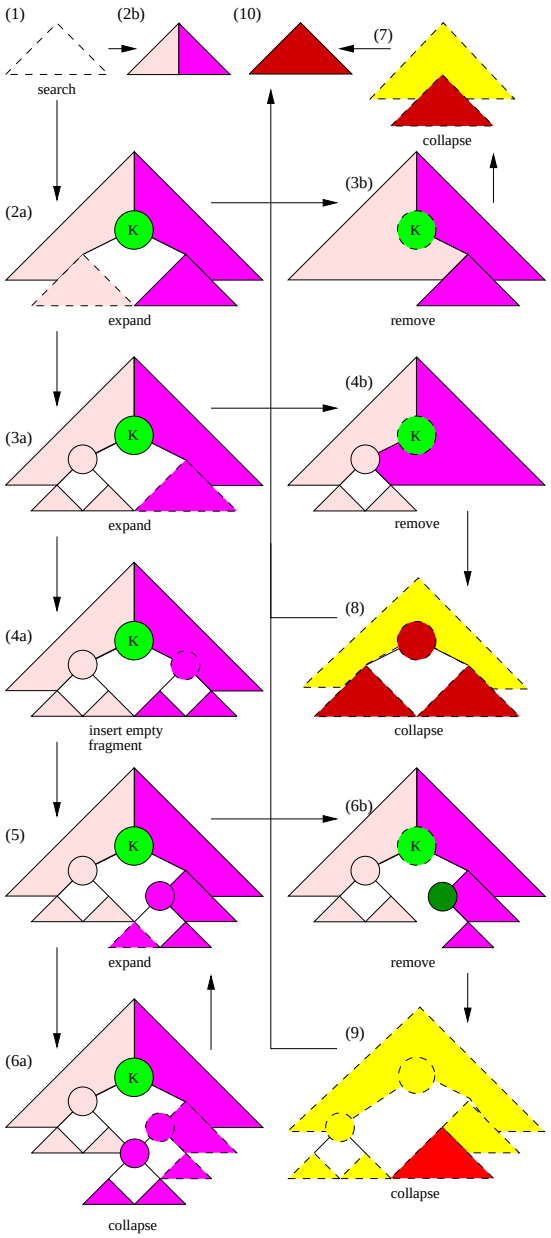


Figure 3. Visual code for binary tree deletion.

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).)

If the fragment is not a subtree then there is some node (or fragment) below it. In this case, the fragment expands by showing the parent of that node (resp. fragment). As that parent may have the node (resp. fragment) 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).) One can also collapse a single node to turn it into a subtree. This can be useful in forming a loop invariant for moving up a tree after the insertion of a node.

The fragments being collapsed may 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.)

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

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 possibly larger class of binary trees 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 a node. (For example, in Figure 1, (1), (2), we add an empty fragment with the property that "no node have key K" and that the fragment's child "is on the search path".)

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. The algorithm is executed on an sample tree in Figure 6, (a).

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 pink while the other half is colored magenta). Moreover, the new fragment's child (i.e., the subtree) is on the search path. 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 Figure 6, (a), 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.)

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 some 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 – that is, that the insertion lies on the search path. (In Figure 2, (2b), the user has established that the insertion point lies on the search path so the he can now insert the node with key K at that point. After the insertion, the search path is no longer required. Correspondingly, the top fragment changes color to yellow which indicates keys not equal to K. We also use blue to indicate a region that includes the inserted node such as in states (3) and (5).)

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). After the deletion, the search path is no longer required. Consequently, the top fragment changes color to yellow which indicates keys not equal to K. We also use red to indicate a region that was previously a child of or containing the node being deleted; see states (7) , (8), and (10).)

If the node to be removed has two children, then the user must first find the inorder predecessor or inorder successor node of K. In this case, the operation moves the node information from the inorder predecessor (or successor) to the node and removes the inorder predecessor (resp. successor), which has at most one child, from the tree. (As an example of such a case, see Figure 3, (6b) and (9).)

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. We also provide "compare info" and "update info" for maintaining additional information in the nodes such as balance numbers for AVL trees. (To see some of

these operations in action, refer to Figure 7 (a), (b), and (c) for sample execution traces of splay tree insertion, AVL tree insertion, and red-black tree insertion algorithms, respectively.)

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. The insertion and deletion algorithms are executed on sample trees in Figures 6, (a) and (b), respectively.

In the binary tree insertion code, we first perform a search (as defined in Figure 1) for key K in the tree. 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 explicitly identified the search path for key K. Thus, we can simply insert the new node. Finally, we collapse the resulting fragments to yield state (5).

In the binary tree deletion code, we again perform a search for key K in the tree. 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) and for a right child in states (3a). The results of the deletion are shown in states (7), (8), and (10).) 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 to perform the deletion (as is done in states (6b), (9), and (10)).

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 structural 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 states 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 Sections 3–6. (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 4.

8.1 User-Interaction

A user implements an algorithm by starting out with the initial state and repeatedly invoking operations on one of the final states at each 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 4.) This process continues until the only remaining final states in the program represent valid results of the algorithm 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

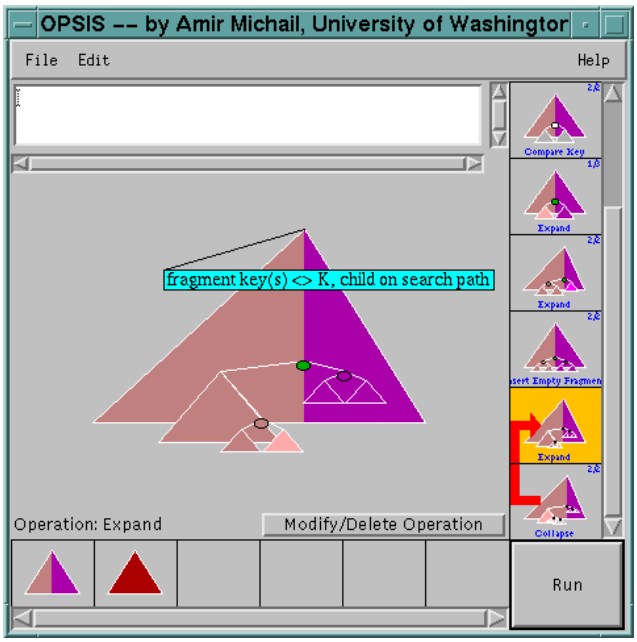


Figure 4. The binary tree deletion algorithm developed in the Opsis system. Editing occurs on the current state which dominates much of the display. On the right, a sequence of states show a history of the computation that leads to the current state. Observe the arrow in the state history list: this 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). The user may add comments to the state in the white region near the top. Finally, observe the bubble help which explains what the various property colors mean.

current state is linked to the one already present *if the user so wishes*; otherwise new states are created as required. One may instruct Opsis to avoid matching two states even if their visual diagrams are identical because the two states are actually different — but the visual formalism is insufficient to distinguish them.

(A sample session using Opsis is shown in Figure 5. In this figure, the user has implemented the binary tree search algorithm: (a) shows the initial state; (b) shows the state obtained after inserting an empty fragment; (c) shows the state in which the key K is found; and (d) shows the case in which the key in the node shown is too small so a collapse operation is performed to match the loop head in order to further explore the right subtree.)

8.2 Imitate Commands

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 may simply implement the algorithm for one case, and then have Opsis automatically generate states for the mirror image case. Specifically, the user can invoke the "imitate mirror image" command on the current state. Opsis then looks for the mirror image state and performs a traversal of the state graph at that location so as to generate the new mirror states for the mirror image case. Although this command is a heuristic, we found it usually leads to the desired results.

It may also be the case that the user may wish to repeat the same code several times in the algorithm. For example, to implement the splay algorithm, one repeats similar code for three cases that arise depending on whether we bubble up the node with key K, its inorder predecessor, or its inorder successor. Moreover, at times we may even want to implement the same code even though the states in each case have trees with slightly different structures. For example, in an AVL insertion algorithm, one might implement the rotations involving the inserted node as a special case, and then use similar code for rotations higher up in the tree (that do not involve the inserted node). To do this in Opsis, the user selects a state, moves to the similar state, and invokes the "imitate selected state" command. This will perform a traversal of the state graph starting at the selected state so as to generate states for the similar case. Again, this command is a heuristic. Unlike the "imitate mirror image" command, this command may ask the user for advice on whether to change information stored at the node in the same way or whether to match a state encountered earlier or to ignore the match. The heuristic usually leads to the expected results.

It may appear that the "imitate" commands are an unnecessarily complicated way to program without functions. Indeed, our motivation for these commands is to allow the

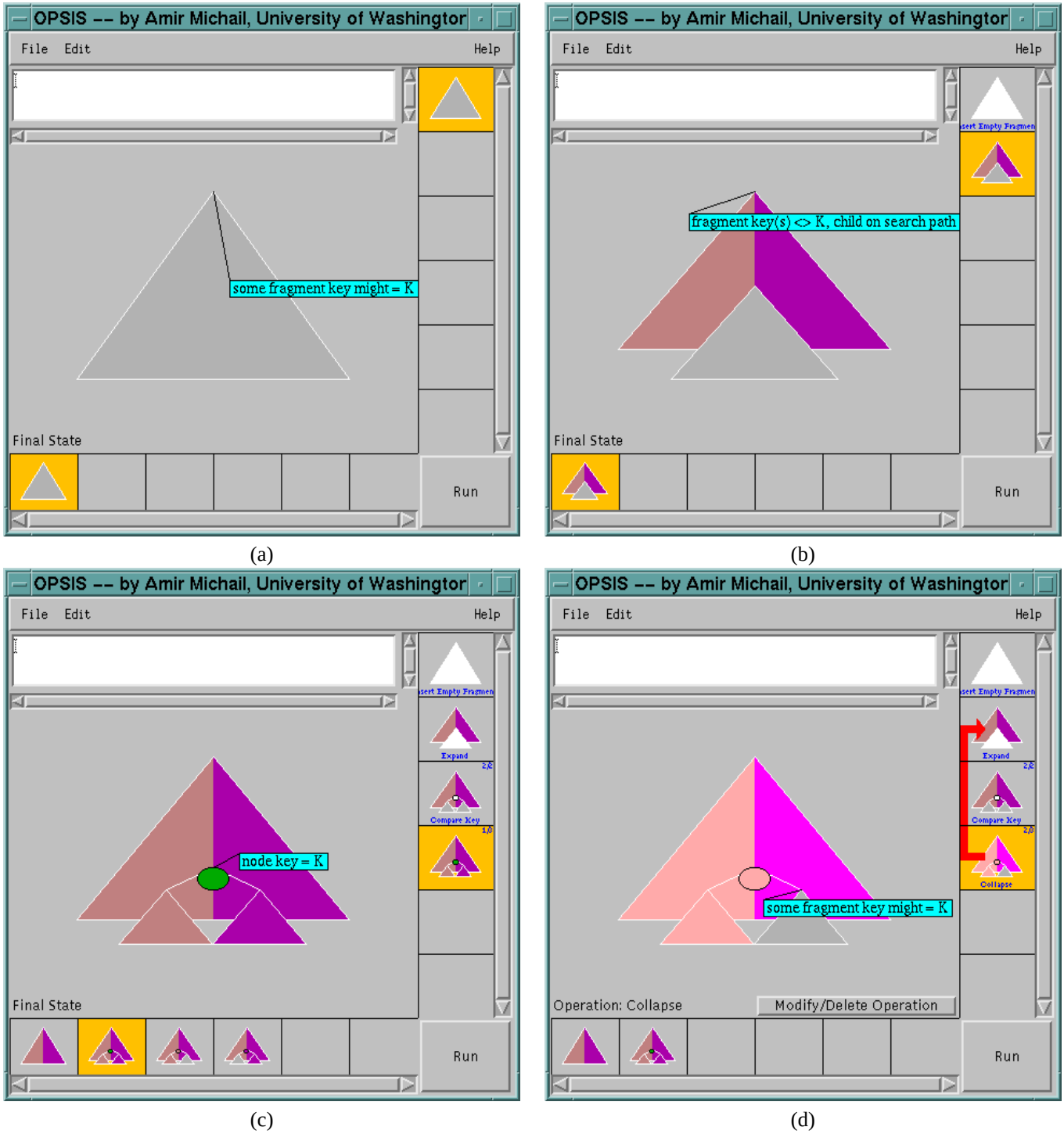


Figure 5. Construction of the binary tree search algorithm in Opsis.

user to program without worrying about structuring their code. That is, they can simply program on a flat state space. There are several advantages to this approach: (1) the student can concentrate on the algorithm rather on how to structure code for the algorithm; (2) the visual abstractions can be kept simple as there is no need to represent functions (or their parameters) and modules; and (3) the user need not consider all cases at once, but just work on one at a time.

9 User Testing

We have done some preliminary user testing with students taking CSE 373, a third year data structures course for non-majors at the University of Washington. Students in this class typically major in engineering, math, or science.

Students in CSE 373 were required to do a final project with one of the possible topics involving Opsis. In the Opsis project, the students implement two balanced binary search tree algorithms of their choice. (For example, AVL insertion counts as one algorithm.) Both algorithms could be done in Opsis or one in Opsis and another in a standard textual programming language. (Most students chose C++ as a textual language.) Students could work alone or in pairs.

Students were given a tutorial and documentation on the basics of the Opsis system. However, we did not explain to them the balanced binary search tree algorithms; they had to understand them on their own from various texts.

Ultimately, eight groups chose to do their final course project using Opsis. Three of these groups consisted of pairs. The following observations are based on the student reports and a questionnaire. Although our user testing did not include enough students for a statistically significant study, we were able to obtain insight into the experiences students had with the Opsis system.

9.1 Learning Curve and Implementation Speed

One of the reasons we built Opsis was to provide a way for students to implement complicated tree algorithms without worrying about low-level implementation details. In particular, students need not perform any pointer manipulation nor structure their code in any way (eg., using functions or modules). In so doing, we hoped that students would concentrate more on the actual algorithm rather than on implementation details.

We were partially successful. Although students indicated that Opsis had a non-trivial learning curve, they also said that once they understood the system, they were able to implement tree algorithms quickly.

Students typically spent 1–5 hours to learn Opsis. Of course, there was almost no initial learning period for textual languages as the students were already quite familiar with them.

After the initial learning period, students spent about about 2–20 hours per algorithm using Opsis. Simpler algorithms such as Splay tree algorithms took about 2–10 hours while more complicated algorithms such as those for AVL tree took between 7-20 hours.

Surprisingly, students spent roughly the same time with textual programming languages. However, one group was not able to implement the Splay tree algorithms in C++ at all. Time for the AVL tree algorithms varied from 2–20 hours. (The students did not report any times for the splay tree algorithms implemented using textual code.)

We expected the Opsis implementation times to be significantly less — not roughly about the same as those for textual code. However, there is tremendous variation in the times observed – probably a result of the variation in the students' abilities and background. Thus a much larger sample is necessary to test this hypothesis. Moreover, some students copied textual code fragments from books. Others experienced technical problems with Opsis, which were eventually resolved.

9.2 Learning Visually versus Learning Textually

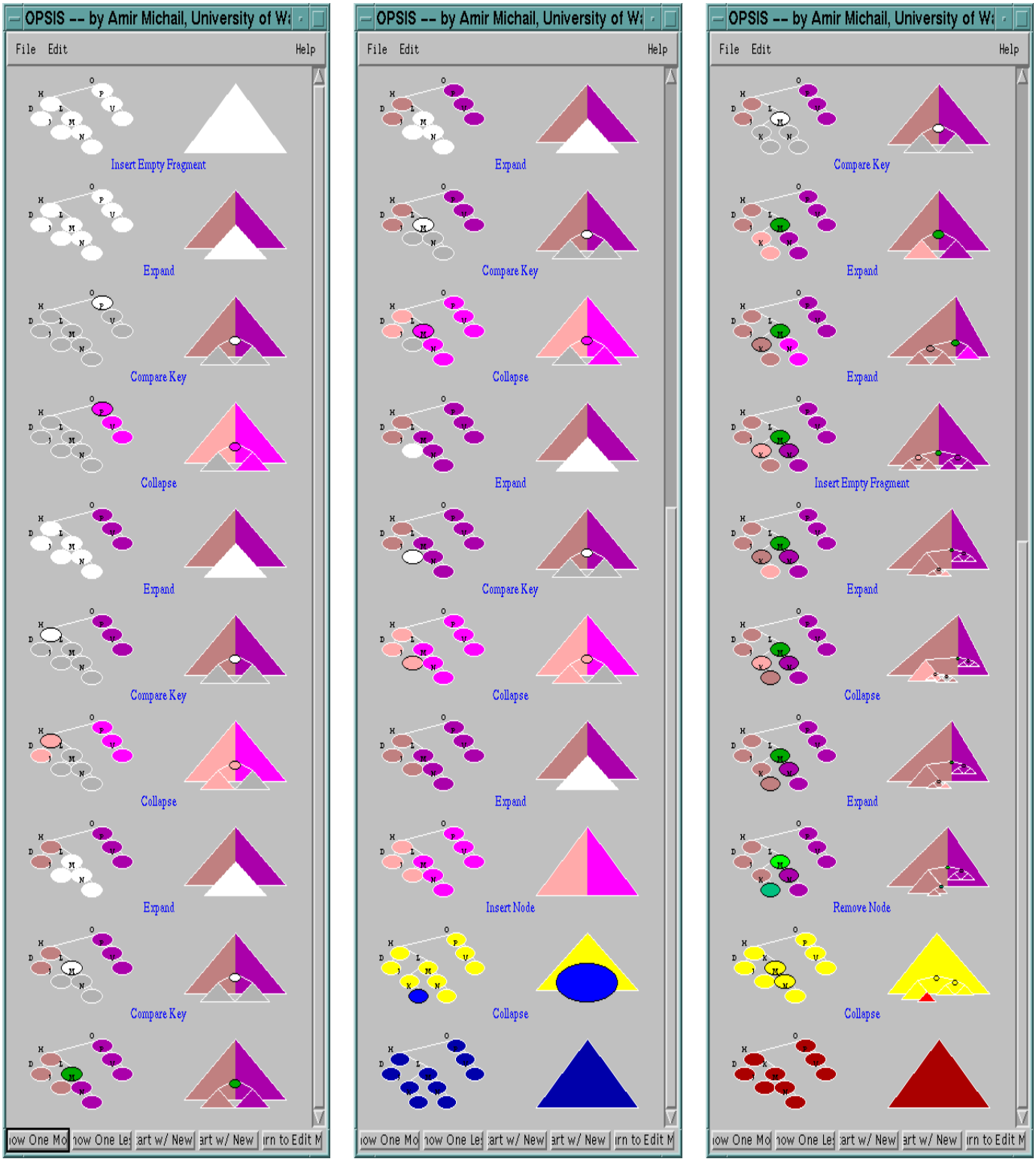
As mentioned earlier, we did not have enough students for a statistically significant study. However, from our experience, we found little difference in how well students learned algorithms in Opsis versus textual programming languages. Of course, students who implemented the algorithm using a textual programming language had a better idea of how to move pointers about in a low-level implementation. On the other hand, for some students the textual implementation took longer or was too difficult to do.

9.3 Proof versus Programming

One interesting aspect of Opsis is that it completely blurs the distinction between programming and proof. Certainly, the creation of loop invariants and structural induction arguments is a process close to proof. Yet, other operations feel like programming.

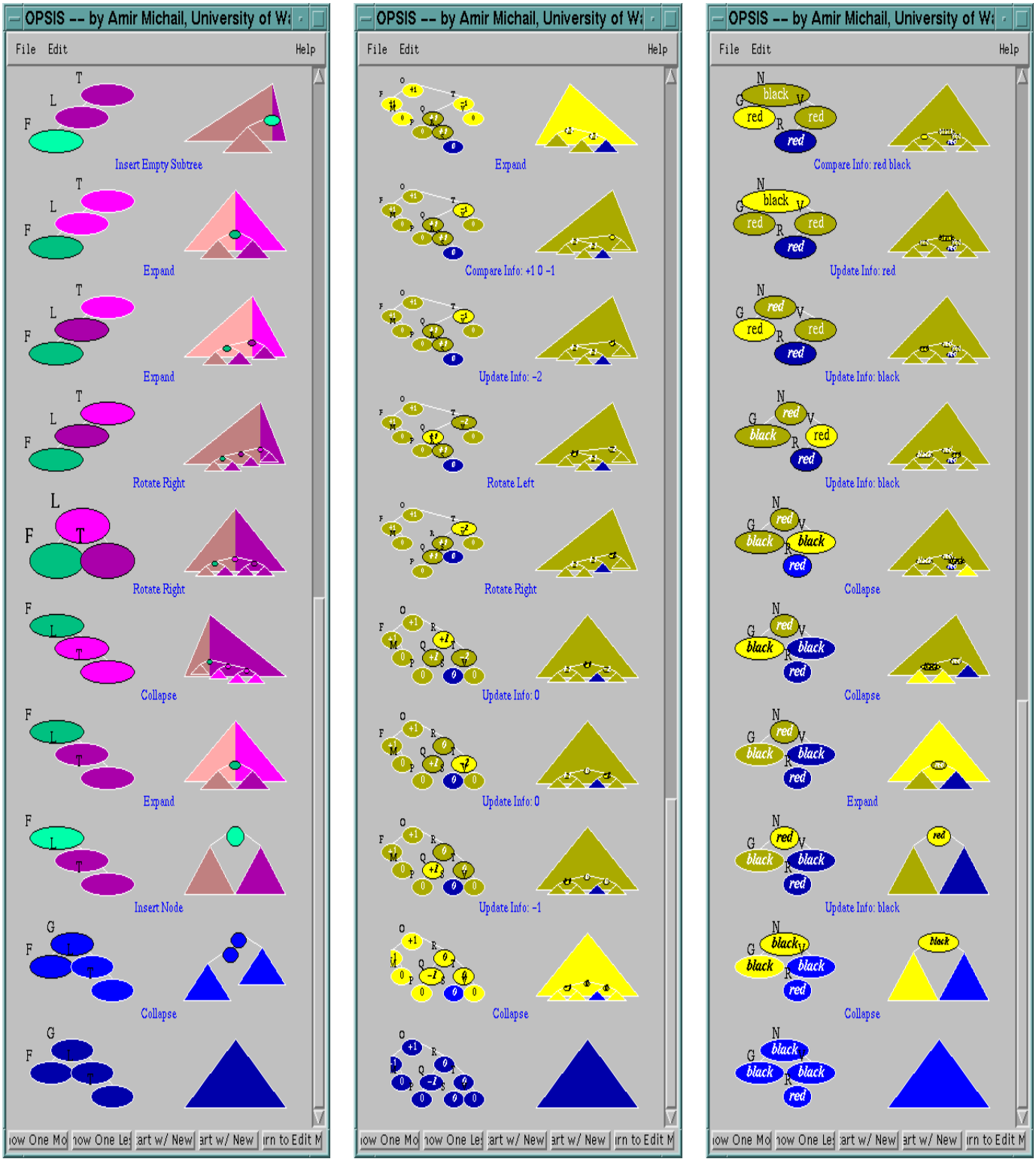
We found it interesting that most students said implementing algorithms in Opsis felt more like programming than proof. However, one student disagreed: "most of the time I am programming in Opsis, I treat it as a 'proof"'. Interestingly, this student, more so than the others, felt that his extensive programming experience in textual languages was a hinderence to using Opsis. Thus, the move to Opsis made the system appear very proof oriented.

Possibly another reason why most students saw implementation in Opsis as mostly like programming is that they



(a) (b) (c)

Figure 6. Opsis execution traces. The algorithms shown are: (a) binary tree search; (b) binary tree insertion; and (c) binary tree deletion.



(a) (b) (c)

Figure 7. Opsis execution traces. The algorithms shown are: (a) splay tree insertion; (b) AVL tree insertion; (c) and red-black tree insertion.

were unsuccessful in forming loop invariants for complicated algorithms. Most of their experience was with parts that felt like programming.

9.4 Debugging

Strictly speaking, we did not need to allow students to run their algorithms on examples. This is because an Opsis program, when properly commented, should be close to a proof of correctness for the algorithm. However, Opsis doesn't completely enforce correctness during implementation so mistakes may still occur. (Although Opsis enforces low-level correctness dealing with pointers and tree structure, it doesn't enforce high-level correctness dealing with information stored at the nodes or amortized analysis arguments.) Moreover, we felt that showing the student the correspondence between the visual states and the concrete states during execution would be helpful in clarifying the abstractions and the algorithm itself.

Indeed, the ability to execute a program turned out to be quite important. Many students indicated that half of their time was spent programming new code while the other half was spent debugging (i.e., repeated execution). We believe there are primarily two reasons for this. First, students like incremental development even if they have a proof of correctness – they would still like to check the proof. Second, students obtain a sense of satisfaction from executing code.

9.5 Visual Clutter

Several students complained that it was difficult to keep track of all the graph states particularly for the more complicated AVL and red-black tree algorithms. We anticipated this problem and created the final and history state lists to alleviate the visual clutter. However, one student felt that this restricted view to the state graph was a actually a problem because it was easy to become lost in the numerous algorithm states. She said "When a change was made and I progressed to the next state it was almost impossible to find where I had previously been working."

Another student who was implementing the red-black tree insertion algorithm complained that Opsis required more states than necessary because: (1) it kept track of which subtree the key was inserted in which is not necessary for the red-black insertion algorithm (but necessary for the AVL tree insertion algorithm); and (2) it didn't allow one to consider nil pointers as black nodes (which would reduce the number of cases in the red-black tree algorithm). Although it is easy to automatically generate these additional states by the "imitate" commands, we can see how new users of the system may view this as unnecessary work and a factor in the visual clutter problem.

9.6 Guidance versus Flexibility

One of our goals with Opsis was to provide the student with some guidance without giving away the solution to an algorithm. This principle permeates throughout our design. For example, we hide the details of pointer manipulation, provide functions to perform rotations automatically, keep track of which subtree an insertion occurs in, and keep track of where a deletion occurs.

Another goal was to allow the student to program without creating any structure in the code. That is, programming is done on a flat state space without any functions or modules. The purpose for doing this is to allow the student to concentrate only on the algorithm and not on how to structure code to implement the algorithm. In addition, we provide "imitate" commands to handle similar cases automatically in the flat state space. By avoiding program structure, we "guide" the student towards the algorithm and away from structuring code. (That is, students can't structure code even if they want to.)

Unfortunately, this guidance has a price — flexibility. Several students complained that they had very little freedom in the way they implemented an algorithm. One student wrote "it has more guideline than a high-level programming language such as C++ but it has less flexibility for the programmer." (Indeed, most implementations for the same algorithm were very similar.) Although we view this as an advantage of using Opsis, it turns out that this lack of freedom hinders the progress of some students.

9.7 Graphical Abstraction

The learning curve mentioned earlier is partly a result of the graphical abstraction. Most students found the abstract states confusing at first but became proficient with them after some experimentation. One student wrote "It takes some getting used to think in tree fragments. Once I am in the 'mode', I don't have any problem with the abstract representation of the tree structure." A few students were confused about how a state before an operation corresponds visually to one after the operation. (This is probably due to the way the fragments move about automatically when the tree is changed.)

Students generally agreed that the graphical representation allowed them to "see" the algorithm clearly. One student wrote "The ability to see an algorithm take shape on the computer screen was an incredibly valuable learning experience." Another wrote "the ability to manipulate abstract tree fragments versus dealing with the switching of pointers makes the algorithm much more understandable and easier to learn".

Some students found their extensive experience in textual programming languages made it harder to learn Opsis.

A student wrote "All the books I have read and the classes that I have attended were all in textual languages, so I can think much faster in a textual language than I can in Opsis." On the other hand, students who said they think better visually found the graphical abstractions to be very valuable.

9.8 Loop Invariants

We expected students to have trouble with building loop invariants while implementing algorithms. However, we believed this to be such a valuable activity that we made loop invariants very explicit in Opsis. Indeed, programs implemented in Opsis are essentially structural induction arguments with the visual loop invariants being the induction hypotheses.

As there is no automated way to come up with a good loop invariant, the process involves guessing and intuition. One student wrote "On the first try, I had lots of trouble building loop invariants." The problem was particularly acute with the more complicated AVL tree algorithms. Indeed, no group was successful in implementing a complete AVL tree insertion algorithm because they couldn't come up with a suitable loop invariant. (Their implementations only handled cases where the rotations are done at the base of the tree just after insertion.) However, students were able to do the simpler splay tree algorithms.

We believe explicit loop invariants clarify the algorithm and make its correctness proof more apparent. One student wrote: "finding the loop invariants helped to solidify the recursive nature of binary tree algorithms in my mind and the graphic nature of the program allowed me to actually see the loop."

9.9 Demonstrating Concepts

Although our motivation behind Opsis was to allow students to implement algorithms through visual programming, several students suggested other uses. In particular, students suggested that Opsis might be useful for demonstration. For example, a professor might demonstrate an algorithm in class by using Opsis. Unlike standard algorithm animations, an Opsis demonstration can be more interactive and can actually show the students the abstract loop invariants and structural induction arguments for the algorithm. In this way, Opsis can be useful in enhancing written textual proofs by showing key steps in an abstract visual manner.

9.10 Teaching Non-majors

Most universities have computer science courses for students majoring in other disciplines. From our experience with CSE 373, it is clear that some of these students do not have the same programming abilities as computer science students. Consequently, Opsis may be particularly suitable for such classes as it allows the professor to assign complicated algorithms in such a way as to avoid laborious low-level implementation details that would otherwise discourage students. (Of course, we do not advocate only using Opsis in such courses; implementation with a textual programming language is still essential.)

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.

To validate our approach, we have built a Java applet, named Opsis, that demonstrates the ideas in this paper. In addition, we have performed a user study in a data structures course. From this experience, we found that although Opsis has a non-trivial learning curve, students were able to learn and use the applet to implement the various algorithms. We also found students felt the system clarified the algorithms by making tree structure and loop invariants explicit visually. It would be of interest to conduct a more extensive study to determine how much faster (if at all) students can implement algorithms with Opsis as compared to textual programming languages.

Finally, we believe visual programming is not only a good way to teach binary tree algorithms (and more generally, data structures), but also a promising 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 like to thank David Notkin for reading a later version of this paper. 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.