Tidy Animations of Tree Algorithms

Technical Report GIT-GVU-92-11

John T. Stasko
Carlton Reid Turner

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

E–mail: {stasko,turner}@cc.gatech.edu

Abstract

In software visualization and algorithm animation it is important that advances in system technologies are accompanied by corresponding advances in animation presentations. In this paper we describe methods for animating tree manipulation algorithms, one of the most challenging algorithm animation domains. In particular, we animate operations on pairing heap data structures which are used to implement priority queues. Our animations use tree layout heuristics and and smooth transitions for illustrating intermediate algorithm states to promote viewer understanding. This paper describes the visual techniques and methodologies used to display the pairing heap operations. The paper also details the implementation requirements and how our particular support platform, the XTango system, facilitates meeting these requirements.

1 Introduction

Algorithm animations[Bro88b] visually depict how algorithms function to promote understanding of the algorithm's methodologies. Usually, the animations contain abstract views of an algorithm's semantics and operations. Recently developed algorithm animation systems[LD85, Bro88a, BB90, Sta90b, BK91, Bro91] primarily have advanced the technology exhibited in algorithm animations, but as the area of algorithm animation matures, these technological system advances must be accompanied by advances in the quality of algorithm presentations. Researchers must focus on providing *effective* algorithm presentations that help communicate the purpose and tactics of sophisticated computer algorithms.

Most of the systems referenced above provide a variety of views of sorting, searching, and graph algorithms. Certain other types of algorithms have proven to be more challenging to animate. A particular class of algorithms considered to be one of the most difficult to animate is the dynamic tree algorithm. This is especially true when the algorithm's input is not predefined or hard-wired and when its animation is dynamic and real-time. Animating tree algorithms is particularly difficult because they challenge the two intrinsic aspects of algorithm animation: layout and action.

Graph and tree layout are known to be challenging problems with an extensive list of research articles on the topic[TE88]. For tree layout algorithms, specifying the placement of tree nodes to avoid edge crossings and to preserve aesthetics is critical. Various methods have been designed for optimally positioning the nodes in a static tree. In algorithm animation, this layout problem is further compounded because unpredictable run-time input causes the tree to grow and shrink. An algorithm animation must reflect the correct tree state without undue delay.

The action component of a tree algorithm animation is challenging because different parts of the tree structure must update synchronously in complex motion sequences. It is critical that the animation view support continuous changes in state with smooth, incremental transitions during operations to help maintain the viewer's context and to explain how the algorithm is operating.

This paper describes animations that we have developed of pairing heap priority queue algorithms. Pairing heaps are tree data structures that are appealing due to their conceptual clarity and their nearly optimal computational complexity[FSST86]. We describe the animation methodology used to animate the heap algorithms that, we believe, provides an effective presentation of the algorithms, promoting and facilitating understanding. We also describe how the animations were implemented using the XTango system, and how XTango's facilities supported the animation design.

2 Problem Domain

Priority queues are data structures that manipulate nodes with key values and that support the operations insert, find min, delete min, delete, and decrease key[AHU74]. Priority queues are widely used with applications such as job scheduling, minimal spanning tree, shortest path, and graph traversal. By using a simple heap data structure, worst case time bounds of $O(\log n)$ are achieved for all operations. Fibonacci heaps[FT84], developed by

Figure 1: Comparison-link action between the two trees. The "losing" tree becomes the new first child of the "winner."

Fredman and Tarjan, achieve amortized[Tar85] time bounds of $O(1)$ for insert, find min, and decrease key and $O(\log n)$ for delete min and delete. Currently, these bounds provide the best running times for a variety of different problems[FT84]. Unfortunately, Fibonacci heaps are quite complex and exceptionally difficult to implement. As a result, they are rarely used in practice. Pairing heaps offer an attractive alternative implementation option, nearly achieving the optimal amortized time bounds of Fibonacci heaps[FSST86, SV87], yet providing a more manageable and less contrived data structure to implement.

Pairing heaps are implemented as multiway trees with the heap property invariant that the children of a node have key values greater than or equal to the node's key value. A number of pairing heap variants exist. This paper focuses on the two-pass variant, named after its delete min operation's methodology.

The two-pass pairing heap algorithm maintains one tree with the minimum key value in the root. The *comparison-link* action is fundamental to all the pairing heap operations. In a comparison-link, two nodes are compared to determine which has the smaller key value. The node with smaller key value "wins" the comparison, and the larger key-valued node is demoted as the new first child of the winning node. The larger key-valued node retains its own children and it becomes a sibling of all the previous children of the winner. Figure 1 shows the result of a comparison-link between two subtrees.

Operations on a two-pass pairing heap work as follows:

- Insert: The algorithm's insert operation simply comparison-links the new node with the tree root node.
- Delete min: The delete min operation removes the root and elevates all its children as a forest of new trees. These nodes are reformed into one tree by a two-pass comparison-link procedure: Nodes are compared in non-overlapping pairs left-to-right, then the

Figure 2: A multiway depiction of a pairing heap (left) and its corresponding binary representation (right).

rightmost tree is repeatedly linked to its left sibling in a right-to-left pass.

- Decrease key: To decrease a node's key value, we remove it and its children from the heap. Since the new value may now be less than the original root, the removed subtree is comparison-linked with the previous root to reform one tree again.
- Delete: To delete a node, we remove it, elevate all the node's children, reform them into one tree using the two-pass method, then comparison-link this tree's root to the previous root.

Because of the difficulties in implementing multiway tree data structures, pairing heaps are often implemented using a binary tree simulation of a multiway tree. This representation maps a node's first child (multiway) to its left child (binary) and its next sibling (multiway) to its right child (binary). Figure 2 shows both the multiway and corresponding binary representations of a tree. The animations we describe in this paper reflect the binary tree representation.

3 Animation Methodology

In this section, we describe how the pairing heap algorithm animations look and work. The animations are implemented using the XTango system[SH90], a derivative of the Tango algorithm animation system[Sta90b]. XTango differs from Tango in that it runs directly on top of the X11 Window System, and it has a simpler architecture model to promote portability and ease-of-use.

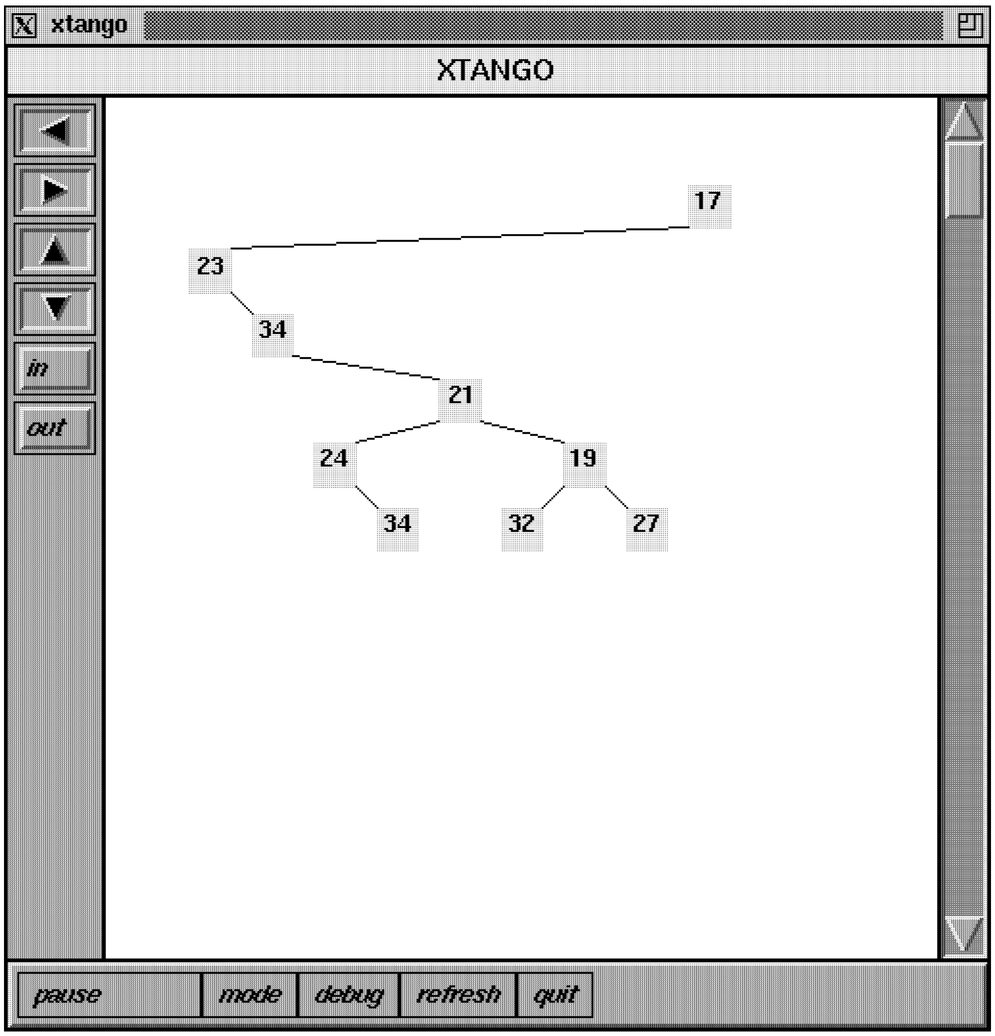

Figure 3: Animation frame from the XTango pairing heap algorithm animation using the conservative layout technique.

3.1 Representation

In our animations, tree nodes are presented as colored rectangles with the root at the top center of the animation window. Left and right children are located below and to the appropriate sides of their parents. Related nodes are connected by thin lines between the closest corners of the two nodes, and each node's key value is superimposed upon its image. Figure 3 shows a frame from the pairing heap animation between operations.

The motion of nodes and subtrees in the heap operations is carried in a smooth animation sequence by incrementally moving objects from their existing positions to their desired positions. For instance, when a forest of subtrees is created by one of the pairing heap operations such as delete min, all the roots of the subtrees ascend smoothly to the level of the original root. All subsequent comparison-links occur with nodes or subtrees at the same height as the root of the tree. The action of the two-pass algorithm for recombining a forest of subtrees is clearly depicted by the animation. The trees are shown being combined in a pairwise fashion on the first pass. On the second pass, the right most tree continually combines with its neighbor until the final heap slides into place in the center of the animation window. The animation was explicitly designed to show the two passes sweeping across the forest creating a new tree.

These smooth portrayals of updates differentiate our animations from those that typically only show the state of the tree after each operation in one comprehensive view update. Animations with smooth updates showing intermediate states help to preserve viewing context and convey the actual algorithm used to manipulate a tree.

3.2 Layout Techniques

The problem of tree layout has been studied extensively[WS79, Vau80, RT81, WI90, Moe90] and many node positioning algorithms exist. These algorithms primarily deal with determining optimal positioning for static trees. Animating tree operations during a program's execution is quite different, however, requiring dynamic updates of node positions based on local changes. It may be desirable to sacrifice optimal node placement for features such as preservation of context and the highlight of particular algorithmic operations.

In a pairing heap, all the operations can be broken down into a series of simple linkings of subtrees. By linking each subtree as close as possible to its parent we can simplify the positioning requirements and avoid making two complete traversals, which are typically required for optimally positioning a static tree, every time a node or subtree changes location. Such a methodology may be computationally expensive for real-time animation, and more importantly it may actually detract from the visual appearance of the animation. Small changes to the tree that modify non-local positionings will disrupt viewer context and make it more difficult to follow an animation.

The animations we have developed can utilize two possible supporting layout techniques. The first technique (we call it *conservative*) creates trees whose steady-state appearance looks like those created by Knuth's algorithm that positions each node according to its position in an inorder traversal[Knu71]. Figure 3 reflects this technique. The second technique, developed specifically for these animations produces trees that we found to be more aesthetically appealing than those of the conservative technique. (This was a subjective opinion based on our own personal tastes.) Trees produced by this approach (we call it *natural*) are more compact and tend to be more balanced than those produced by the conservative approach. The figures at the end of this paper reflect the natural technique. Both layout methods render trees with basic "tidy" properties[WS79]: parents are drawn above children; nodes on the same level lie on a horizontal line; left children are drawn to the left of their parent (and vice-versa).

To facilitate both layout techniques, local positioning information is encoded using the following strategy: For each node, two successor-width values are defined. All leaves have successor-widths of 0-0 which corresponds to the width of their left and right subtrees. A node whose subtree is a single leaf will have a successor-width of 0-1 or 1-0 based upon whether the leaf is a right or left child. The two layout techniques differ in how the successor width is defined for a general node. For the conservative scheme, a node's left successor width value is determined by adding 1 to the sum of its left child's successor-width values.

Likewise, the right successor-width value is 1 plus the sum of the right child's successor width values.

In the natural scheme, the left successor width value for any node is defined to be the maximum of (1 + left child's left successor width value) and (the right child's left successor width value - 1). Similarly, the right successor width is defined as the maximum of (1 + the right child's right successor width) and (the left child's left successor width - 1). Once the local positioning information is defined, it is used during the comparison-link action to guide the horizontal placement of each subtree. The root of each subtree is positioned as close to its parent as possible during linking. If a subtree is linked as the right child of a node, then it is positioned with its root located $n + 1$ units to the right of its parent, where n is the left successor extent of the node and a unit is a fixed, arbitrary spacing between nodes. A fundamental difference between the two approaches is that the natural approach allows a subtree to extend underneath its ancestors while the conservative approach does not.

What is critically important here is that our technique's updates modify only local portions of the tree. We do not traverse the entire tree to perform an expensive update. Were we to strictly enforce a static layout strategy, all the small modifications that happen during the heap algorithm would result in a constantly changing, difficult-to-interpret tree structure. Operations such as decrease key, which removes a subtree from the heap, do not modify the surrounding structure of the heap near the removal, thereby avoiding a ripple up the tree. Consequently, over time the trees in our animations may not exhibit the pure appearance of an absolute layout heuristic. That is, the trees may not always support the other tidy properties such as having subtrees appear the same no matter where they are.

We believe that such differences from the "tidiest" layout techniques are more than compensated for by the benefits of this approach: Our update methodologies are fast, they produce natural-looking trees, and most importantly they preserve viewing context from operation to operation.

3.3 Animating the Operations

In designing the pairing heap animations, we strived to create actions that would be natural in appearance and that would activate the pattern matching capabilities of human viewers. Naturally, the static nature of the figures in his paper does not do justice to the animations.[1] Nevertheless, below we describe some of the key methods used in the animations.

When the animation presents a comparison-link between two nodes, it utilizes an invariant that we have labeled "loser moves." A comparison between the lowest key values of two trees in a forest is presented by flashing the root nodes of the two trees in alternating colors. The node with the losing (larger) key value, reverts to the standard node color and smoothly incorporate itself into the winning (smaller key) node's tree. The winning node remains stationary and is highlighted by the alternate color until the losing node is integrated into the winning tree. If the right tree of the pair won the comparison, the consolidated tree moves to the original position of the left tree. (A win by the left tree re-

[1]We have prepared a videotape of the heap animations that provides a much better impression of their dynamic behavior.

quires no post-merge movement.) Enforcing this "loser moves" invariant brings consistency to the animation, promoting comprehension through pattern recognition by the animation viewers.

Another important feature of the animations is smooth motion of subtrees as rigid structures. Rather than move a subtree one component image at a time, all the images which comprise the subtree move along the same path simultaneously. Rigid tree transitions provide visual animation sequences that are consistent with the logical operations on the pairing heap. In the underlying data structure, an entire subtree is linked as the child of a node at once by the single operation of assigning a child pointer inside the parent node. This semantic component of the algorithm operation is reinforced by the visual effect of manipulating subtrees as rigid structures.

The delete and decrease key operations in the animation are interesting because they allow viewers to interactively select the affected node by using the mouse. The animation reports the selection back to the underlying heap program which then carries out the operation. This "hands-on" capability helps to unify the user of the system with the animation being presented.

One difficult problem in animating tree algorithms results from the dynamics of tree growth. Because input is received "on-the-fly" at program execution time, it is not possible to predict how the tree will grow or shrink in the future.[2] Certain sets of input operations might make the tree grow rapidly in one or more directions and exceed the boundary of the animation view.

Our animations address this problem by zooming the animation view. XTango animations are implemented on top of a real-valued world coordinate system that is automatically mapped to integer pixel coordinates by the system. XTango includes a simple "zoom" function for altering the world coordinates that correspond to the animation window boundaries.

Our tree animations incorporate the XTango zoom feature to cope with growing trees. After every operation, the positions of the tree's extremities are measured and compared with the coordinate boundaries of the animation window. When a node exceeds a coordinate boundary, the animation smoothly zooms out: the coordinate boundaries are increased and the representation of the tree shrinks to again fit within the animation window. When subsequent pairing heap operations cause the tree's extremities to recede within the original coordinates, the animation automatically zooms in and the tree's components grow to their normal size.

Viewers of the animations have found this visual effect to be aesthetically appealing, and it helps preserve the context of the animation's operations and view. Of course, this technique goes only so far. After about three zoom-outs the nodes become too small to discern. To display extremely large trees, other visualization techniques such as tree-maps[JS91] are better. For learning the details of an algorithm, however, small to medium size trees are most useful, and our zoom technique works well in these cases.

[2]By using a two pass animation in which the first pass simply reads all input, it is possible to plan for future tree growth and do "smarter" layout. Our animations do not assume this capability—they must adapt instantaneously to reflect program operations as they initially occur.

4 Tree Animation Algorithms

To create an algorithm animation with XTango, a programmer must 1) augment the program being animated with *algorithm operations*, parameterized event calls that activate sets of animation routines 2) define the animation routines to represent the program operations using the XTango implementation of the path-transition paradigm[Sta90a]. The animation paradigm is based on four simple abstract data types (image, location, path, and transition) and their operations. XTango implements the data types in the C programming language using the X11 Window System.

In the pairing heap animation, we used four algorithm operations named Add, Pop, Delete and DecreaseKey. We placed the algorithm operations into the pairing heap source code at the appropriate points to inform the animation component about the heap algorithm's actions. The operations map to three animation routines, AnimAdd, AnimPop, and AnimReduce which are defined in an independent source file. These three routines make extensive use of routines named AnimComparisonLink and AnimTwoPass. An additional animation routine, AnimBounds, is called after every pairing heap operation to manage the zoom facility.

To remove any dependence of the animation routines on the underlying pairing heap implementation, the heap data structure is mirrored by a similar data structure in the animation component. An animation structure node contains the following XTango objects: the image of the node rectangle, the image of its key value, and the images of links to right and left children.

This duplication of some program data structure is valuable because it supports a clean separation of the underlying program and its animation. If the animation routines were to access program data structures, their reuse for animating other tree algorithms would be restricted. By not relying on program data, just the algorithm operations and their parameters, our tree animation description code can be reused with minimal changes for animating other variants and algorithms.

To help explain how the animation routines are implemented, we describe the delete min operation's animation, a few key frames of which are shown in Figures 5-8. Delete min, signified by the Pop algorithm operation, removes the lowest valued node (the root) from the pairing heap. AnimPop animates this action with the following sequence: First, the root node image disappears from the animation view. Next, the left child of the former root severs its link to its right child and ascends with its left subtree to the location formerly occupied by the root. In turn, each of the orphaned right children ascends with its left subtree creating a forest of subtrees with root nodes at the same level. Finally, the AnimTwoPass animation routine is called to animate the reformation of the forest into a pairing heap.

AnimTwoPass presents the two-pass reformation operation on a forest of subtrees that are created when a node is deleted. This animation routine does a left-to-right comparison-link pass by calling AnimComparisonLink on each pair starting from the left. Next, a right to left pass is made with the animation again carried out by AnimComparisonLink. The result of the right-to-left pass is a new pairing heap.

In Figure 4 we provide pseudo-code for the AnimComparisonLink routine. (*Left* and *Right* refer to the root nodes of the left and right subtrees being linked.)

```
BEGIN
IF the pointer to Right is a null pointer, RETURN
Create and perform transition to cause Left and Right to
  flash and to remain in the highlight color
IF Left has lower key value
        IF Left has a left child
                Delete the image of the link to Left's left child
                Move Left's left child subtree aside
                ENDIF
        Determine target position for Right as new left child of Left
        Create a path from Right's position to target
        Apply the path to entire right subtree
        Create an image to link Right as Left's left child
        Move right subtree to target and make link visible
        Make Right return to the normal node color
        IF Left previously had a left child
                Determine target position for child as Right's right child
                Create a path to target
                Apply path to entire subtree
                Create an image of link to Right's right child
                Move Left's left child and make link visible
                ENDIF
        Make Left return to the normal node color
        ENDIF
ELSE Right has lower key value
        IF Right has a left child
                Delete the image of the link to Right's left child
                Determine target position for Right's left child as Left's right child
                Create a path to target
                Apply path to entire subtree
                Create an image of link to Left's right child
                Move Right's left child and make link visible
                ENDIF
        Determine target position for Left as Right's left child
        Create a counterclockwise path from Left to target
        Apply the path to entire left subtree
        Create an image to link Left as Right's left child
        Move left subtree to target and make link visible
        Make Left return to the normal node color

        Move Right (which is the root of the resulting tree) into the position formerly occupied by Left
        Make Right return to the normal node color
        ENDELSE
END
```

Figure 4: Pseudo-code for the AnimComparisonLink routine.

By using the layout technique we discussed earlier, it is straightforward to determine the target location to which a node should move. Because bounding box locations of XTango images are available, the departure position for a motion path is also readily available. Once these two endpoints of a movement action are determined, the motion is described by creating a path between the two endpoints. Frequently this path is simply linear, and offsets in the path are small enough to insure a smooth transition from the initial point to the target. Nonlinear paths are used in linking a left subtree into a right subtree by following a counterclockwise semi-circle from the node's current location to the target.

Moving an entire subtree is an important visual effect in our animations. To accomplish this, we use the following steps: First, we determine a path of motion for the root node of the subtree. This path, along with an initial dummy transition and a pointer to the root node are passed to a recursive function which returns a new composite XTango transition. In XTango, composing transitions causes the actions embodied in the transitions to occur simultaneously in the animation window. The recursive function creates a transition which applies the path to the node's image and composes this transition with the transition that was passed in as its parameter. The function calls itself recursively with the motion path, composed transition, and pointer to its existing children. The transition returned from the very first call is finally the composition of all the transitions which were created by applying the path of motion to each node in the subtree. When the transition is performed, all visual components of the subtree simultaneously move along the same path causing smooth motion of an entire subtree.

5 Summary

We have described new methodologies for animating tree algorithms, in particular, operations on a pairing heap data structure. These methodologies utilize basic tidy layout techniques together with smooth state transitions to help preserve viewing context and to promote comprehension. We have developed a constant-time heap update layout technique that minimizes modifications to the tree from operation to operation, and we have described how it is implemented.

In an ongoing project we are conducting user testing on students interacting with the pairing heap animations to study the animations' effect on algorithm comprehension. We speculate that these animations will help students learn how pairing heaps function. By comparing student understanding both with and without viewing the animations, we seek to acquire statistical evidence of the animations' utility.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[BB90] Ronald M. Baecker and J. W. Buchanan. A programmer's interface: A visually enhanced and animated programming environment. In *Proceedings of the 23rd*

Hawaii International Conference on System Sciences, pages 531–540, Kailua-Kona, HI, January 1990.

[BK91] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1), Winter 1991.

[Bro88a] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[Bro88b] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.

[Bro91] Marc H. Brown. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9, Kobe Japan, October 1991.

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The Pairing Heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, March 1986.

[FT84] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proceedings of the 25th Annual Symposium on the Foundations of Computer Science*, pages 338–344, West Palm Beach, FL, October 1984.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: A space filling approach to the visualization of hierarchical information structures. In *Proceedings of the IEEE Visualization '91*, pages 284–291, San Diego, CA, October 1991.

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[LD85] Ralph L. London and Robert A. Duisberg. Animating programs using Smalltalk. *Computer*, 18(8):61–71, August 1985.

[Moe90] Sven Moen. Drawing dynamic trees. *IEEE Software*, 7(4):21–28, July 1990.

[RT81] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, March 1981.

[SH90] John Stasko and J. Douglas Hayes. *The XTANGO Algorithm Animation System, User Documentation.* GVU Center, College of Computing, Georgia Tech, Atlanta, GA, December 1990.

[Sta90a] John T. Stasko. The Path-Transition Paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, September 1990.

[Sta90b] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[SV87] John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30(3):234–249, March 1987.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal Algorithm Disc. Meth.*, 6:306–318, April 1985.

[TE88] Roberto Tamassia and Peter Eades. Algorithms for drawing graphs: An annotated bibliography. Technical Report CS 89-09, Brown University, Providence, RI, February 1988.

[Vau80] Jean G. Vaucher. Pretty-printing of trees. *Software–Practice and Experience*, 10:553–561, 1980.

[WI90] John Q. Walker II. A node positioning algorithm for general trees. *Software–Practice and Experience*, 20(7):685–705, July 1990.

[WS79] Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, September 1979.

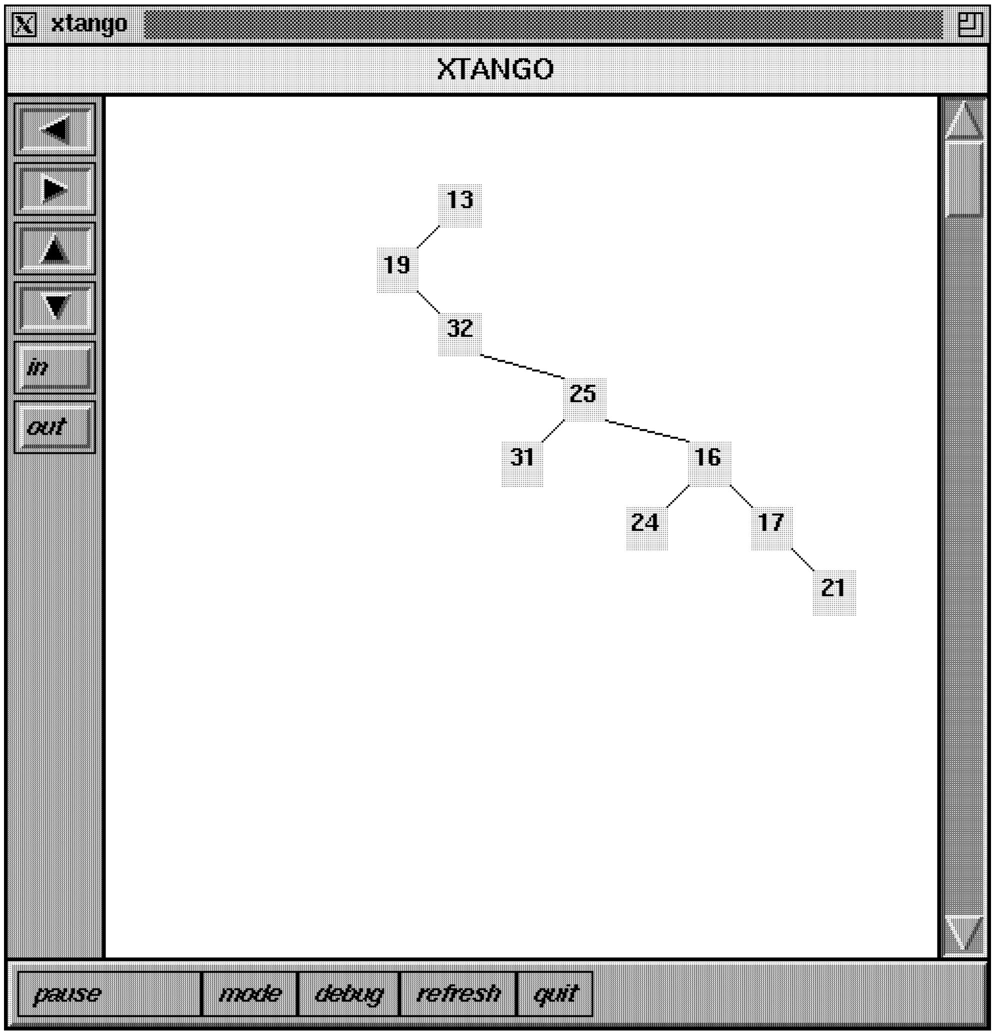

Figure 5: View of the pairing heap prior to a delete min operation. This animation uses the *natural* layout technique.

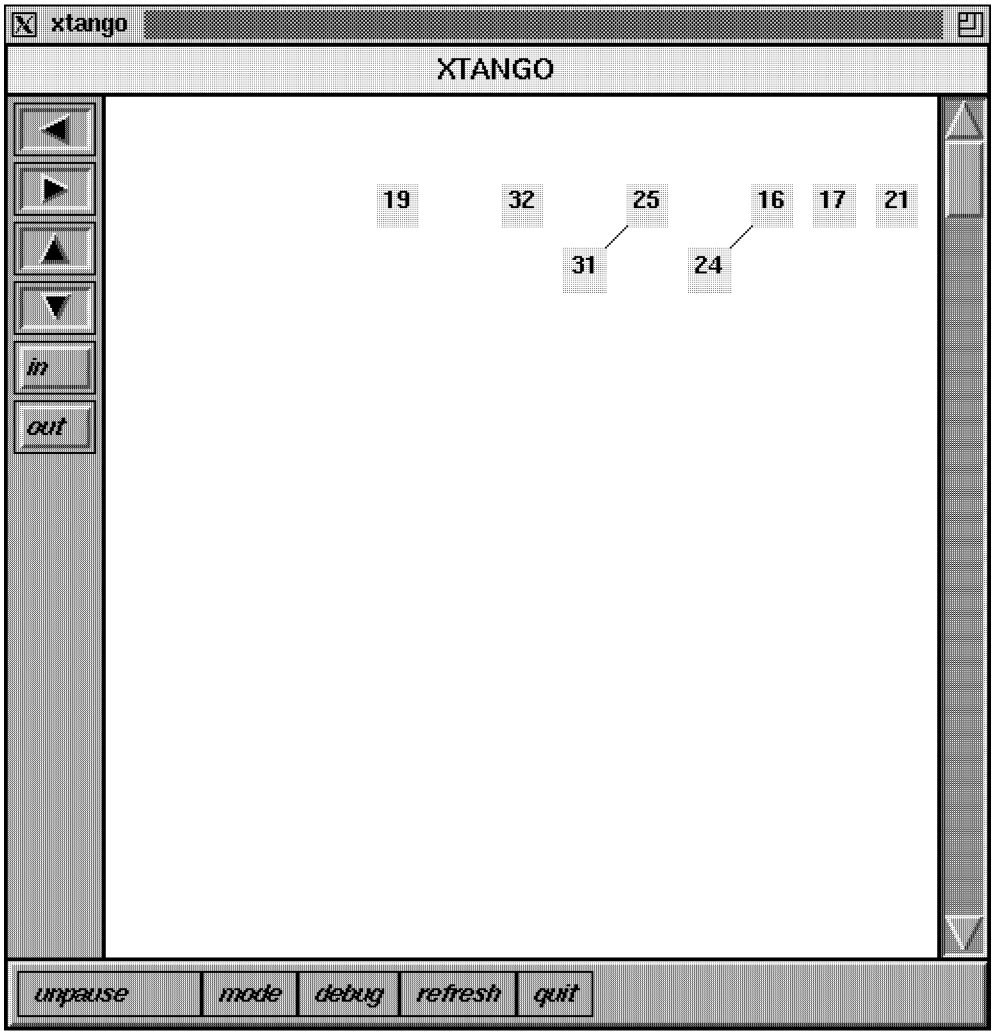

Figure 6: In the delete min, the root of the heap has been removed and all its children have ascended smoothly up as trees in a new forest.

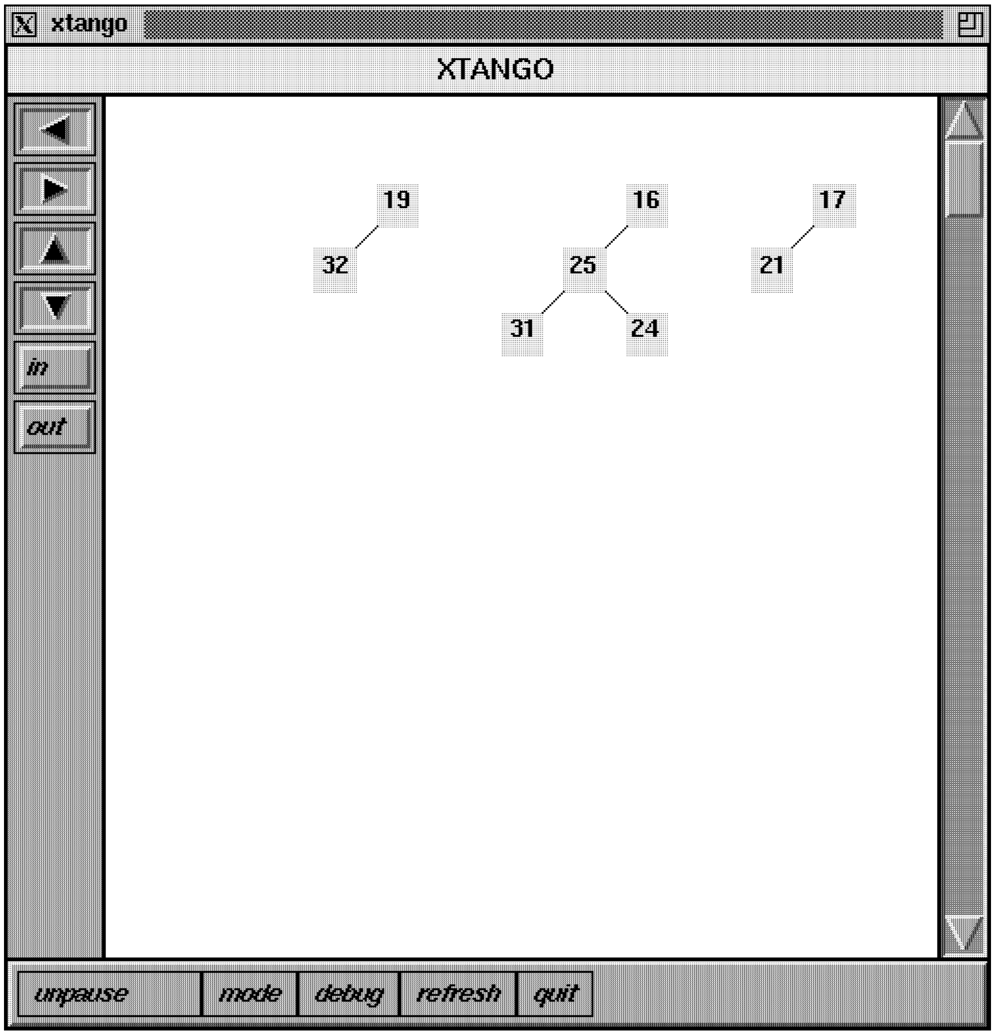

Figure 7: View of the reformation following pass 1 of the two passes. Neighboring pairs of trees have been linked together.

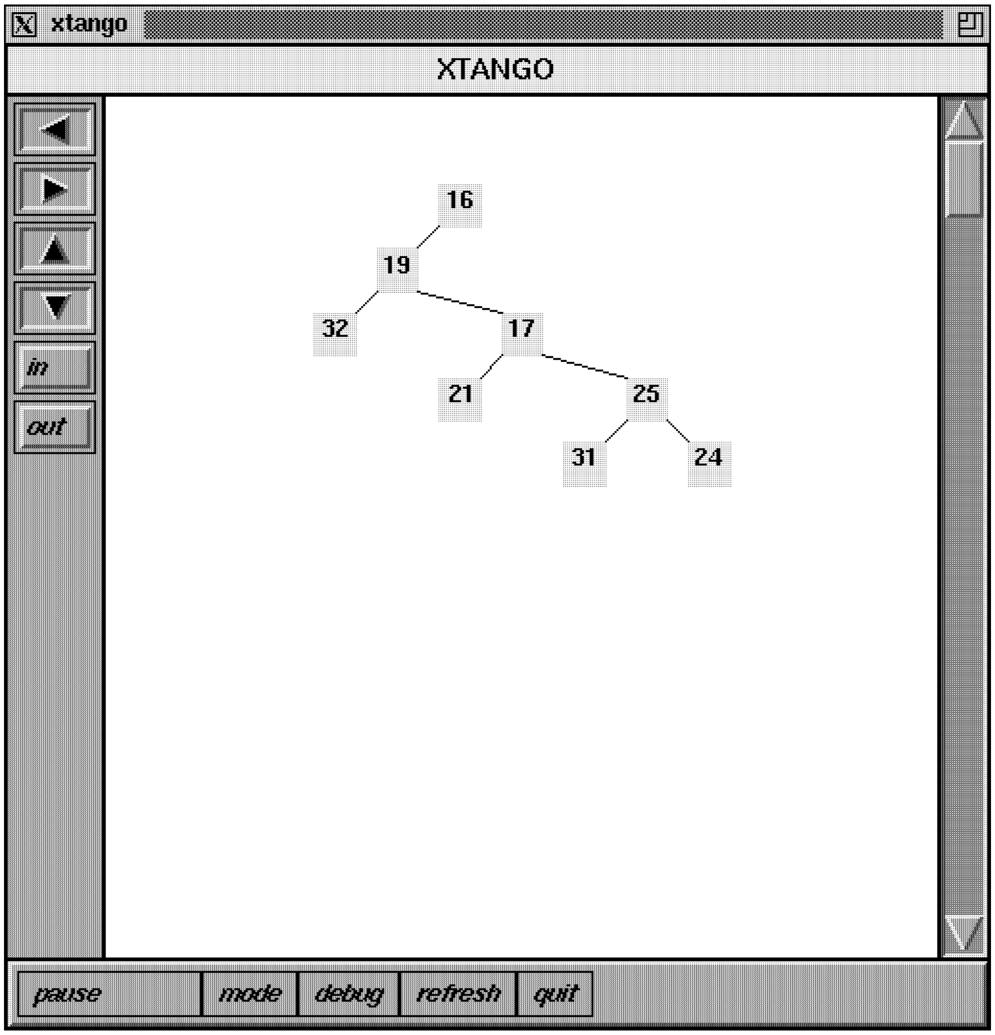

Figure 8: View of the heap following the delete min. Here, the final comprehensive linking pass from right-to-left has just occurred.