VISUALIZATION AND ANIMATION OF SORTING ALGORITHMS

By

JIAN SU

Bachelor of Science

Chinese People's University

Beijing, China

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment
of the requirements of
the Degree of
MASTER OF SCIENCE
December 1999

VISUALIZATION AND ANIMATION OF

SORTING ALGORITHMS

Thesis approved:

Jacques C. LaFrance

Thesis Adviser

M. Samadzadeh

J Chandler

Wayne B. Powell

Dean of the Graduate College

PREFACE

Sorting is one of the most important and challenging topics in the study of data structures and algorithm analysis. The time complexity of most sorting algorithms is in the range between $O(n^2)$ and O(n log n). Commonly used sorting strategies include: 1) sorting by exchanging adjacent elements, 2) sorting by using a binary tree structure, and 3) sorting by using a divide and conquer strategy. The typical examples of above sorting strategies are insertion sort, heapsort, and quicksort, respectively. The speed of a sorting process is heavily depended on the algorithm employed.

For educational purposes, it is important for students to understand the underlying step by step processes of sorting algorithms. Visualization and animation of sorting algorithms may provide a useful aid to achieve this goal. The advantage of visualization is that it provides a direct sensation of a complicated abstract concept.

This study visualized and animated the processes of insertion sort, two variants of heapsort, and quicksort, by displaying every single comparison, movement and exchange of each algorithm in detail. The study also executed and animated the four algorithms in the same time at the same screen for a same set of data to provide a real time comparison of different sorting algorithms. Finally, the study also compared and analyzed the performances of different algorithms for large data sets, based on the running time calculated from the counts of comparisons, swaps and moves that each algorithm took.

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to Dr. Jacques E. LaFrance, Chairman of my advisory committee, for his guidance, assistance, and patience throughout my study at Oklahoma State University. I would also like to thank my committee members, Dr. John Chandler and Dr. Mansur H. Samadzadeh, for their helpful contributions and advice.

A deep-felt thanks goes to my late father, Peizhi, for his love, and to my mother, JiaYuan, and sister, Li, for their unending encouragement and emotional support throughout the years.

Finally, to my husband, Sean Zhang, I wish to express my deepest appreciation for his love, extra patience, and understanding. Completion of this thesis would not have been possible without his unfailing and indispensable support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure page

CHAPTER I

INTRODUCTION

Sorting is one of the most important and challenging topics in the study of data structures and algorithm analysis, not only because sorting is important in practice, but also it provides a very challenging topic for theoretical studies in computer science [Gonner and Baeza-Yates 1991]. As a result, "sorting has received more attention in the computer science literature than any algorithmic task" [Moret and Shapiro 91]. There exist various sorting algorithms, including insertion sort [Knuth 98], shellsort [Shell 59], mergesort [Knuth 81], heapsort [Floyd 64], [Williams 64], [Huang and Langston 88], quicksort [Hoare 62] and a number of other sorting algorithms [Ford and Johnson 59], [Batcher 68], [Weiss 93]. Each algorithm has its own advantages and disadvantages depending on simplicity, time complexity, and its applications. Some algorithms are simple but inefficient. Some algorithms are fast but complicated. Some algorithms have a good average execution time but do not exclude a bad worst case time. Some complicated algorithms perform well for large data sets but are inefficient for small data sets due to higher overhead.

According to algorithm analysis, the time complexity of most sorting algorithms is in the range between $O(n^2)$ and O(n log n). Slower sorting algorithms like insertion sort have an average time complexity of $O(n^2)$. Faster sorting algorithms like heapsort have an average running time of O(n log n). The fastest known sorting algorithm in general

practice, in the average case, is quicksort. The average time complexity of quicksort is also O(n log n) but it is faster than heapsort in practice [Weiss 93]. Insertion sort, heapsort, and quicksort employ three typical sorting strategies. They are: 1) sorting by exchanging adjacent elements, 2) sorting by using a binary tree structure, and 3) sorting by using a divide and conquer strategy.

It should be mentioned that some special sorting algorithms, such as radix sort, bucket sort and counting sort, have time complexity of O(n). But these algorithms usually can only apply to data with small integers or to values in a narrow range, and they usually require extra memory spaces. They are not feasible for sorting arrays of arbitrary integers in general practices. These special sorting algorithms are not included in this study.

Any algorithm that sorts by exchanging adjacent elements requires $O(n^2)$ time on average [Weiss 96]. Insertion sort employs a strategy that involves mostly exchanging adjacent elements. Insertion sort is one of the simplest sorting algorithms. Insertion sort is convenient for small data sets but inefficient for large data sets because it is too slow.

The algorithm of heapsort is more complicated than insertion sort, but it has a better time complexity which is O(n log n) for both worst case and average case. Heapsort takes two steps: the build-heap process which takes O(n) time and the delete-min process which takes O(n log n) time. Therefore, the whole process takes O(n) + O(n log n) = O(n log n) time. Number of variants of heapsort have been developed to improve heapsort performances [Carlsson 87], [McDiarmid and Reed 89], [Xunrang and Yuzhang 90], [Wegener 93]. One way to improve heapsort is to reduce the number of comparisons and swaps in the delete-max phase of heapsort. Floyd proposed a fast heapsort [Floyd 64]. Instead of putting the last leave at the root and sift it down during the delete-max phase,

which takes two comparisons and one swap if needed, Floyd's fast heapsort starts at the empty root node and promotes the larger child, all the way down. This takes only one comparison and at most one move at each level. After this is finished, the last leaf is inserted into the hole and sifted it up, if its key is larger than that of its parent. Because the key of a leaf is usually small and half of elements in a heap are located in leaf level, it is unlikely that the leaf moves up more than one level, and usually it does not move up at all. This algorithm can be further improved if we sift up the hole before inserting the last leaf. This avoids swaps during the sift up process [Knuth 98].

Quicksort employs the divide and conquer strategy. As its name implies, quicksort is one of the quickest sorting algorithms, which takes only O(n log n) average-case time. Although the worst-case time of quicksort is still possible to be $O(n^2)$, by carefully choosing the pivot, such as using a median-of-three pivoting strategy, the chance of the worst case occurring is almost negligible.

For educational purposes, it is important for students to understand not only the running time and strategy employed by each algorithm, but also its underlying step by step process. Visualization and animation of computing algorithms may provide a useful aid to achieve this goal [McCormick et al. 87]. The advantage of visualization is that it provides a direct sensation of a complicated abstract concept. This is especially important for computer science education because computer science involves many abstract concepts which are not directly visible. In most cases, deep imagination is needed to understand those abstract concepts. The use of dynamic visualization will make the teaching of abstract concepts more efficient.

The objectives of this study are 1) to provide a direct and visible approach for understanding abstract sorting concepts by visualizing and animating various sorting algorithms, and 2) to compare the performances of different sorting algorithms, in term of sorting speed. Although a similar study has been done by Muktavaram [Muktavaram 96], his study is more concentrated on the design of Graphic User Interface (GUI) and its underlying functionality to provide a graphical tool for sorting visualization. This study is more concentrated on the animation and visualization process itself. This study displays and animates every single comparison, move, and exchange of a given sorting algorithm in detail. Furthermore, this study can execute and animate several algorithms at the same time in the same screen using the same set of data. The implementation imposes a arbitrary amount time to each step to slow down the execution, so that a user can compare the speed of different algorithms in real time. This study also compares the performances of various sorting algorithms for relatively large size of data, which will make the results more likely reflect the real situations in practice. These features are not in the contents of Muktavaram's study.

Algorithms selected for this study include quicksort, insertion sort, and two variants of heapsort, because quicksort, heapsort, and insertion sort represent three typical sorting strategies. Although it is well known that insertion sort is slow and has no practical merits for problems with sizable data set, one of the purposes of the study is to demonstrate, by using real examples, that the selection of an algorithm will make a great difference in practices. This is important for computer science education. Insertion sort is chosen just for this purpose. Two variants of heapsort are used for this study because the two variants of heapsort and quicksort are all O(n log n) on the average-case time. It

is also important to demonstrate that algorithms with the same time complexity may perform differently in practice. Thus, this study compares the performances of algorithms with different time complexities, and also the performances of algorithms with the same order of time complexity. It is the intention of the author that this study may provide a useful aid for computer science education.

CHAPTER II

RELEVANT LITERATURE

2.1 Sorting

Sorting is one of the oldest problems in computer science and has been the subject of thousands of publications. Knuth summarized the various sorting algorithms, including insertion sort, and their history [Knuth 81]. Gonnet and Baeza-Yates provided a summary of some more recent developments, as well as a comprehensive reference for sorting [Gonnet and Baeza-Yates 91].

Heapsort was invented by Williams [Williams 64], but the linear-time algorithm for heap construction was developed by Floyd [Floyd 64]. The basic algorithm of quicksort was first introduced by Hoare [Hoare 62]. Sedgewick made important contributions to the quicksort algorithm. He did the detailed analysis and empirical studies. Many important results for quicksort can be found in his Ph.D. dissertation [Sedgewick 77a] and his three other publications [Sedgewick 77b, 1978a, 1978b].

2.2 Visualization

The study of visualization as an aid for computer science education started in the 1960s. In the early days, visualizations were mostly done in the form of video tapes and films. Those visualization are static and lack the user-machine interaction. Users could not actively participate in the simulation process during running time. Examples of early

static visualizations include the works of Knowlton [Knowlton 66] and Baecher [Baecher 81]. Interactive visualization started to emerge in the mid-1970s, followed by the introduction of multimedia animation software and tools in the 1980s [Bentley and Brian 87], [Brown 88], [London and Duisberg 85], [Myers 83]. Entering 1990s, computer visualization and animation have become one of the most rapidly growing fields in computer science [Blattner and Dannenberg 92], [Jones 96], [Vikas 96] . With the introduction of Microsoft Visual Basic and Sun Microsystem's Java, computer visualization becomes practical and manageable. Besides Visual Basic and Java, which are programming languages with visualization features, some software such as Macromedia Director [Macromedia 96] are also developed specifically for multimedia purposes. Those development tools made advanced computer visualization and multimedia applications possible. Nowadays, computer visualization and multimedia animation is widely used in many areas in business, science, military, and education.

CHAPTER III

IMPLEMENTATION

The animation is implemented using Java programming language. Java is a multipurpose programming language. Like other powerful programming language such as C and C++, Java can implement very complicated system programs and advanced data structures. Java can also implement window programs, create GUI (Graphic User Interface) and computer animation. More important, because the machine independent feature of Java language; programs implemented by Java can be executed by internet web browser, and are easy to be accessed remotely, regardless the underlying platform.

This study includes two animation projects. The one that animates heapsort, and the one that executes four algorithms concurrently, and compares the performances of the four different algorithms.

3.1 HeapSort Animation

This implementation is designed primarily for animation purpose. Because heapsort implementation employs binary tree structure, animation of heapsort is more complicated than those of other algorithms, and requires more display space. The heapsort interface (Figure 1, top) consists of three panels, the control panel (bottom) that holds buttons, the output panel (top) that display the counts of comparison, swap, and move (move one node only, different from swap which exchanges two nodes), and the center display field where

the actual animation takes place. This same interface handles both variants of heapsort, which are the same in build-heap process but different in the delete-max process. The regular delete-max process put the last leaf at the root and sift it down. Each sift-down will take two comparisons and one swap if needed. The fast delete-max process does not put the last leaf at root after delete root. Instead, it promotes the larger child, all the way down. After this process is done, then sift up the hole if the key of the last leaf is larger than the key of the parent node of the hole. The last leaf will not be put at the hole until the final position is located. This takes only one comparison and at most one move at each level, and no swaps are involved.

Before the animation starts, the display field is empty. Upon pressing the Random-Data button, a set of randomly generated data will be displayed on the screen, as an array and also as a binary tree. The random data will be inserted into the tree in the order that they are generated. The binary tree created by this way will be a complete tree - each parent node will have two children except probably the last subtree. The heapsort interface in Figure 1 has 22 randomly generated data. Theoretically, the program can handle any number of data, but the screen in this implementation can only display 31 nodes at maximum. The default size is 15 nodes.

After a set of random data has been generated, pressing the Build-Heap button will start the build-heap process. When the build-heap process is done, the execution ceases and waits for further user instruction. Thus, users can take time to examine the heap built by a given set of data. Then a user can press Delete-Max button to start the regular delete-max process, or press the Fast Del-max button to invoke the fast delete-max process. For the regular delete-max process, once the root is deleted, it swaps

immediately with the last leaf. But for the fast delete-max process, a temporary node is required to store the deleted root before it is placed in the last leaf. The temporary node is also displayed on the screen outside the heap. A user can also run the whole heapsort process, including data-generation, build-heap, and delete-max operations, by pressing Run-All button. The default delete-max process invoked by this way is the regular delete-max process which is simpler than the fast delete-max method.

The animation displays each movement of the sorting process. When two nodes are in comparison, they are highlighted by a different color. If a swap takes place, one can see how two nodes move towards to opposite directions to replace each other. Once a max-key is deleted, the color of that node changes so it can be distinguished to other nodes. Because a sorting process is mainly data comparisons, swaps, and moves, the counts of comparisons, swaps, and moves will be updated for each corresponding step.

Although heapsort takes the structure of binary tree, it is actually an array. It just takes the abstract concept of binary tree for implementation. This is one reason that visualization and animation may help understanding abstract concepts of data structures.

3.2 Animation of four algorithms together

This program can execute all four algorithms concurrently. The bottom graph in Figure 1 is the interface of the animation process. Currently, each algorithm has an identical set of random data. Animation of the four algorithms will take place in the same time on the same screen. By this way, users can directly observe the performances of different algorithms, because several sorting algorithms may start in the same time but finish in different time. The interface for this animation consists five panels, the overall

control panel which contains the Run-All and Reset buttons and the text field for entering data set size, the quicksort panel, the insertion panel, the heapsort panel, and fast heapsort panel. The panel for each sorting algorithm in turn consists of two panels, the control panel for this particular sorting algorithm, which holds Start button, labels, and text fields for printing the counts of comparisons, swaps and moves (for fast heapsort only), and the display panel that displays the data array and sorting animation of this algorithm.

When the program is first loaded, it displays 20 data by default. This number can be changed by entering the number of data in the text field named Size, and then pressing Reset button. If the size entered is smaller than or equal to 25 (the maximum number of data that can be animated), a new set of random data will be displayed on the screen. If the size is larger than 25, a warning message "Size > 25, run with no animation?" will be displayed, indicating that the program can still be executed, but with no animation. The counts of comparisons, swaps and moves will still be updated during the execution. Thus, the program can be used to test the performances of algorithms on large set of data.

By pressing the Run-All button, the sorting animation starts for all four algorithms. During the execution, comparison of two nodes will cause the two nodes change color. Once swap takes place, the two nodes being swapped will change to a new color and move towards to opposite directions to replace each other. For fast heapsort, instead of swap, each move involves only one node which moves to a blank hole during its delete-max process. This node is also highlighted when it moves. Again, the counts of comparisons, swaps and moves will be continuously updated, respectively, for each algorithm during the execution. Because this animation will actually demonstrate the speed of different algorithms in real time fashion, a certain amount of time is assigned to

the swap, move and comparison processes. One unit of time is assigned to each comparison and each move, and three units of time are assigned to each swap, because a comparison or a move (reassign a data) takes only one step but a swap takes three steps. An unit of time is an arbitrary amount of time. By imposing a given amount of time to each step of a process to slow down the execution, users can actually see which sorting algorithm finishes first, and which one finishes last. This will give a direct sensation, instead of a abstract concept, of the efficiency of different sorting algorithms.

Each sorting algorithm can also be executed separately by pressing the Start button of the algorithm selected. In this case, only the algorithm selected will be executed, and the other three algorithms will remain stationary. We can also run two or three algorithms at a time, or run all four algorithms at a random fashion by randomly pressing the Start buttons of the four algorithms. The execution can be stopped any time by pressing the Reset button, which will generate a new set of random data, and display them on the screen.

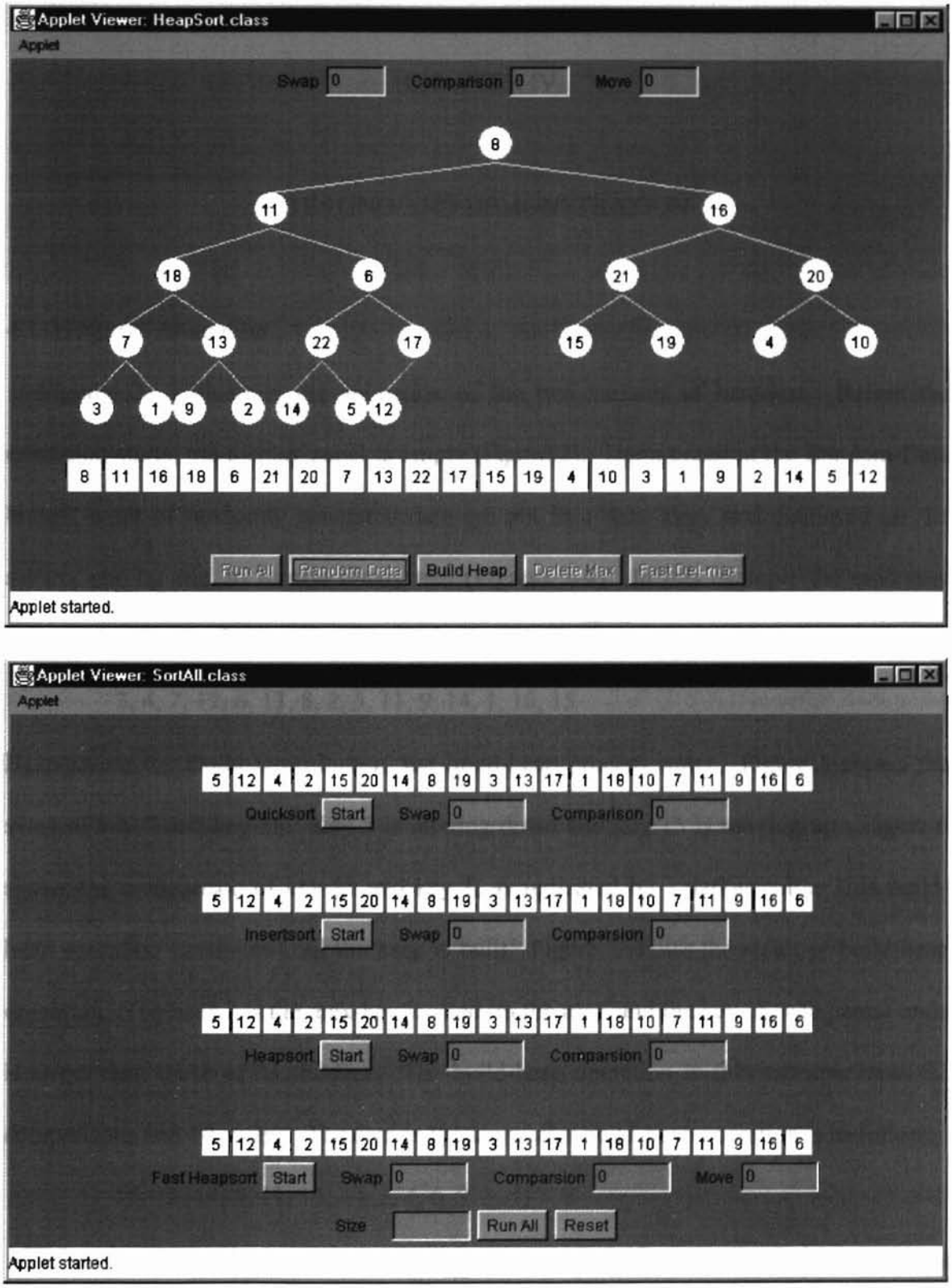

Figure 1. The interfaces for heapsort (top) and for four algorithms combine (bottom).

CHAPTER IV

TESTING AND DEMONSTRATION

4.1 Heapsort Animation

Figures 2-13 illustrate the animation of the two variants of heapsort. Before the execution starts, the display panel is empty (Figure 2). Upon pressing the Random-Data button, a set of randomly generated data are put in a data array and displayed on the screen, one by one, to form a binary tree (Figures 3-4). In this example the randomly generated data is as follow:

3, 4, 7, 12, 6, 13, 8, 2, 5, 11, 9, 14, 1, 10, 15

By pressing the Build-Heap button, the build-heap process starts. Figure 5 shows the swap of key 7 and key 15. Key 7 is moving down and key 15 is moving up. Figure 6 shows the comparison of key 13 and key 1, as indicated by a darker color. This build-heap operation continues until the heap is built. Figure 7 shows the result of build-heap operation. The heap has the max-key located in the root, and the key of each parent node is larger than those of its children. The build-heap operation in this example takes 22 comparisons and 10 swaps. The data in the array after build-heap operation is as follow:

15, 12, 14, 5, 11, 13, 10, 2, 4, 6, 9, 3, 1, 7, 8

When build-heap is done, the execution is suspended. By pressing Del-Max button, the execution resumes and the regular delete-max process starts. Once a node is deleted, it is put in the last cell of the heap and its color changes so it can be distinguished from

other nodes which are still in the heap. The color change appears in both the heap and the data array below the heap (Figure 8). Note that in Figure 8, key 15, 14 and 13 have already been deleted as indicated by a darker color, and key 12 is been deleting. As the root of the heap, key 12 is moving towards to the last cell of the heap. In the meantime, the last node with key 3 is moving towards the root. Even though the deleted keys are showing still in the heap, they are not part of the heap structure. They just occupy the space left by each last leaf which moved up to replace the deleted root. Once a max-key is deleted, heap structure is destroyed and it is rebuild by pre-located down approach. Figure 9 shows the rebuild process after key 12 is deleted. Key 3 is locating down and swapping with key11. Key3 will continue to sift down until its appropriate location is found. The delete-max operation will repeat and repeat again until all data are sorted (Figure 10). Note that all nodes in Figure 10 have changed to a darker color, indicating that the delete-max process is over. The whole heapsort process takes 73 comparisons and 49 swaps. Among them, 51 comparisons and 39 swaps took place in the delete-max operation. Although the counts of comparisons and swaps vary from one data set to another data set, it is obvious that delete-max takes more time than build-heap process, as expected by algorithm analysis (O(n) for build-heap and O(n log n) for delete-max).

Figures 11-13 illustrate the fast delete-max process. Because fast heapsort and regular heapsort are the same in the build-heap phase, there is no need to repeat the build-heap process again. After the heap is build (Figure 7), pressing the Fast Del-max button invoked the fast delete-max process (Figure 11). The root (key 15) is deleted and put in a temporary node outside the heap (the node is in the middle of the screen with a different color). The larger child of the root, key 14, is moving toward the root. Then key 13 will

be promoted, followed by promoting key 3. No swaps are involved in this process - only one node moved each time. The count of move is updated for each move. Figure 12 shows that after the deletion of key 6 and the sift down process, the correct location for the last leaf is found, and it (key 1) is currently moving toward the hole (from location to location 5). The deleted root (key 6) then will be put at the hole left by key 1. After the fast delete-max process is done (Figure 13), one can notice that fast heapsort process took fewer comparisons and swaps than regular heapsort did for the same set of data. Most node movements involved only one node, as indicated by the count of moves . The number of comparisons, swaps and moves for the fast heapsort is 60, 10 and 69, respectively.

4.2 Animation of four algorithms together

Figures 14-19 demonstrate the sorting processes of all four algorithms together. All four algorithms have the same set of randomly generated data (Figure 14). Upon pressing Run-All button, sorting starts for all four algorithms simultaneously. As figure 15 shows, quicksort just finished swap key 13 and 14 while insertion sort is swapping key 20 and 4, heapsort is swapping key 1 and 13, and fast heapsort is comparing key 15 and 3. Note that the color for the comparing nodes is darker than the color for the swapping nodes. Also, there is a hole for the data array of the fast heapsort, indicating it is in the fast delete-max process. The counts of comparisons, swaps and moves are updated for each step accordingly. When the execution of quicksort finished (Figure 16), fast heapsort, heapsort and insertion sort are still in progress. Figure 16 shows that insertion sort is currently swapping two nodes (the tow nodes are crossing each other), heapsort is in

comparison, and fast heapsort is moving a node to the hole left by its parent. Fast heapsort finished the second (Figure 17), followed by heapsort (Figure 18) , and insertion sort finished the last (Figure 19).

The maximum number of data that the screen can display is 25 in this implementation. Figure 20 and 21 show that when the data size is larger than 25, the program will execute without animation. It only updates the counts of comparison, swap and move during the execution. In this example, the data size is 100.

Figure 22 demonstrates that the program can also execute individual algorithm separately, even through it is designed to execute all four algorithms concurrently. In this graph, quicksort is the only sorting process that is in execution. Insertion sort, heapsort and fast heapsort remain stationary. Currently, quicksort is swapping key 16 and 5, and there is no action is going on for other sorting algorithms.

CHAPTER V

EFFICIENCY ANALYSIS

Since the efficiency of a sorting algorithm depends on how fast when it applies to large data sets, the implementation is also tested for data size of 100, 1000 and 10000 to determine the relationship between data size and algorithm efficiency. Table 1 is the summary and statistics of 10 executions for each data size. The virtual time is calculated as (number of swap * 3 + number of comparison + number of move), because each swap takes 3 steps and each comparison and each move takes only 1 step. Comparisons in algorithm efficiency among different algorithms are made on the base of virtual time. Table 1 shows that insertion sort has much more comparisons and swaps than other algorithms at each size level, so is its virtual time, and the differences increase significantly as data size increases. The virtual time that insertion sort takes at each size level is in the order of $O(n^2)$, as expected by algorithm analysis. For size=100, quicksort takes 12% of the time that insertion sort takes. For size=100, quicksort takes only 2% of the time that insertion sort takes. When the size reaches 10000, the percentage decreases to 0.3%. Similar results are also obtained between fast heapsort and insertion sort, and between heapsort and insertion sort. The results indicate that the selection of an algorithm could make a significant difference in practice. Obviously, insertion sort can not be used as a practical algorithm for large data.

The differences in running time between heapsort and quicksort are rather constant regardless data size. For the three levels of data size, the time that quicksort takes is about 74% - 80% of the time that fast heapsort takes, and about 39%-45% of the time that regular heapsort takes. This result is expected because heapsort and quicksort are both O(n log n) on the average. The difference between heapsort and quicksort should only be a constant.

Fast heapsort takes fewer comparisons and swaps than regular heapsort at each size level, but fast heapsort takes a sizable one-node moves. After converting these counts into virtual time, fast heapsort takes about 60%, 53%, and 50% of the time that heapsort takes for data size of 100, 1000, and 10000, respectively. Thus, fast heapsort is about twice as fast as regular heapsort. Theoretically, the differences in time complexity among quicksort, fast heapsort, and heapsort are not significant since they differ only by a small constant ratio.

Table 1. Summary and statistics of ten executions of the four algorithms together. The data size is 100, 1000, and 10000, respectively. Virtual time = number of swaps * 3 + number of comparisons + number of moves. Ratios on virtual time between two algorithms are also calculated to determine algorithm efficiency.

		Average of 10 trials				Ratio of each method to:		
Size	Algorithm	Comparison	Swap	Move	Virtual Time	FastHeap	Heap	Insert.
100	Quicksort	692	180	--	1232	.74	.45	.12
	Fast Heapsort	715	69	739	1661	--	.60	.16
	Heapsort	1027	578	--	2761	--	--	.27
	Insertion Sort	2591	2492	--	10067	--	--	--
1000	Quicksort	10096	2594	--	17878	.76	.41	.02
	Fast Heapsort	10543	762	10676	23505	--	.53	.02
	Heapsort	16856	9088	--	44120	--	--	.04
	Insertion Sort	249011	248012	--	993047	--	--	--
10000	Quicksort	139157	33436	--	239465	.80	.39	.003
	Fast Heapsort	139126	7652	140224	302301	--	.50	.003
	Heapsort	235467	124399	--	608664	--	--	.007
	Insertion Sort	21477094	23586595	--	92236879	--	--	--

CHAPTER VI

CONCLUSION AND FUTURE WORK

Sorting is one of the most important and challenging topics in the study of data structures and algorithm analysis. For educational purposes, it is important for students to understand the underlying step by step processes of sorting algorithms. This study visualized and animated the processes of insertion sort, two variants of heapsort, and quicksort, by displaying every single comparison, movement and exchange of each algorithm.

The study also executed and animated the four algorithms in the same time at the same screen for a same set of data to provide a real time comparison of different sorting algorithms. Because a certain amount of time is assigned to each step of an algorithm to slow down its execution, the animation can demonstrate the differences in sorting speed among different sorting algorithms, even for small data sets.

This study also compared and analyzed the performances of different algorithms for large data sets, based on the counts of comparisons, swaps and moves that each algorithm took. Running time calculated from the test results for each algorithm is very close to its theoretical value.

Future study should add the features that can: 1) suspend the execution of the animation process whenever a user requests, and 2) rollback the animation process for one or more steps so that users can trace the previous steps of an executing process.

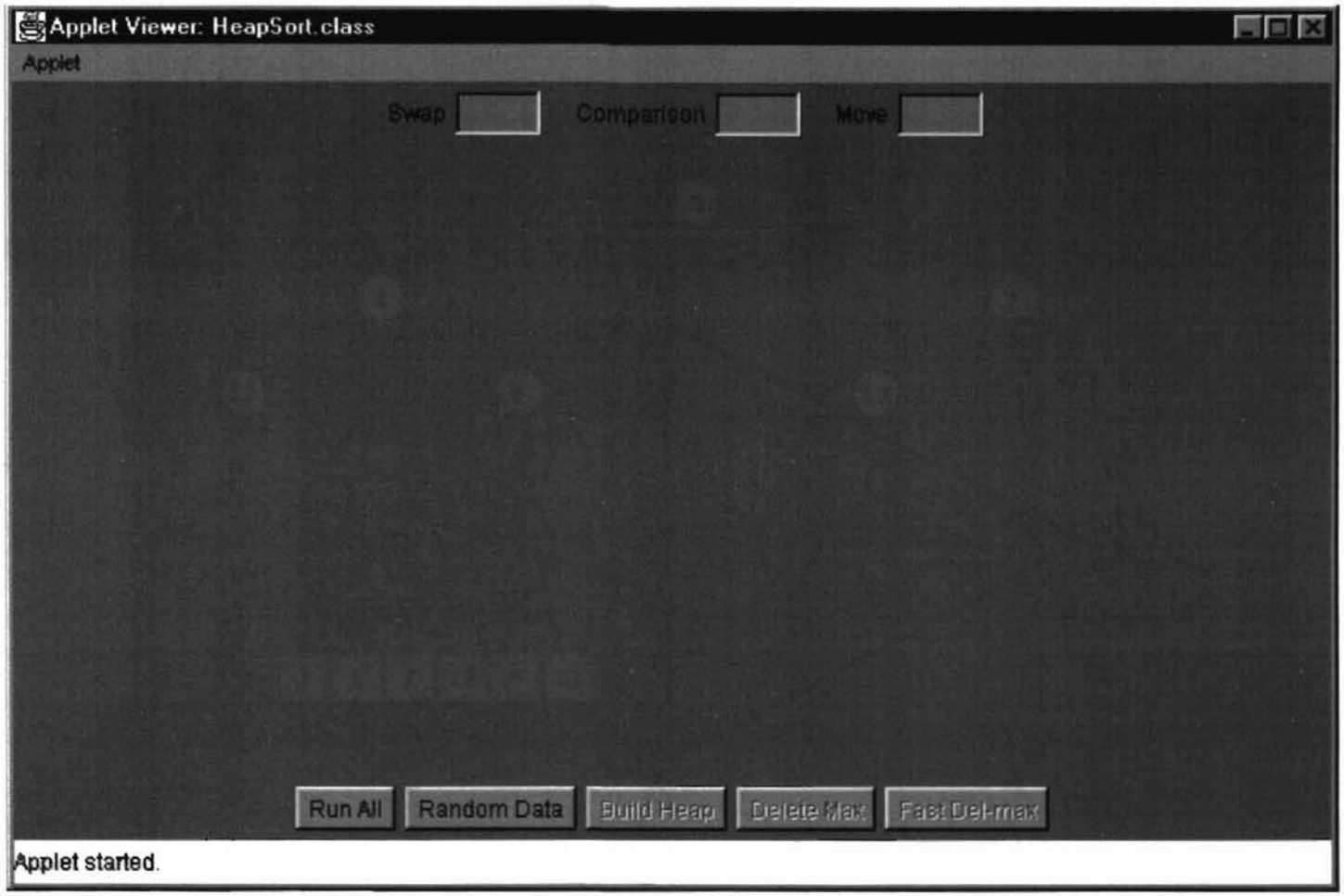

Figure 2. Heapsort interface. The display area is empty until Run All or Random Data button is pressed.

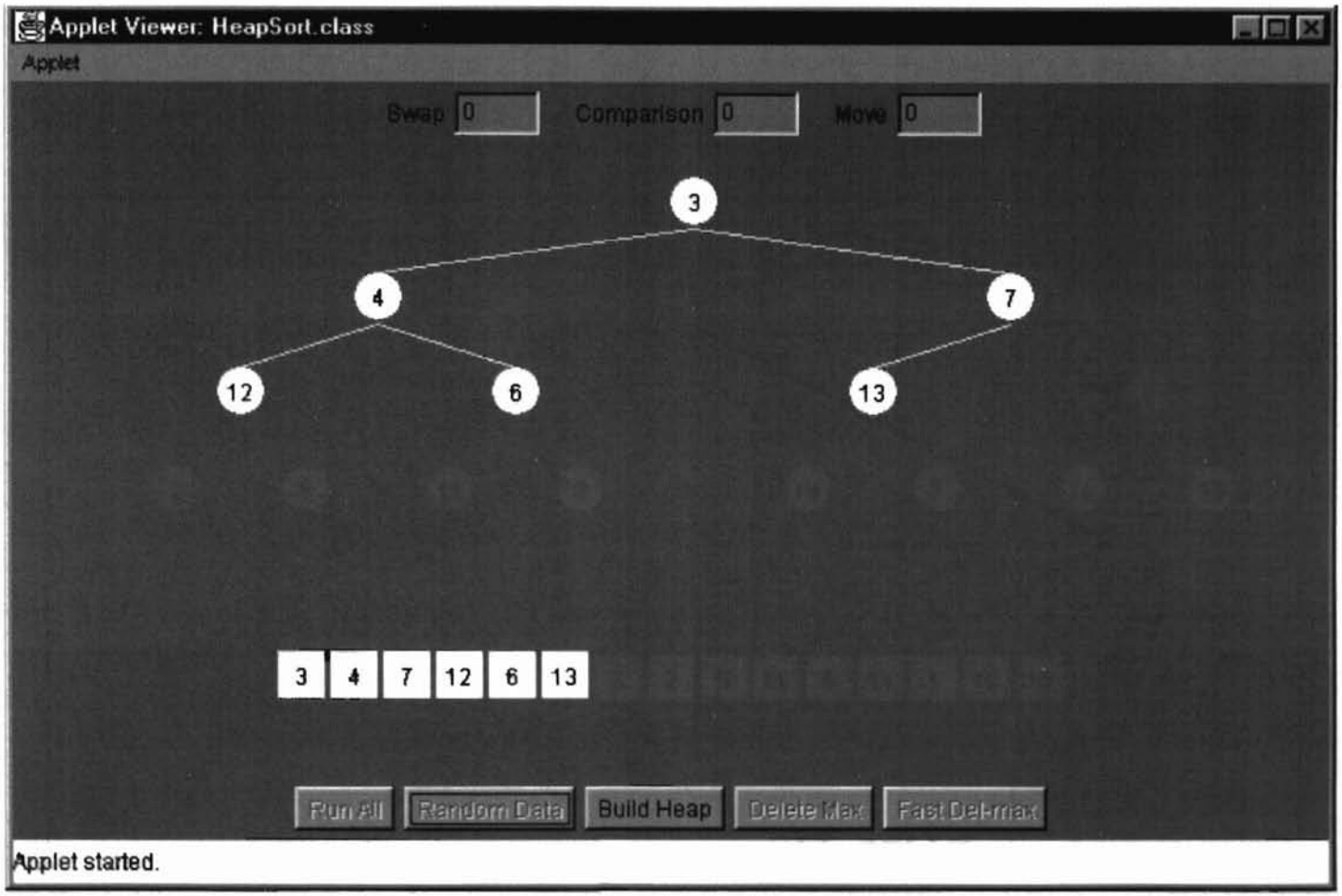

Figure 3. Once Random Data button is pressed, random data are displayed, one by one, on the screen.

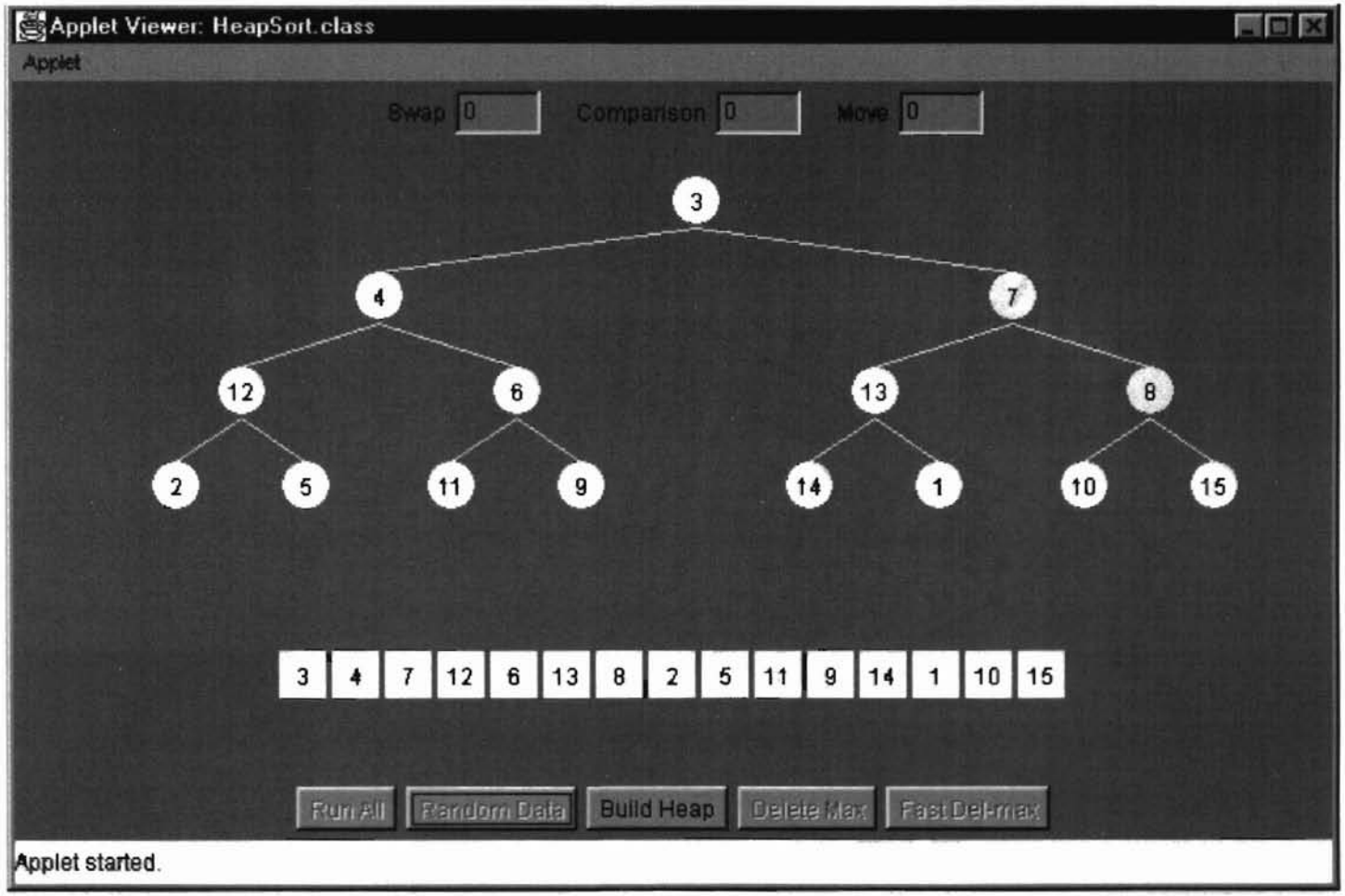

Figure 4. After all 15 random data are generated, the execution of the program is suspended.

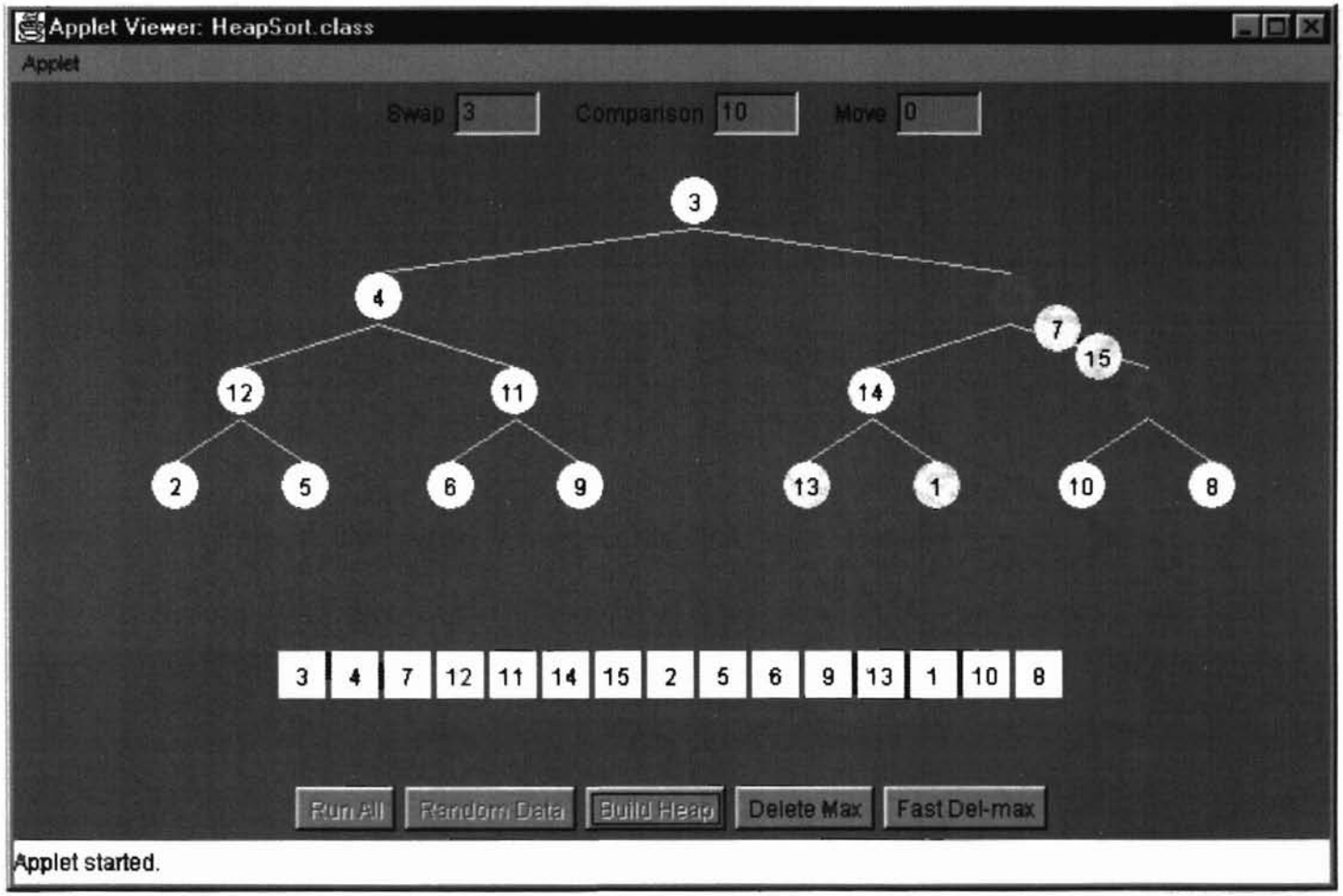

Figure 5. Build-heap process starts once Build Heap button is pressed. Currently, the process is swapping key 7 and 15. The numbers of comparisons and swaps are updated accordingly.

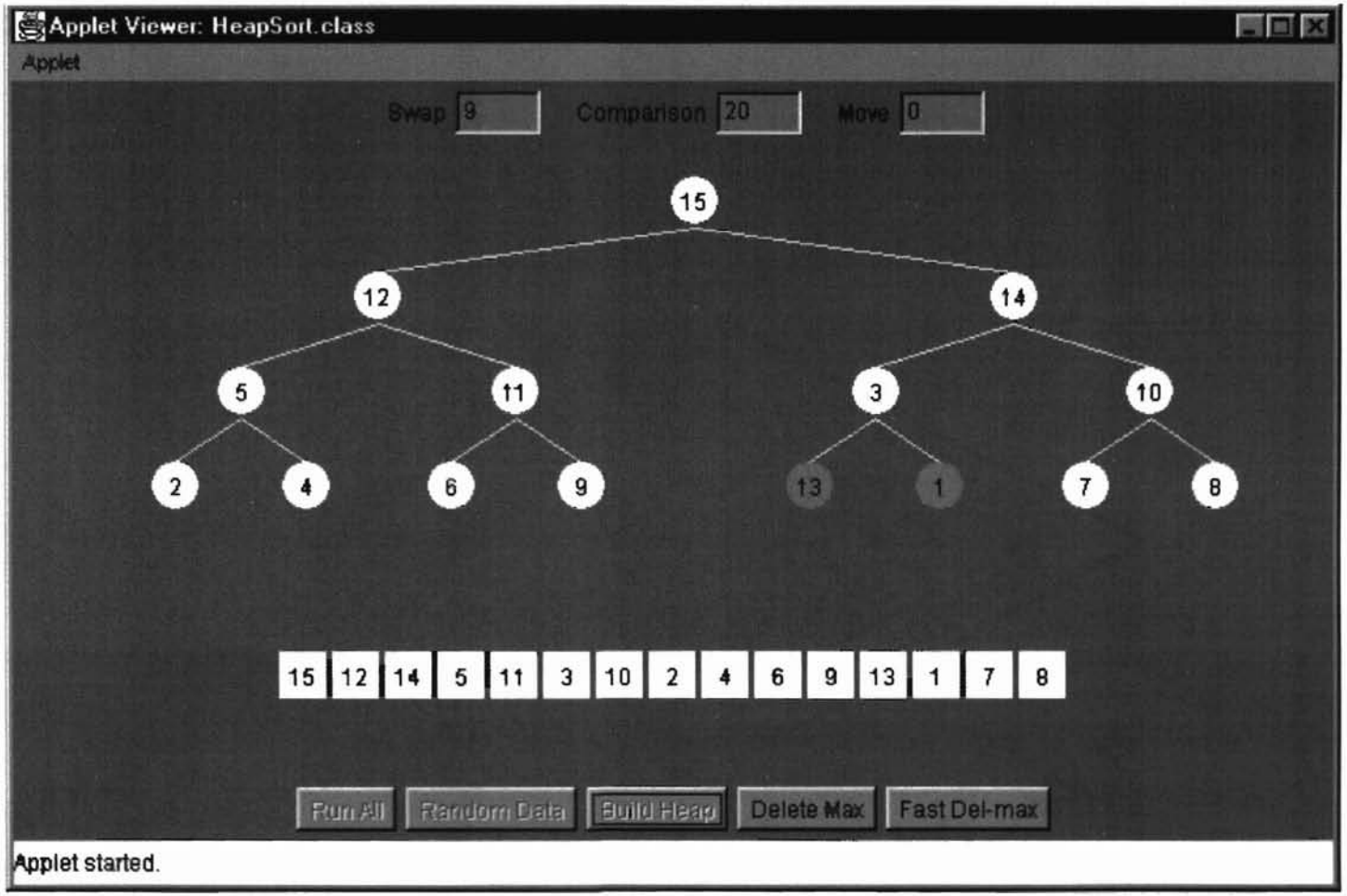

Figure 6. Key 13 and key 1 are currently in comparison, as indicated by a darker color.

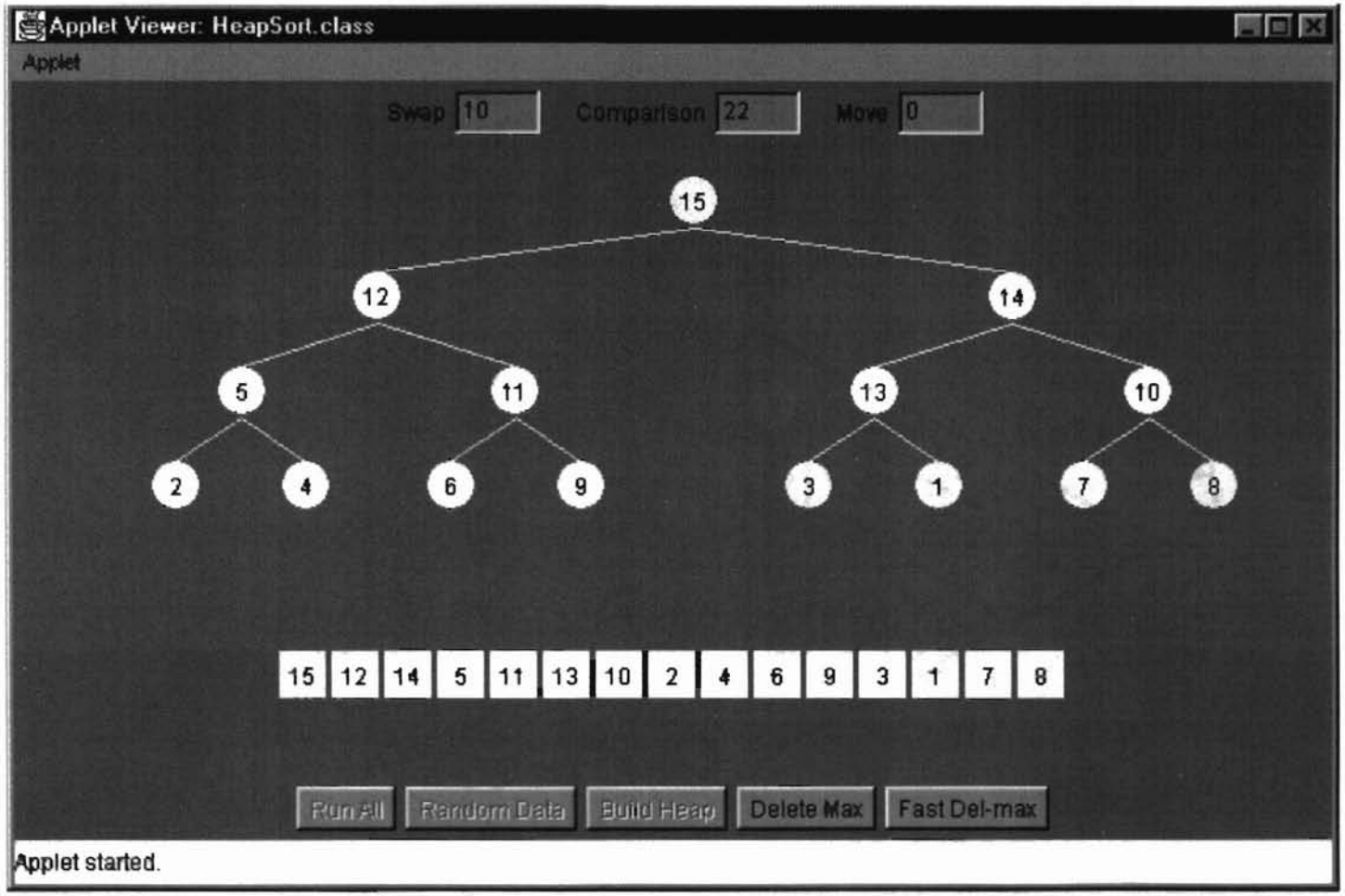

Figure 7. The heap built by build-heap process which made 10 swaps and 22 comparisons. The execution is suspended once the heap is built.

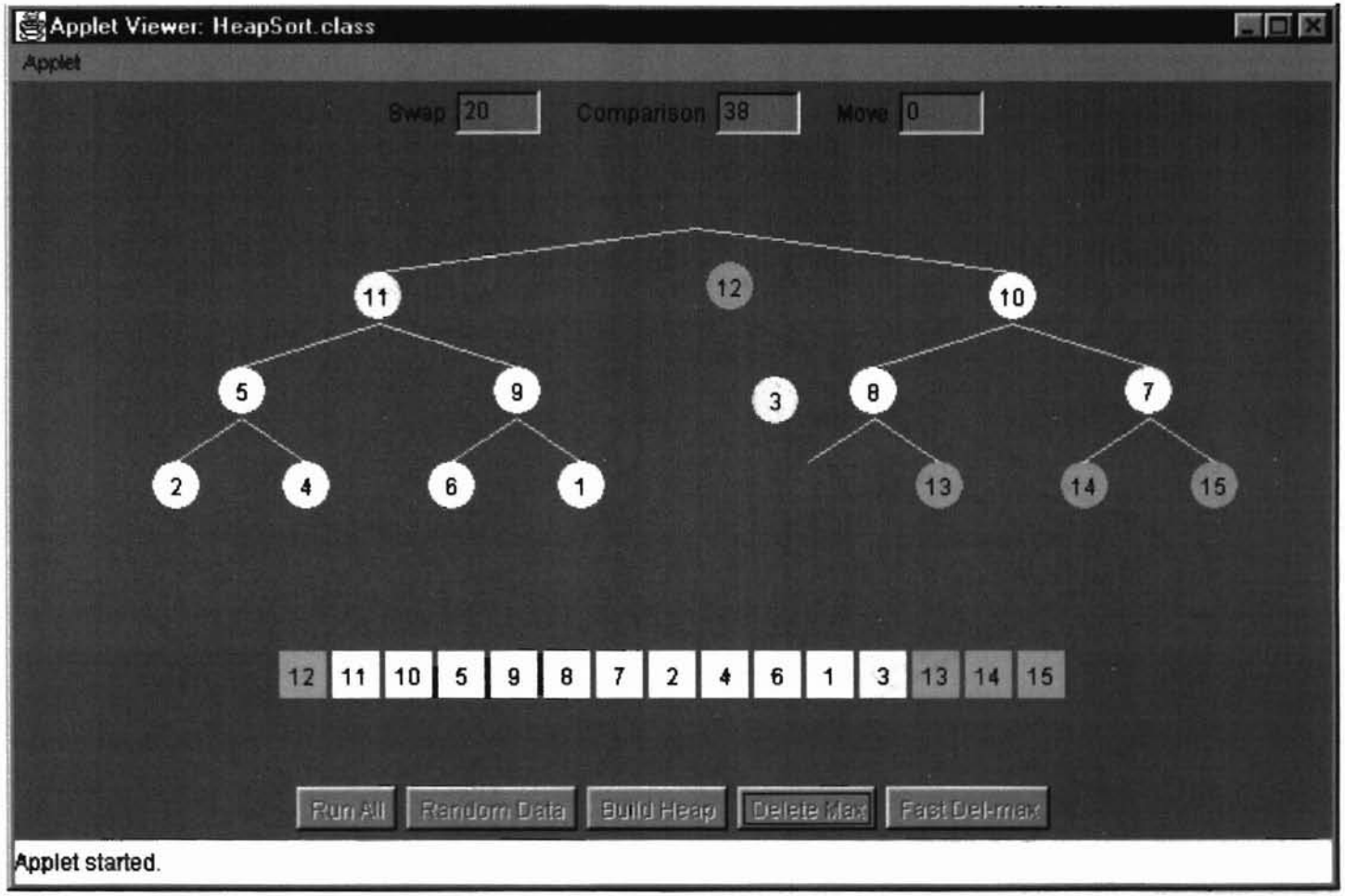

Figure 8. The regular delete-max process invoked by pressing Delete Max button. Currently, key 12 (root) is swapping with key 3 (last leaf). Note that key 15, 14, and 13 have already been deleted, as indicated by a darker color.

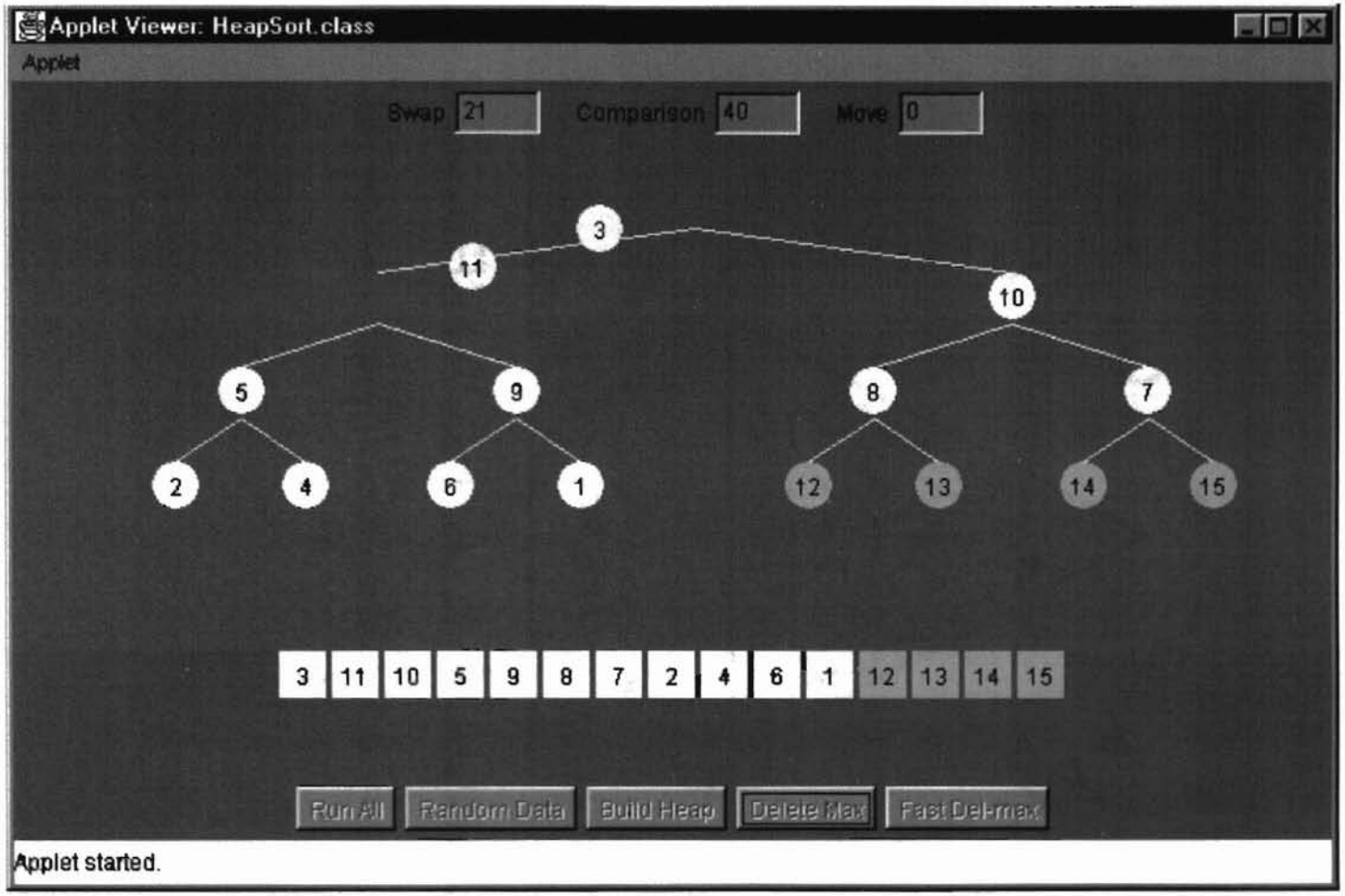

Figure 9. The rebuild-heap process after key 12 is deleted. Currently, key 3 is swapping with key 11.

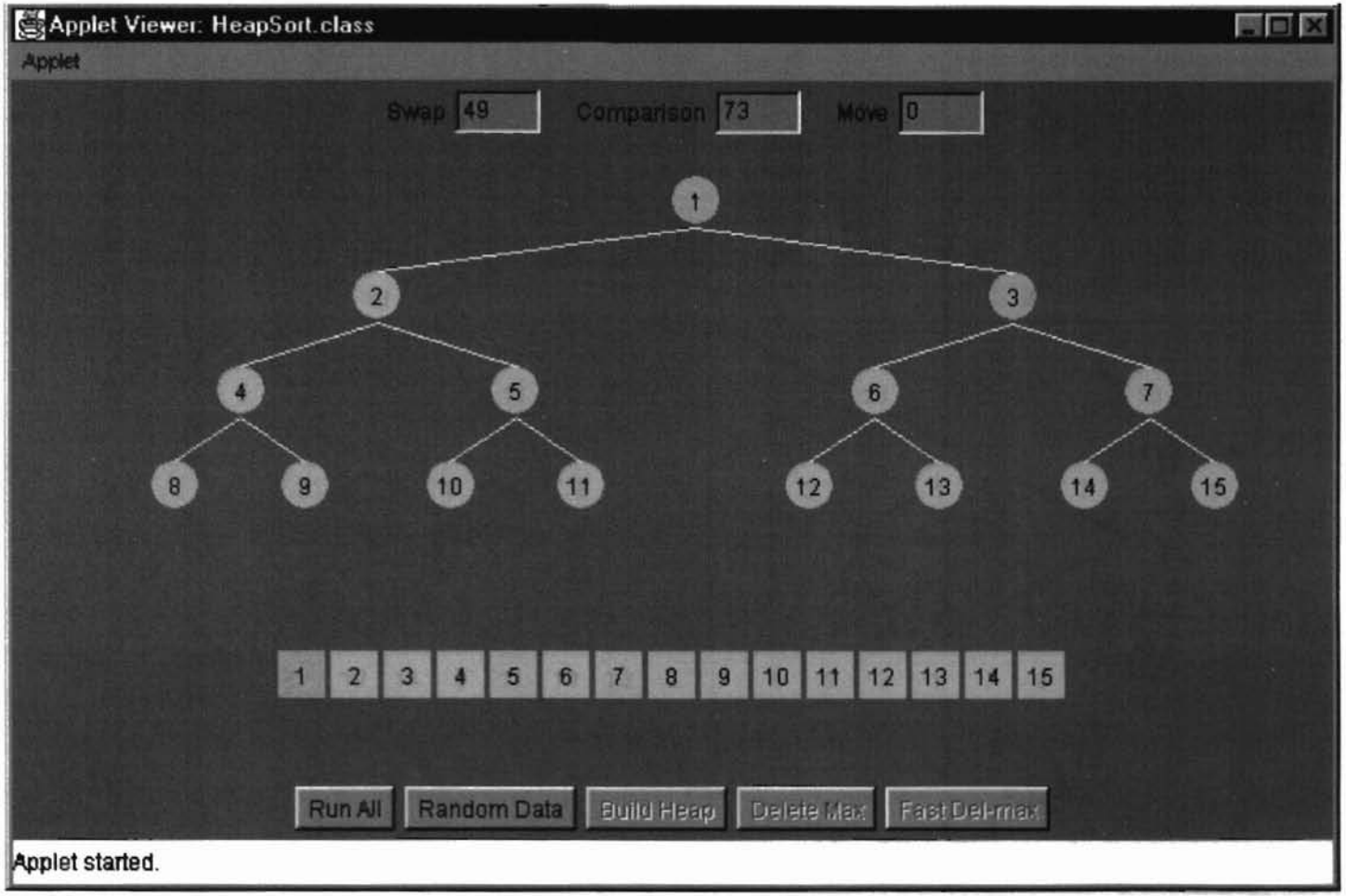

Figure 10. Heapsort completed. This heapsort process made 49 swaps and 73 comparisons

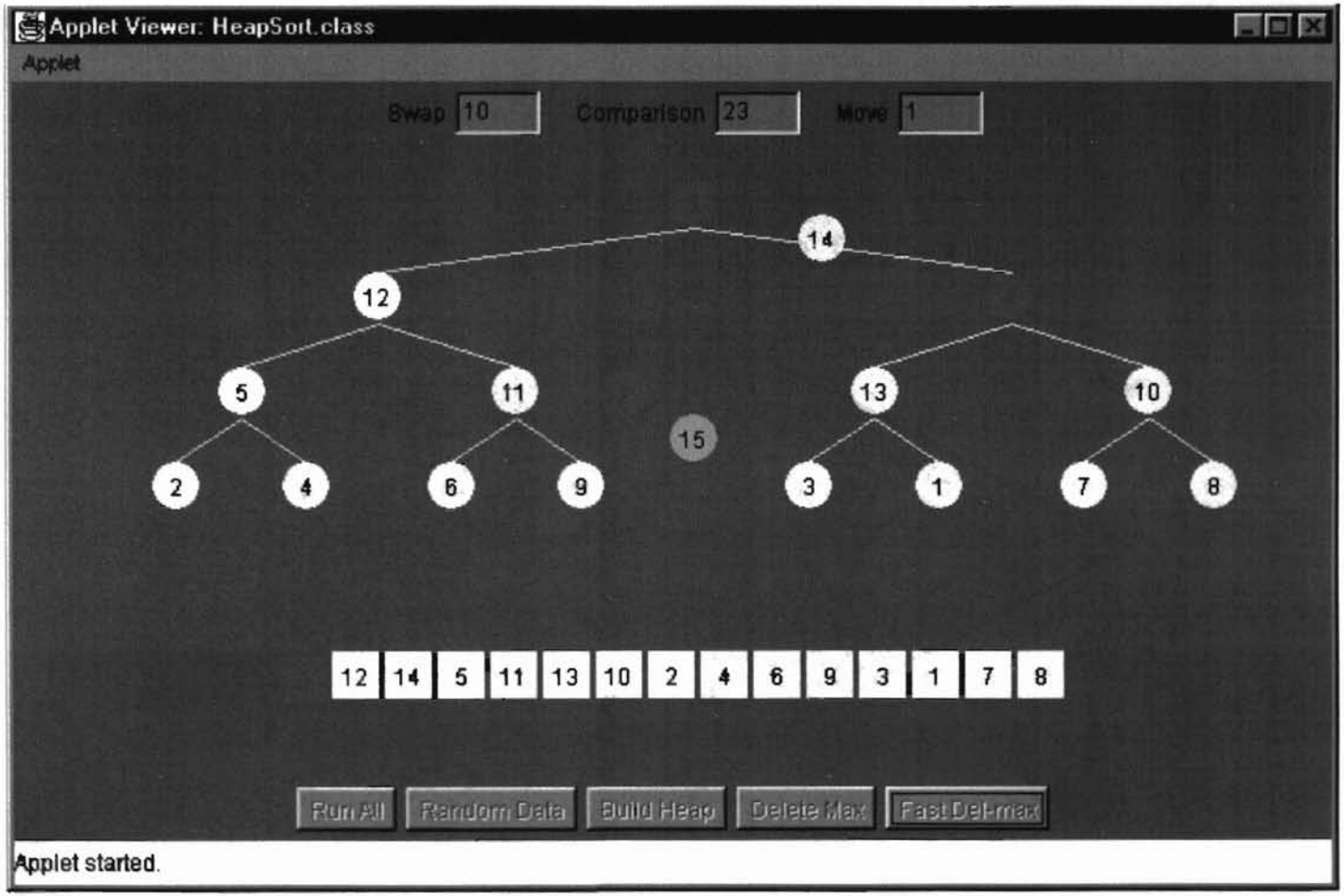

Figure 11. The fast delete-max process which does not put last leaf at root. Instead, it sifts larger child up. In this example, after key 15 is deleted, it is put at a temporary node outside the heap, and key 14 is promoted to root.

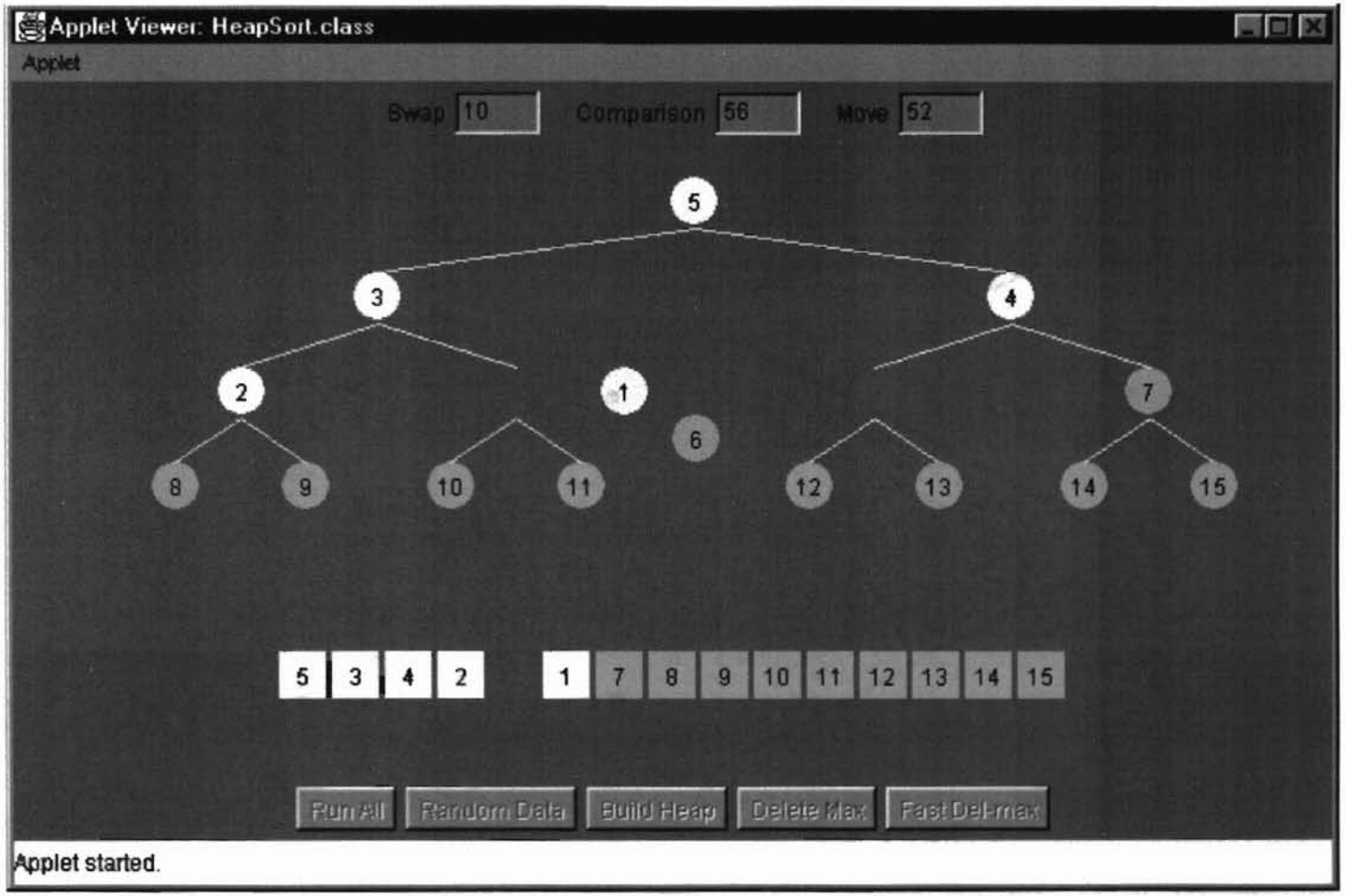

Figure 12. Fast delete-max process continues. After the correct location is found, last leaf (key 1) is moving to the location (the hole below key 3). The deleted key (key 6) will then move to the hole left by the last leaf (the hole below key 4).

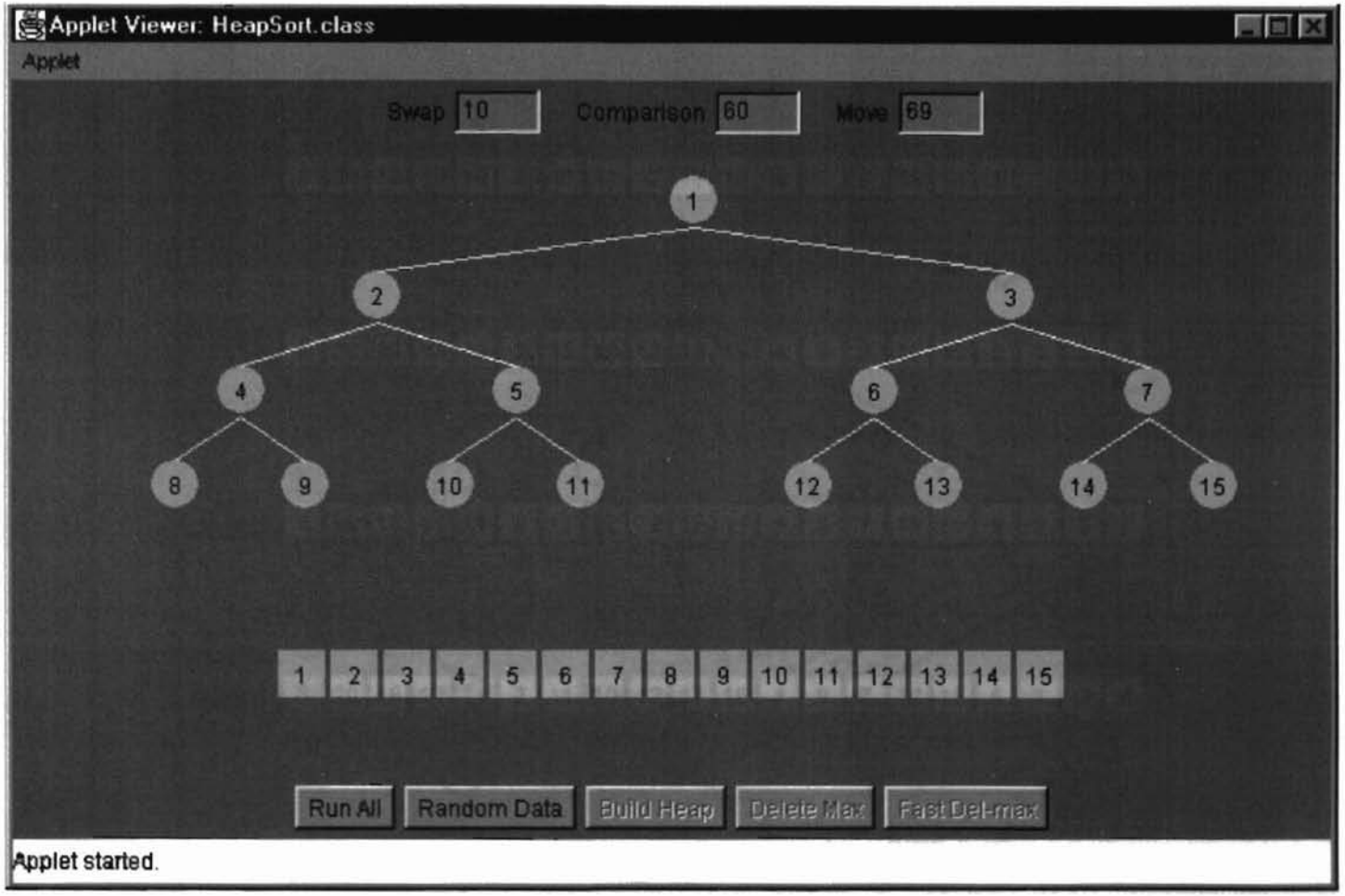

Figure 13. Fast delete-max process completed, and all data are sorted. The fast delete-max made 69 moves. The whole heapsort process also made 10 swaps and 60 comparisons.

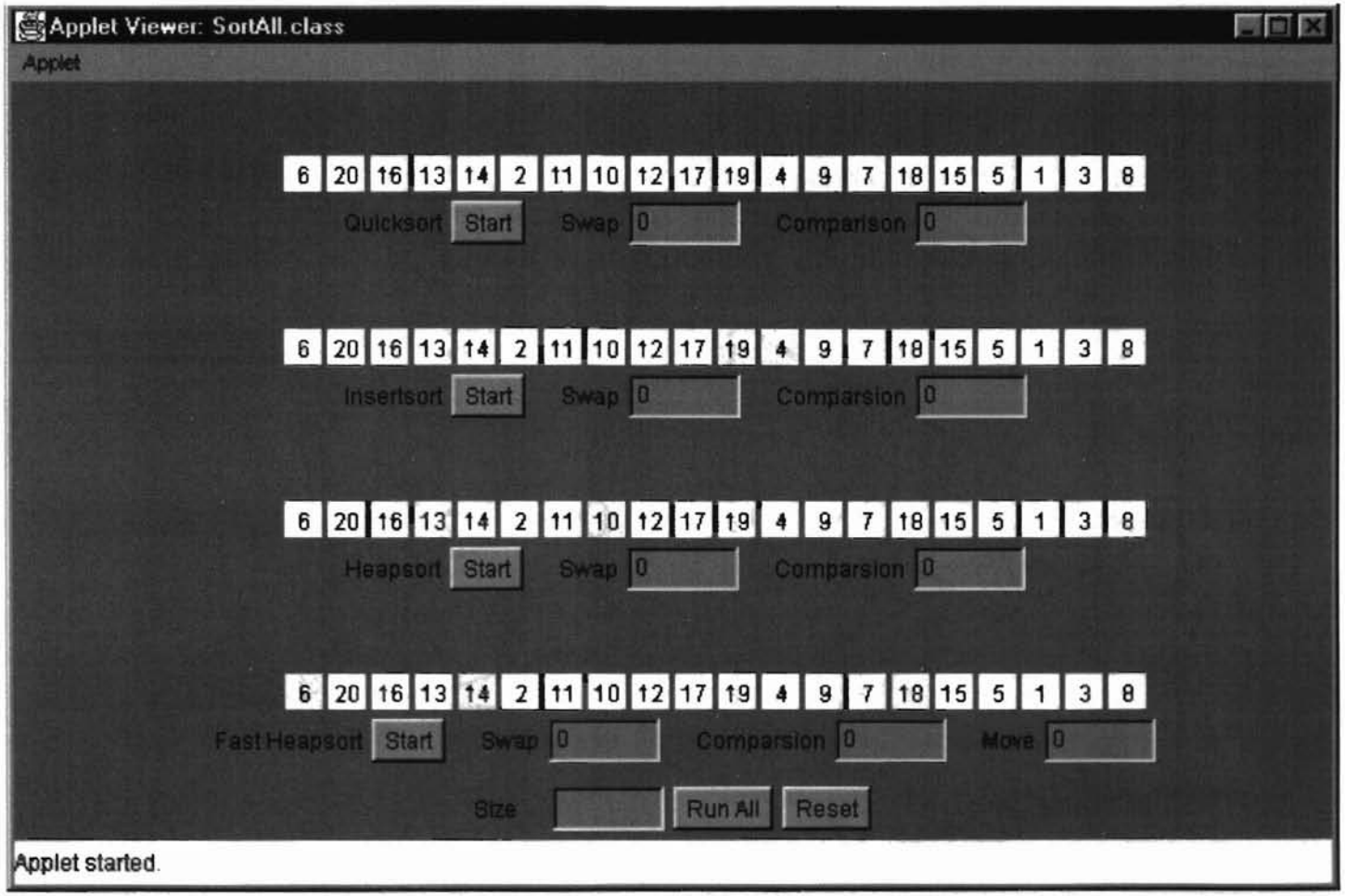

Figure. 14. The interface for the program that executes quicksort, heapsort, fast heapsort, and insertion sort concurrently. The data for each algorithm are identical.

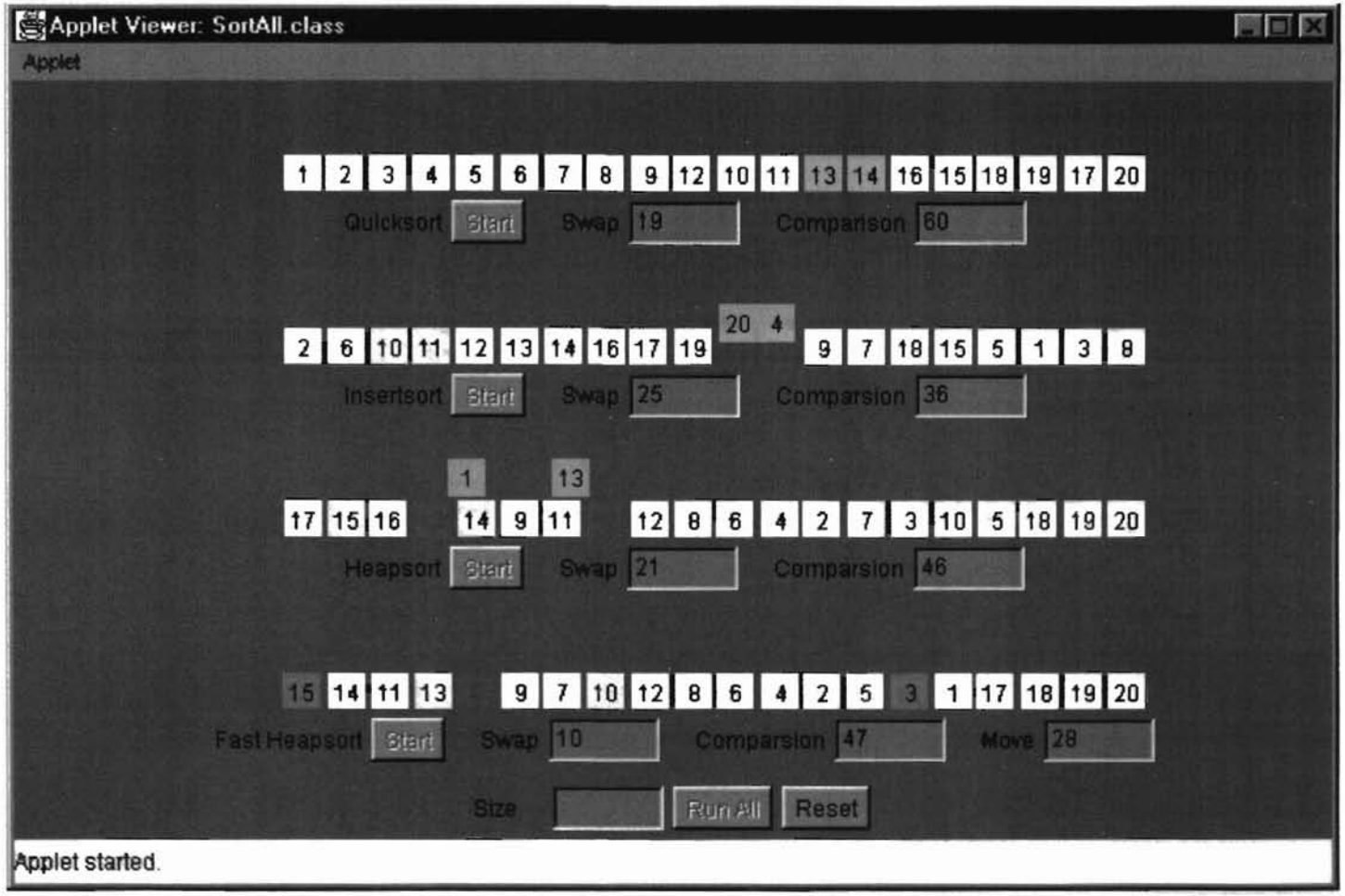

Figure 15. Upon pressing Run All button, the executions of all four algorithms start in the same time. Currently, quicksort just finished a swap, insertion sort and heapsort are swapping two nodes, and fast heapsort is comparing two nodes.

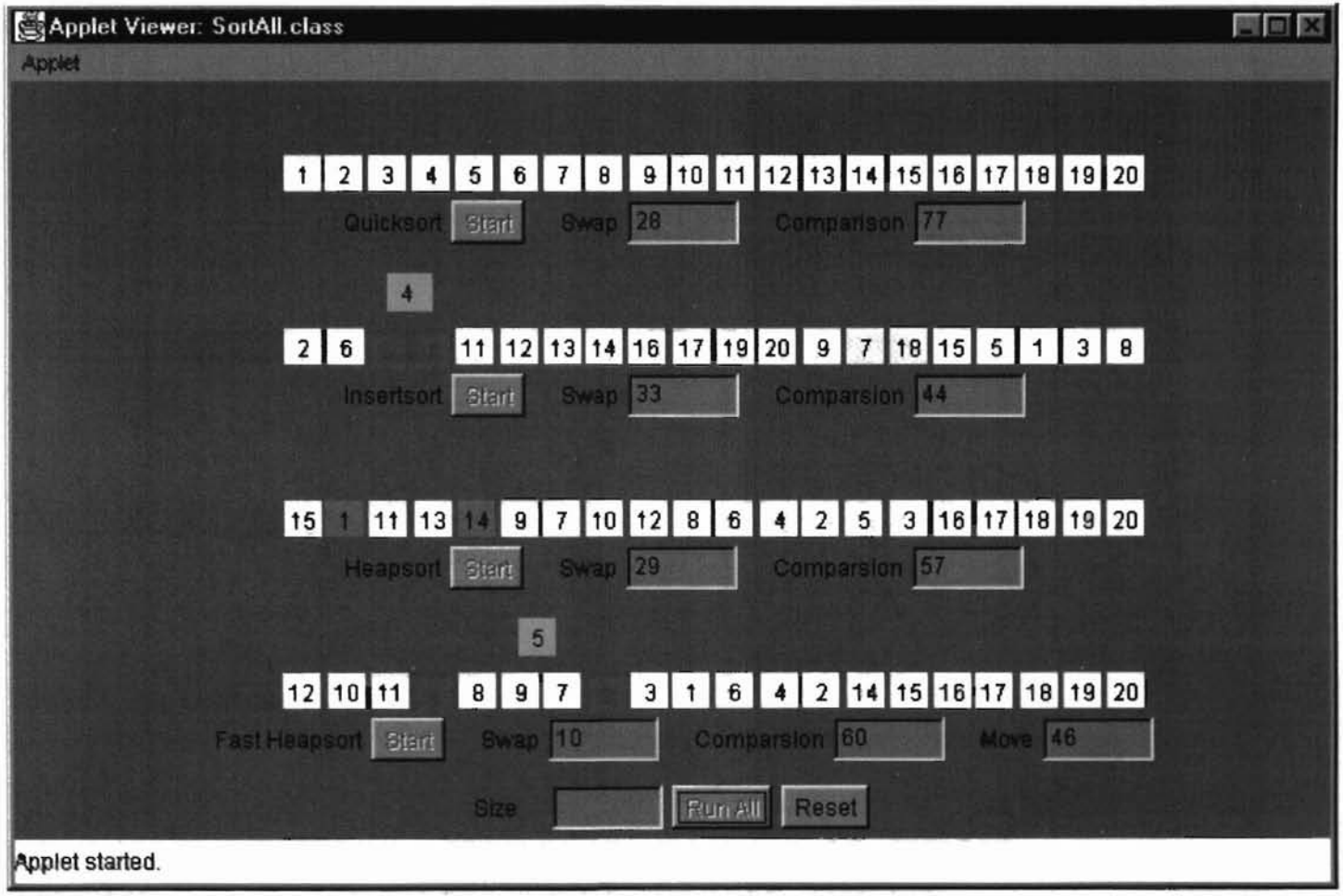

Figure 16. Quicksort completed while other three algorithm are still in progress. Note that insertion sort is swapping two nodes (the two nodes are crossing each other), and fast heapsort has only one node in move.

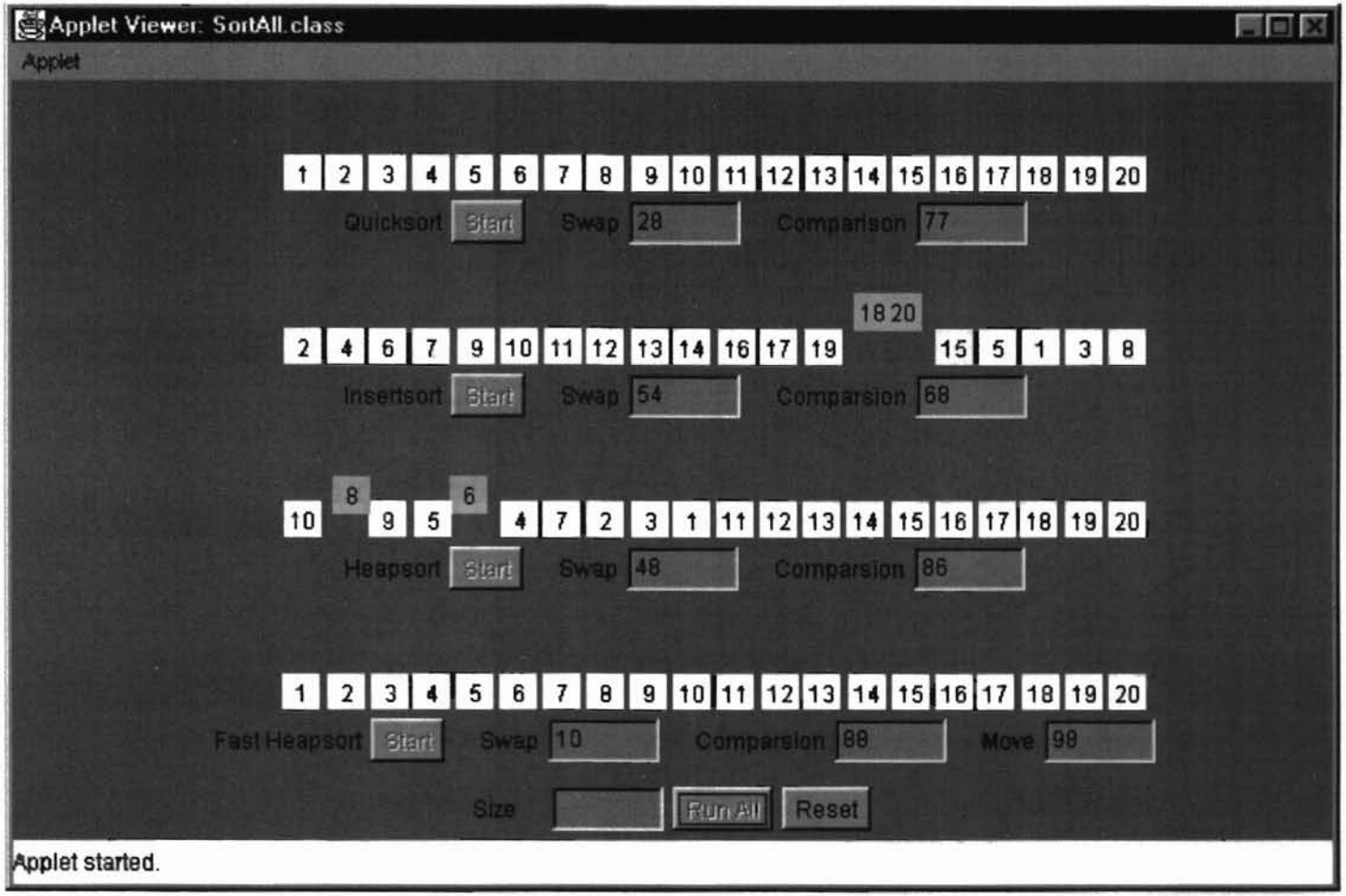

Figure 17. Fast heapsort completed and heapsort and insertion sort are still in progress.

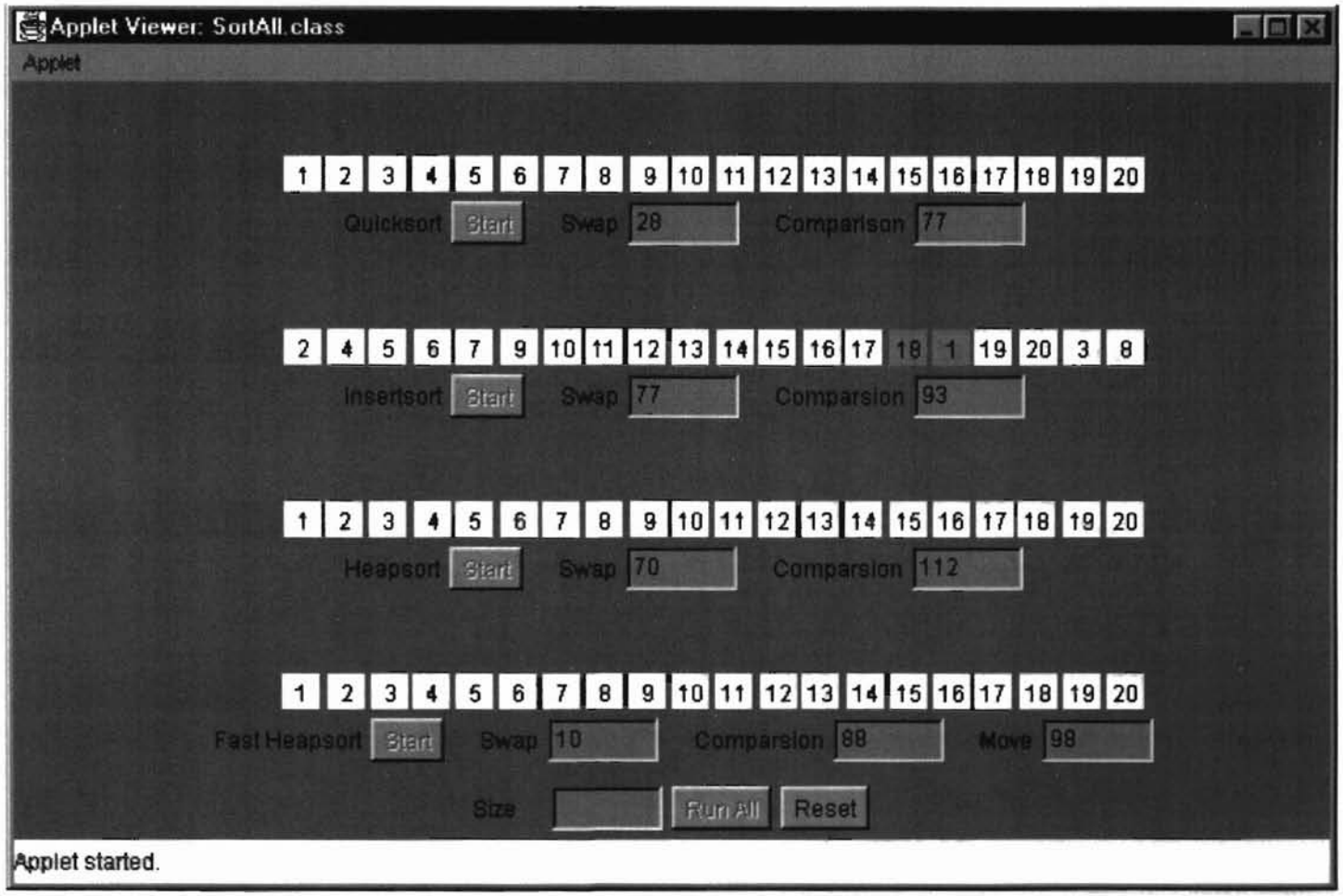

Figure 18. Heapsort completed. Currently only insertion sort is still in progress.

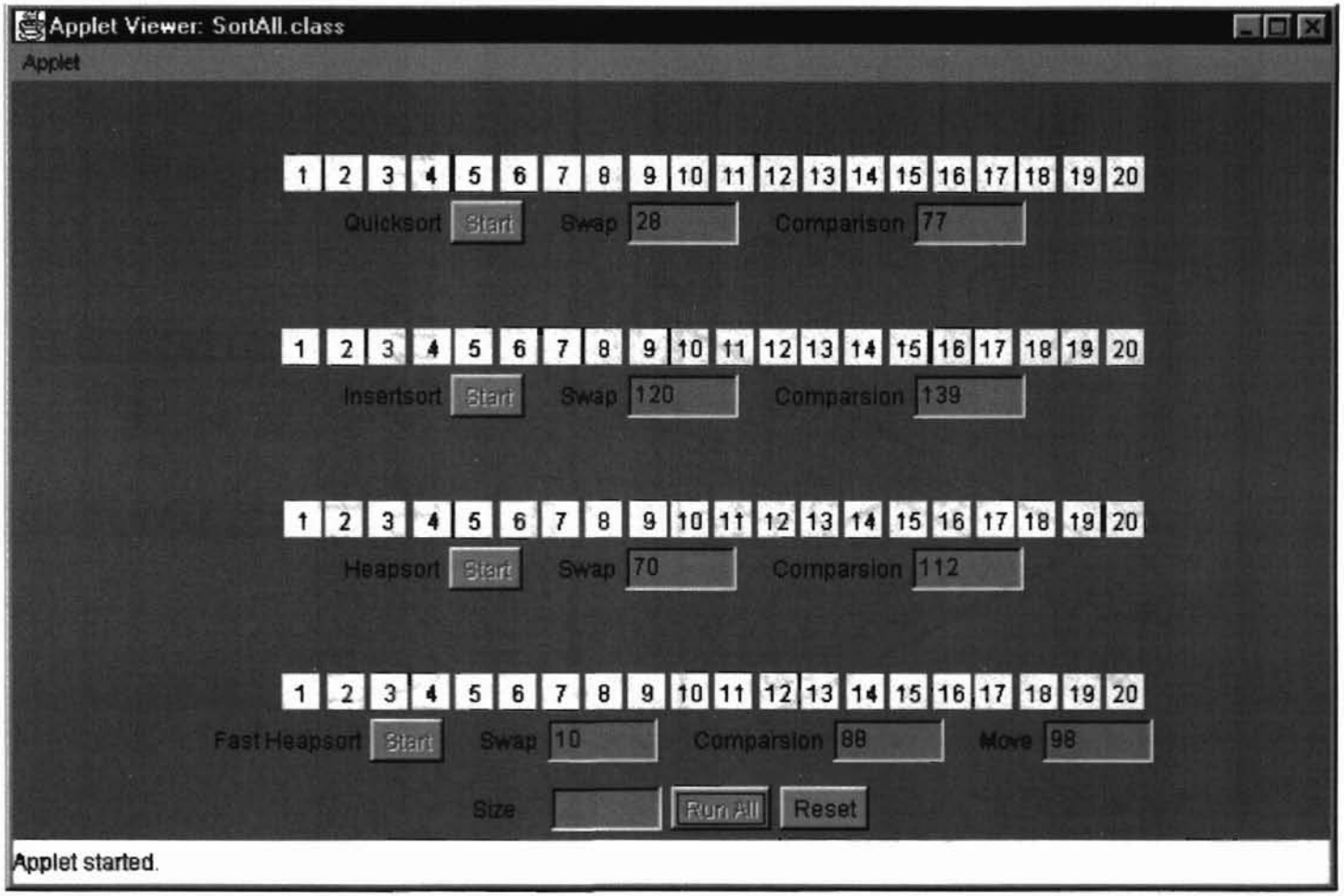

Figure 19. Insertion sort completed, and the data for all four algorithms are sorted.

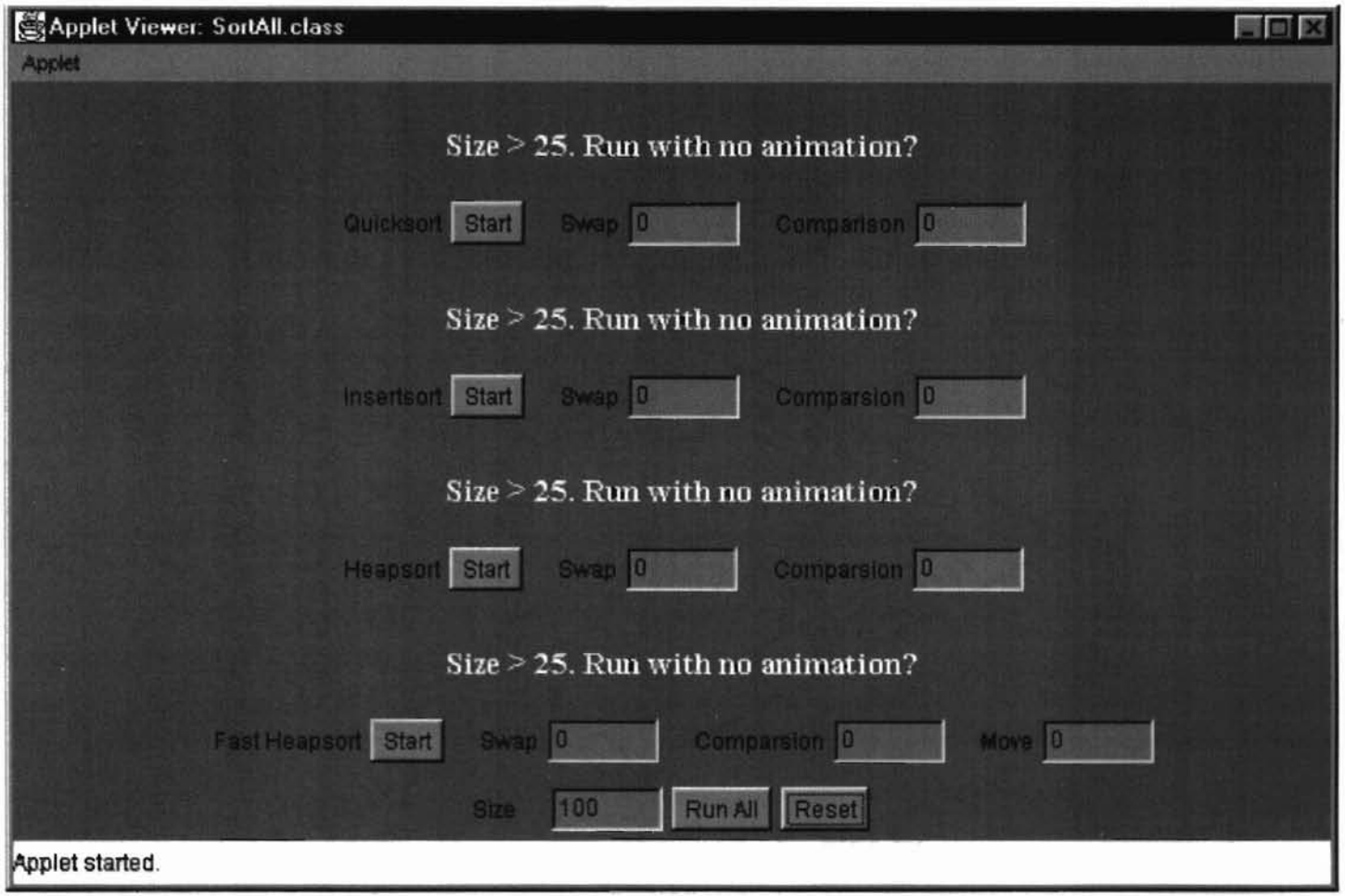

Figure 20. If data size is larger than the default size 25 (currently 100), the program can still be executed, but animation can not be shown.

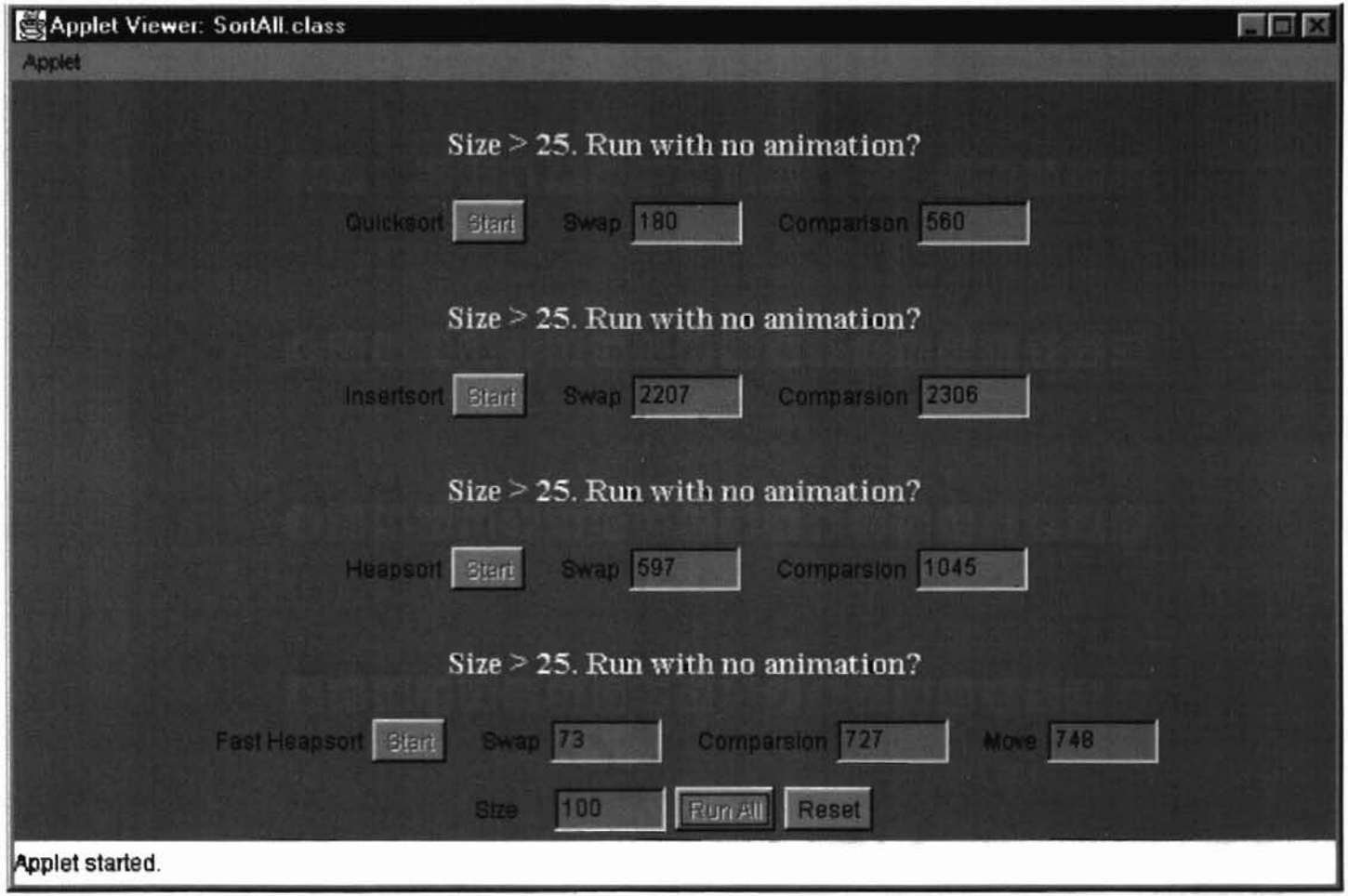

Figure 21. The results of executing 100 data. The counts of comparisons, swaps and moves were updated during execution, but no animation was shown.

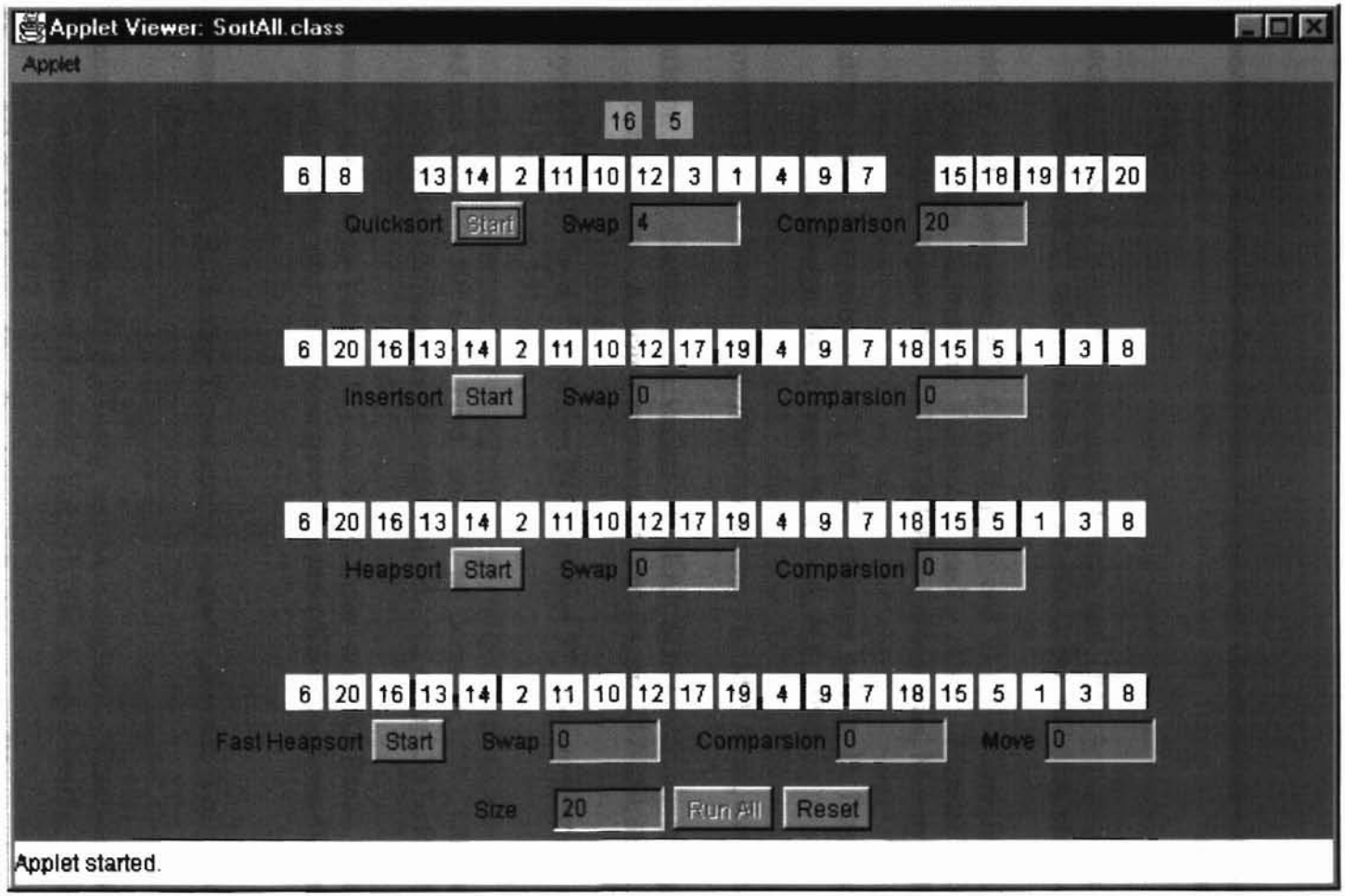

Figure 22. This graph demonstrates that this program can execute individual algorithm separately , by pressing the Start button of each algorithm. Currently, quicksort is the only algorithm in execution.

REFERENCES

[Baecher 81] R.. M. Baecher, *Sorting Out Sorting*, 16mm colour sound film, 25 minutes, Computer Science Department, University of Toronto, Toronto, ON, Canada, 1981.

[Batcher 68] K. E. Batcher, "Sorting Networks and Their Applications." *Proceedings of AFIPS SJCC, AFIPS Press*, Vol. 32, No. 2, pp. 307-314, Montvale, N. J., 1968.

[Bentley and Kernighan 87] J. L. Bentley and B. W. Kernighan , "A System for Algorithm Animation: Tutorial and User Manual", *Computer Science Technical Report*, No. 132, AT&T Bell Laboratories, Murray Hill, NJ, 1987.

[Blattner and Dannenberg 92] M. M. Blattner and R. B. Dannenberg, *Multimedia Interface Design,* ACM Press, New York, NY, 1992.

[Brown 88] M. H. Brown, *Algorithm Animation*, The MIT Press, Cambridge, MA, 1988.

[Carlsson 87] S. Carlsson, "A variant of HEAP SORT with almost optimal number of comparisons", *Inform. Process. Lett.* Vol. 24, pp. 247-250, 1987.

[Floyd 64] R. W. Floyd, "Algorithm 245: Treesort 3", *Communications of the ACM,* Vol. 7, No.12, p. 701, 1964.

[Ford and Johnson 59] L. R. Ford and S. M. Johnson, "A Tournament Problem", *American Mathematics Monthly* Vol. 66, No. 6, pp. 387-389, 1959.

[Gonner and Baeza-Yates 91] G. H. Gonner and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1991.

[Hoare 62] C. A. R. Hoare, "Quicksort", *Computer Journal,* Vol. 5, No. 1, pp. 10-15, 1962.

[Huang and Langston 88] B. Huang and M. Langston , "Practical In-place Merging", *Communications of the ACM,* Vol. 31, No. 3, pp. 348-352, 1988.

[Jones 96] C. V. Jones , *Visualization and Optimization*, Kluwer Academic Publishers, Norwell, MA, 1996.

[Knowlton 66] K. C. Knowlton, "L6: Bell Telephone Laboratories Low-Level Linked

Language, Two Black and White Sound Films", Bell Telephone Laboratories, Murray Hill, NJ, 1966.

[Knuth 98] D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*, Second edition, Addison-Wesley, Reading, MA, 1981.

[London and Duisberg 85] R. L. London and R. A. Duisberg, "Animating Programs Using Smalltalk", *IEEE Computer*, Vol. 18, No. 2, pp. 61-71, 1985.

[Macromedia 96] Macromedia Inc., *Macromedia Director Lingo Dictionary*, Version 5, 1996.

[McCormick et al. 87] B. H. McCormidk, T. A. DeFanti, and M. D. Brown, "Visualization in Scientific Computing", *Computer Graphics*, Vol. 21, No. 6, pp. 1-14, 1987.

[McDiarmid and Reed 89] C. J. H. McDiarmid and B. A. Reed, "Building heaps fast", *Journal of Algorithms*, Vol. 10, pp. 352-365, 1989.

[Moret and Shapiro 91] B. M. E. Moret and H. D. Shapiro, *Algorithms from P to NP, Volume I: Design & Efficiency*, Benjamin-Cummings, Redwood City, CA, 1991.

[Muktavaram 96] Vikas Muktavaram, *Visualization of Sorting Algorithms*, Master of Science Thesis, Computer Science Department, Oklahoma State University, Stillwater, Oklahoma, 1996

[Myers 83] B. A. Myers, "INCENSE: A System for Displaying Data Structures", *Computer Graphics*, Vol. 17, No. 3, pp. 115-125, 1983.

[Sedgewick 77a] R. Sedgewick, "Quicksort with Equal Keys", *SIAM Journal on Computing*, Vol. 6, No. 2, pp. 240-267, 1977.

[Sedgewick 77b] R. Sedgewick, "The Analysis of Quicksort Programs", *Acta Informatica*, Vol. 7, No. 4, pp. 327-355, 1977.

[Sedgewick 78a] R. Sedgewick, "Implementing Quicksort Programs", *Communications of the ACM*, Vol. 21, No. 10, pp. 847-857, 1978.

[Sedgewick 78b] R. Sedgewick, *Quicksort*, Garland Publishing, New York, NY. (Originally presented as the author's Ph.D. Dissertation, Stanford University), 1978.

[Shell 59] D. L. Shell, "A High-Speed Sorting Procedure", *Communications of the ACM*, Vol. 2, No.7, pp. 30-32, 1959.

[Wegener 93] I. Wegener, " BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small)", *Theoretical Computer Science*, Vol. 118, pp. 81-98, 1993.

[Weiss 93] M. A. Weiss, *Data Structures and Algorithm Analysis in C*, Benjamin-Cummings, Redwood City, CA, 1993.

[Weiss 96] M. A. Weiss, *Algorithms, Data Structures and Problem Solving with C++*, Reading, MA, 1996.

[Williams 64] J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM*, Vol. 7, No. 6, pp. 347-348, 1964.

[Xunrang and Yuzhang 90] G. Xunrang and Z. Yuzhang, "A New HEAPSORT Algorithm and the Analysis of Its Complexity", *Computer Journal*, Vol. 33, pp. 281-282, 1990.

APPENDIX: SOURCE CODE

1. HeapSort Animation

```
/*************************************************************************
                            Program : HeapSort
                             Author : Jian Su
                               Data : December 1999
                 Programming Language : Java

There are two classes in this program-the Node class and the HeapSort
class. The Node class which consist of x and y coordinates of the node
and a key.  The HeapSort handles the sorting and animation process.
HeapSort class inherits methods from Applet which is a Java build-in
class.  It also uses the ActionListener interface which handles button
events, and Runnable interface which makes multi-thread possible.
Because Runnable is an abstract data type, its Run() function must be
implemented or overridden.

Java build-in functions used in this program:
init() -- The first function executed when the program is loaded into
browser.  It executes only once.
start() -- Execution following init().
run() -- Executes when a thread starts. Invoked by start() of a
threader.
paint() -- Paint screen.  It is normally called by update().
update() -- Update screen.  It is invoked by repaint().
repaint() -- It invokes update().
stop() -- It is invoked when the program exits the browser.

All other functions are user defined.
*************************************************************************/

import java.awt.*;
import java.util.*;
import java.applet.Applet;
import java.awt.event.*;

//************************************************************************
// Node class: a node contains, a key (data), its x and y coordinates
//             on the screen, and del-max and comparison flags.
//************************************************************************
class Node {
  float x;
  float y;
  int key;
  boolean DelMax;                 //indicate this node is deleted
  boolean comp;                   //indicate this node is in comparison
}

//************************************************************************
// HeapSort class: This is the main class - execution starts from here
//                 after it is loaded into browser by html program.
//************************************************************************
public class HeapSort extends Applet implements ActionListener,
Runnable{
   int nMax = 15;
   int nodeSize = 25;
   int pause = 100;
   int step = 20;
```

```
Node nodes[] = new Node[nMax+1];
Node nodesBak[] = new Node[nMax+1];          //backup nodes
Thread runner;
Image offscreen;                             //new screen
Graphics offgraphics;                        //graph for paint
float apart, depth;
float x, y;
int xMax, yMax;                              //max coordinates
int Array[] = new int[nMax+1];
Color backColor = new Color(0, 140, 140);    //background color
Color nodeColor = new Color(255, 255, 0);
Color lineColor = new Color(255, 255, 255);
Color keyColor = new Color(0, 0, 0);
Color delMaxColor = new Color(0, 255, 255);  //del-max node color
Color compColor = new Color(0, 255, 0);      //comparing node color
int numSwap, numCompare, numMove;            //counts
boolean build_heap, del_max, fastDelMax;     //flags

//create buttons, labels and text areas
Button dataButton = new Button("Random Data");
Button buildButton = new Button("Build Heap");
Button delMaxButton = new Button("Delete Max");
Button delMaxButton1 = new Button("Fast Del-max");
Button runButton = new Button("Run All");

Label swapLabel = new Label("Swap", Label.RIGHT);
Label compareLabel = new Label("Comparison", Label.RIGHT);
Label moveLabel = new Label("Move", Label.RIGHT);
TextField swapResult = new TextField(3);
TextField compareResult = new TextField(3);
TextField moveResult = new TextField(3);

//************************************************************************
// init(): This function executes only once when program starts.
//************************************************************************
public void init(){
   // set the base screen layout
   setLayout(new BorderLayout());

   //set action listener for each button
   dataButton.addActionListener(this);
   runButton.addActionListener(this);
   buildButton.addActionListener(this);
   delMaxButton.addActionListener(this);
   delMaxButton1.addActionListener(this);

   //set enable and disable for each button
   dataButton.setEnabled(true);
   runButton.setEnabled(true);
   buildButton.setEnabled(false);
   delMaxButton.setEnabled(false);
   delMaxButton1.setEnabled(false);

   //create and arrange operation panel which contains buttons
   Panel operatePanel = new Panel();
   operatePanel.setLayout(new FlowLayout());
   operatePanel.setBackground(backColor);
   operatePanel.add(runButton);
   operatePanel.add(dataButton);
   operatePanel.add(buildButton);
   operatePanel.add(delMaxButton);
```

```
    operatePanel.add(delMaxButton1);

    //create and arrange output panel which displays counts
    Panel outputPanel = new Panel();
    outputPanel.setLayout(new FlowLayout());
    outputPanel.setBackground(backColor);
    outputPanel.add(swapLabel);
    outputPanel.add(swapResult);
    outputPanel.add(compareLabel);
    outputPanel.add(compareResult);
    outputPanel.add(moveLabel);
    outputPanel.add(moveResult);

    //arrange operation and output panels in the base screen
    add("South", operatePanel);
    add("North", outputPanel);

    //set background color for the base screen
    setBackground(backColor);

    //init variables
    numSwap = 0;
    numCompare = 0;
    numMove = 0;
    build_heap = true;
    del_max = true;
    fastDelMax = true;
}

//*********************************************************************
// start(): Do nothing here since everything is activated by
//          actionPerformed() in this program.
//*********************************************************************
public void start(){
}

//*********************************************************************
// actionPerformed(): This function handles button pressing which
//                    activates a particular process.
//*********************************************************************
public synchronized void actionPerformed(ActionEvent e){
  //get button object - the button which is pressed
  Object source = e.getSource();

  //run generate-data, build-heap and del-max all three processes
  if(source == runButton){
    //disable all buttons except stopButton
    dataButton.setEnabled(false);
    runButton.setEnabled(false);
    buildButton.setEnabled(false);
    delMaxButton.setEnabled(false);
    delMaxButton1.setEnabled(false);

    //make sure build-heap and del-max functions are enabled
    build_heap = true;
    del_max = true;                          //default operation
    fastDelMax = false;                      //non-default operation

    //create and run a new thread
    if(runner != null){
      runner = null;
```

```
    }
    if(runner == null){
      runner = new Thread(this);
      runner.start();
    }
  }

  //generate and display data
  if(source == dataButton){
    //enable build-heap button because it is next step
    dataButton.setEnabled(false);
    runButton.setEnabled(false);
    delMaxButton.setEnabled(false);
    delMaxButton1.setEnabled(false);
    buildButton.setEnabled(true);

    //suspend other processes for now
    build_heap = false;
    del_max = false;
    fastDelMax = false;

    //create and run a thread
    if(runner != null){
      runner = null;
    }
    if(runner == null){
      runner = new Thread(this);
      runner.start();
    }
  }

  //run build-heap after data is generated
  if(source == buildButton){
    //enable del-max button because it is next step
    delMaxButton.setEnabled(true);
    delMaxButton1.setEnabled(true);
    buildButton.setEnabled(false);

    //enable build-heap process
    build_heap = true;

    //notify wait() that the button is pressed and wait is over
    notify();
   }

  //run del-max after build-heap
  if((source == delMaxButton)||(source == delMaxButton1)){
    delMaxButton.setEnabled(false);
    delMaxButton1.setEnabled(false);
    if(source == delMaxButton)      //if regular del-max operation
      fastDelMax = false;
    else                            //if fast del-max operation
      fastDelMax = true;

    //enable del-max function and notify waiting is over
    del_max = true;
    notify();
  }
}

//*****************************************************************
```

```
 // stop(): Build-in function: invoked when exits browser.
 //*************************************************************************
 public void stop(){
   //kill the thread
   if(runner != null){
     runner = null;
   }
 }

//*************************************************************************
// reset(): This function is called by run() to reset buttons and
//          flags after execution is done.
//*************************************************************************
public void reset(){
  runButton.setEnabled(true);
  dataButton.setEnabled(true);
  buildButton.setEnabled(false);
  delMaxButton.setEnabled(false);
  delMaxButton1.setEnabled(false);
  build_heap = true;
  del_max = true;
  fastDelMax = true;
}

//*************************************************************************
// cleanScreen(): This function is called by run() to clean screen
//                when restart a new animation.
//*************************************************************************
public void cleanScreen(){
  int i;

  numSwap = 0;
  numCompare =0;
  numMove = 0;

  swapResult.setText(Integer.toString(numSwap));
  compareResult.setText(Integer.toString(numCompare));
  moveResult.setText(Integer.toString(numMove));

  for(i=0; i<=nMax; i++){
    nodes[i] = null;
    nodesBak[i] = null;
    Array[i] = 0;
  }
  repaint();                              //it invokes update()
  try{
    Thread.sleep(1000);                   //sleep 1000 millisecond
  } catch (InterruptedException e){}      //catch up exceptions if any
}

//*************************************************************************
// setup(): Because init() can be used only once, use this function
//          to reset everything whenever we rerun the animation.
//*************************************************************************
public void setup(){
  apart = (float)300;                     //distance between nodes
  depth = 1;                              //depth of a node
  xMax = getSize().width;                 //get max x coordinate
  yMax = getSize().height;                //get max y coordinate
  x = (float)xMax/2;                      //x of 1st node
  y = (float)50;                          //y of 1st node
```

```
    //create random data
    createRandomData();

    //setup each node - starts from node 1 (not node 0)
    for(int i=1; i<=nMax; i++){
      //create node and allocate memory
      Node n = new Node();                    //current node
      Node m = new Node();                    //corresponding backup node

      //setup x and y for current node
      n.x = x;
      n.y = y;
      n.key = Array[i];                       //assign value
      n.DelMax = false;                       //not been del-max yet
      n.comp = false;                         //no comparison yet
      nodes[i] = n;                           //put it in node array

      //setup x and y for backup node
      m.x = x;
      m.y = y;
      nodesBak[i] = m;

      //paint the node on screen
      repaint();
      try{
        Thread.sleep(1000);                   //slow down by sleep a while
      } catch (InterruptedException e){}

      // setup y for next node
      if(i==((int)Math.pow(2,depth)-1)){
        depth++;                              //increase depth
        apart = (apart/depth)+20-depth;       //change space between nodes
        y = y + 50;                           //increase y for 50 pixels
      }

      // setup x coordinate for next node
      if((i%2)==1){                           //if left node
        x = nodes[i/2+1].x - apart;           //(i/2 + 1)=parent of node i
      }
      else{                                   //if right node
        x = nodes[i/2].x + apart;
      }
    }
  }

  //*********************************************************************
  // run(): This is the main function which executes whenever
  //        thread.start() is called.
  //*********************************************************************
  public void run() {
    int i, n;
    n = nMax;

    // get current thread
    Thread thisThread = Thread.currentThread();

    // first, clean the screen
    cleanScreen();

    // then setup everything
    setup();
```

```
    // If build_heap flag is not set, suspend buildHeap process.
    // Build_heap flag is set by pressing buildButton. wait() can be
    // waked by notify() which is also invoked by pressing buildButton.
    try{
      Thread.sleep(100);
      synchronized(this){
        while(!build_heap){
          wait();
        }
      }
    }catch(InterruptedException e){}

    //execute buildHeap() buildButton is pressed
    buildHeap();
    repaint();

    //print the result of build-heap on server side - not on browser
    for(i=1; i<=n; i++)
      System.out.print(nodes[i].key+" ");
    System.out.println("(heap): comparision="+numCompare +"
      swap="+numSwap +"move="+numMove);

    //Suspend delMax process if del_max flag is not set
    try{
      Thread.sleep(100);
      synchronized(this){
        while(!del_max){
          wait();
        }
      }
    }catch(InterruptedException e){}

    //execute delMax() once delMax button is pressed
    if(!fastDelMax)                          //if regular del-max
      delMax(n);
    else                                     //if fast del-max
      delMaxFast(n);

    //print sorted results on the server side for developer
    for(i=1; i<=n; i++)
      System.out.print(nodes[i].key+" ");
    if(!fastDelMax)
      System.out.println("(sorted-regu): comparision="+numCompare +"
        swap="+numSwap +" move="+numMove);
    else
      System.out.println("(sorted-fast): comparision="+numCompare +"
        swap="+numSwap+" move="+numMove);

    reset();                                 //reset everything after done
  }

  //*********************************************************************
  // buildHeap(): This function performs build-heap operation.
  //*********************************************************************
  public void buildHeap(){
    int i, n;
    n = nMax;
    for(i=n/2; i>0; i--)
      perc_down(i,n);
  }
```

```
//*********************************************************************
// delMax(): This function performs the regular delete-max operations
//           which put the last leave at the root after the root is
//           deleted, and then sift it down by swapping it with its
//           larger child. This method takes two comparisons and one
//           swap at each level.
//*********************************************************************
public void delMax(int size){
  int j, n = size;
  for(j=1; j<=n; j++){
    if(size>1){
      //delete root
      nodes[1].DelMax = true;                    //mark it to change color
      swap(1, size);                             //swap root and last leaf

      //rebuild heap after deleting root
      size--;
      perc_down(1, size);
    }
  }
  nodes[1].DelMax = true;                        //mark last node as deleted
  repaint();                                     //paint once more
}

//*********************************************************************
// perc_down(): Utility function - handle perlocate down process.
//*********************************************************************
public void perc_down(int i, int size){
  int child, tmp;
  for(; i*2<=size; i=child){
    child = 2*i;                                 //left child

    if(child<size){
      updateComparison(child, child+1);
      if(nodes[child].key<nodes[child+1].key)
        child++;                                 //right child
    }

    updateComparison(i, child);
    if(nodes[i].key < nodes[child].key){         //swap parent and child
      swap(i, child);
    }
    else{
      break;
    }
  }
}

//*********************************************************************
// swap(): Utility function - exchange two nodes.
//*********************************************************************
public void swap(int i, int j){
  Node tmpNode = new Node();

  // animate the move of two nodes in opposite directions
  MoveNode(i, j);

  // then swap the two nodes in the array
  tmpNode = nodes[j];
  nodes[j] = nodes[i];
```

```
  nodes[i] = tmpNode;

  //update swap counts and print out result
  numSwap++;
  swapResult.setText(Integer.toString(numSwap));
}

//*********************************************************************
// delMaxFast(): Delete-max operation, using Floyd algorithm.  This
//               fast del-max operation starts at the empty root and
//               promotes the larger child, all the way down. This
//               takes only one comparison and one move at each level.
//               After the sift-down process, then sift up the hole
//               before inserting the last leaf, if last leaf has
//               larger value than the parent of the hole. This will
//               avoid swaps during sift up process.
//*********************************************************************
public void delMaxFast(int size){
  int last, parent, maxKey, child;
  parent = 1;
  child = 2;

  //1st node is not part of the heap. Use it to hold deleted root
  Node firstNode = new Node();
  firstNode.x = nodes[1].x;
  firstNode.y = 175;
  firstNode.DelMax = true;
  firstNode.comp = false;
  firstNode.key = -1;
  nodes[0] = firstNode;

  while(size>1){
    if(size!=2){
      nodes[1].DelMax = true;
      oneWayMove(1, 0);                          //delete max
    }

    //sift-down the hole, starting from root
    for(parent=1; parent*2<size; parent=child){
      child = 2*parent;

      if(child<size){
        updateComparison(child, child+1);
        if(nodes[child].key<nodes[child+1].key)
          child++;
      }
      oneWayMove(child, parent);                 //sift up larger child
    }

    last = size--;                               //reduce heap size by one

    //sift-up the last hole
    for(parent=child/2; parent>1; parent=child/2){
      if((child==last/2)||(parent==last/2)||(child==last))
        break;
      updateComparison(last, parent);
      if(nodes[last].key > nodes[parent].key){
        oneWayMove(parent, child);               //sift down parent
        child = parent;
      }
      else                                       //found correct hole
```

```
            break;
        }

        //Special case for size=2. This could be outside while()loop.
        if(size==1){
          updateComparison(1, 2);
          if(nodes[1].key > nodes[2].key){
            //swap node[1] and nodes[2] using one-way move
            nodes[1].DelMax = true;
            oneWayMove(1, 0);              //delete root
            oneWayMove(2, 1);              //move child up
            oneWayMove(0, 2);              //put deleted root to nodes[2]
          }
        }
        else{
          if(last != child)
            oneWayMove(last, child);       //fill the hole with last leaf
          oneWayMove(0, last);             //put deleted key to last node
        }
        repaint();
      }
      nodes[1].DelMax = true;              //change color of last 2 nodes
      nodes[2].DelMax = true;
      repaint();
    }

    //*********************************************************************
    // onWayMove(): Move one node-differ from swap which moves two nodes
    //*********************************************************************
    public void oneWayMove(int from, int to){
      Node tmpNode = new Node();

      MoveNode(from, to);

      // swap x, y coordinates
      tmpNode = nodes[to];
      nodes[to] = nodes[from];
      nodes[from] = tmpNode;
      numMove++;                           //update and display counts
      moveResult.setText(Integer.toString(numMove));
    }

    //*********************************************************************
    // updateComparison(): Visualize comparison nodes and update
    //                     comparison counts.
    //*********************************************************************
    public void updateComparison(int i, int j){
      nodes[i].comp = nodes[j].comp = true;
      repaint();
      try{
        Thread.sleep(pause*step/3);        //takes only 1/3 of swap time
      } catch (InterruptedException e){}

      nodes[i].comp = nodes[j].comp = false;
      numCompare++;
      compareResult.setText(Integer.toString(numCompare));
    }
```

```
//*********************************************************************
// update(): This function is invoked by reparint() to update the
//           screen. This function uses double-buffer technical which
//           creates an image as a new screen (offscreen) and a graph
//           (offgraphics) in the new screen. Then paint nodes in the
//           graph, and draw the new screen above old screen. Thus,we
//           do not have to erase the old paint during each update.
//*********************************************************************
public synchronized void update(Graphics g){
  Dimension d = getSize();

  //create an image as a new screen
  offscreen = createImage(d.width, d.height);

  //create a graph in the new screen
  offgraphics = offscreen.getGraphics();

  //paint the graph
  paintNode(offgraphics);
  paintLine(offgraphics);
  paintData(offgraphics);

  //draw the new screen above old screen
  g.drawImage(offscreen, 0, 0, null);
}

//*********************************************************************
// paintNode(): Paint nodes and their keys of the heap.
//*********************************************************************
public void paintNode(Graphics screen){
  int Xsize = 5, Ysize = 18;          //location of a key inside node
  super.paint(screen);

  for(int i=0; i<=nMax; i++){
    if((nodes[i]!=null)&&(nodes[i].key!=-1)){

      // paint node
      screen.setColor(nodeColor);     //set node color
      if(nodes[i].DelMax)             //set color for deleted node
        screen.setColor(delMaxColor);
      if(nodes[i].comp)               //set color for comparison node
        screen.setColor(compColor);
      screen.fillOval((int)nodes[i].x, (int)nodes[i].y, nodeSize,
        nodeSize);

      // fill the key inside node
      if(nodes[i].key > 9)            //adjust location of the key
        Xsize = 5;
      else
        Xsize = 9;
      screen.setColor(keyColor);      //set key color
      screen.drawString(Integer.toString(nodes[i].key),
        (int)(nodes[i].x+Xsize), (int)(nodes[i].y+Ysize));
    }
  }
}

//*********************************************************************
// paintLine(): Paint lines that link nodes.
//*********************************************************************
public void paintLine(Graphics screen){
```

```
  int parent;

  for(int i=1; i<=nMax; i++){
    if((i!=1)&&(nodesBak[i]!=null)){
      parent = i/2;
      if(parent < 1)
        parent = 1;
      screen.setColor(lineColor);
      screen.drawLine((int)(nodesBak[parent].x+nodeSize/2),
        (int)(nodesBak[parent].y+nodeSize+2),
        (int)(nodesBak[i].x+nodeSize/2), (int)(nodesBak[i].y));
    }
  }
}

//*********************************************************************
//This function paint the data array.
//*********************************************************************
public void paintData(Graphics screen){
  int Xsize, Ysize = 18;
  int x, y, width, height;
  int space = 3, i;

  //setup x and y of the first node
  width = getSize().width;
  height = getSize().height;
  y = height - 100;
  x = (width/2)-((nMax*nodeSize)/2)-((space*nMax-1)/2);

  super.paint(screen);
  // paint node
  for(i=1; i<=nMax; i++){
    if((nodes[i]!=null)&&(nodes[i].key!=-1)){
      screen.setColor(nodeColor);
      if(nodes[i].DelMax)
        screen.setColor(delMaxColor);
      screen.fillRect(x, y, nodeSize, nodeSize);

      // fill key inside node
      if(nodes[i].key > 9)
        Xsize = 5;
      else
        Xsize = 9;
      screen.setColor(keyColor);
      screen.drawString(Integer.toString(nodes[i].key),
        x+Xsize, y+Ysize);
    }
    //set x of next node
    x+=nodeSize + space;
  }
}

//*********************************************************************
// MoveNode(): Handles the animation process-determine the movement
//             of one or two nodes.
//*********************************************************************
public void MoveNode(int i, int j){
  float slope1=0;
  float slope2=0;

  // calculate slopes between two points
```

```
    if((i!=j)&&(nodes[i].x != nodes[j].x)){
      slope1 = Slope(nodes[i].x, nodes[j].x, nodes[i].y, nodes[j].y);
      slope2 = Slope(nodes[j].x, nodes[i].x, nodes[j].y, nodes[i].y);
    }

    // determine the increment of each step of a movement
    float x1Inc = (nodes[j].x-nodes[i].x)/step;
    float x2Inc = (nodes[i].x-nodes[j].x)/step;
    float y1Inc = (nodes[j].y-nodes[i].y)/step;
    float y2Inc = (nodes[i].y-nodes[j].y)/step;

    for(int n=0; n<step; n++){
      // update x and y
      nodes[i].x= (float)(nodes[i].x+x1Inc);
      nodes[j].x= (float)(nodes[j].x+x2Inc);
      if(nodes[i].x != nodes[j].x){
        nodes[i].y= (float)(nodes[i].y+slope1*x1Inc);
        nodes[j].y= (float)(nodes[j].y+slope2*x2Inc);
      }
      else{           //no slope if x1=x2 because slope=(y2-y1)/(x2-x1)
        nodes[i].y = (float)(nodes[i].y+y1Inc);
        nodes[j].y = (float)(nodes[j].y+y2Inc);
      }

      // repaint each move
      repaint();
      try{
        Thread.sleep(pause);
      } catch (InterruptedException e){}
    }
  }

  //*************************************************************************
  // Slope(): Calculate the slope between two nodes.
  //*************************************************************************
  public float Slope(float x1, float x2, float y1, float y2){
    float slope;
    slope = (float)((y2-y1)/(x2-x1));
    return slope;
  }

  //*************************************************************************
  // createRandomData(): Generate random data using Math.random().
  //*************************************************************************
  public void createRandomData()
  {
    int flag, theData, i, j;

    i=1;
    while(i<=nMax){
      theData=(int)(nMax * Math.random())+1;
      flag=0;
      for(j=1; j<=i; j++){
        if(theData==Array[j]){            //check if the data exists
      flag=1;
      break;
        }
      }
      if(flag==0){
        Array[i++]=theData;               //save the data if not exists
        System.out.print(Array[i-1]+" ");
```

```
        }
      }
      System.out.println("(presort)");
    }
}
```

HTML document

```
//*************************************************************************
// This html program brings above java applet (HeapSort) into browser
//*************************************************************************
<html>
<applet code="HeapSort.class" height=400 width=700>
</applet>
</html>
```

2. Animation of Four Algorithms Together

```
/*************************************************************************
                            Program : SortAll
                             Author : Jian Su
                               Data : December 1999
                 Programming Language : Java

This program is designed to execute quicksort, fast heapsort, heapsort,
and insertion sort concurrently.  It can also execute individual
algorithm separately.

The program contains 7 classes:
Node:          Contains basic elements of a node
SortAll:       Main class of this program - program starts from here
SortAnimation: Handle animation process
SortGeneral:   Base class for all sorting algorithms
quickSort:     Derived class from SortGeneral to handle quicksort
heapSortFast:  Derived class from SortGeneral to handle fast heapsort
heapSort:      Derived class from SortGeneral to handle heapsort
insertSort:    Derived class from SortGeneral to handle insertion sort
*************************************************************************/

import java.awt.*;
import java.util.*;
import java.applet.Applet;
import java.awt.event.*;

//************************************************************************
// Node class: a node contains, a key (data), its x and y coordinates
//             on the screen, and swap and comparison flags.
//************************************************************************
class Node {
  float x;
  float y;
  int key;
  boolean swap;
  boolean comp;
}

//************************************************************************
// SortAll class: This is the main program - program starts from here.
//                This class set-up interface.
//************************************************************************
public class SortAll extends Applet implements ActionListener{
  int nMax = 20;                //default max number of nodes
  int upLimit = 25;             //upper limit of data size for animation
  int nodeSize = 20;            //size of a node
  int fieldSize = 5;            //the size of text field
  int Array[];

  Panel controlPanel;           //overall control panel
  Panel displayPanel;           //display sorting animation
  Panel qSortPanel;             //quicksort panel
  Panel iSortPanel;             //inserton sort panel
  Panel hSortPanel;             //heapsort panel
  Panel hSortPanel1;            //fast heapsort panel
  Panel qSortCtrlPanel;         //quicksort control panel
  Panel hSortCtrlPanel1;        //fast heapsort panel
  Panel iSortCtrlPanel;         //inserton sort control panel
  Panel hSortCtrlPanel;         //heapsort control panel
```

```
sortAnimation qSort;          //quicksort class
sortAnimation iSort;          //inserton sort class
sortAnimation hSort;          //heapsort class
sortAnimation hSort1;         //fast heapsort class

Button startButton;           //start all sorting animation
Button resetButton;           //reset and generate data
Button qStartButton;          //start quicksort
Button hStartButton;          //start quicksort
Button iStartButton;          //start inserton sort
Button hStartButton1;         //start fast heapsort

Label qLabel;                 //quicksort label
Label iLabel;                 //insertion sort label
Label hLabel;                 //heapsort label
Label hLabel1;                //fast heapsort label
Label qSwapLabel;             //label for swap count of quicksort
Label iSwapLabel;
Label hSwapLabel;
Label hSwapLabel1;
Label qCompLabel;             //label for comparison count of quicksort
Label iCompLabel;
Label hCompLabel;
Label hCompLabel1;
Label hMoveLabel1;
Label dataSizeLabel;          //label for data size

TextField qSwapText;          //show result of swaps for quicksort
TextField iSwapText;
TextField hSwapText;
TextField hSwapText1;
TextField qCompText;          //show result of comparison for quicksort
TextField iCompText;
TextField hCompText;
TextField hCompText1;
TextField hMoveText1;
TextField dataSizeText;       //for enter data size

public Color backColor;       //background color
public Color nodeColor;
public Color keyColor;        //data color
public Color swapColor;       //color for swapping nodes

//***********************************************************************
// init(): This function executes only once when program starts.
//***********************************************************************
public void init(){
  Array = new int[nMax+1];

  //setup colors
  backColor = new Color(0, 140, 140);
  nodeColor = new Color(255, 255, 0);
  keyColor = new Color(0, 0, 0);
  swapColor = new Color(0, 255, 255);

  //set-up screen layout and color
  setLayout(new BorderLayout());
  setBackground(backColor);

  //setup overall control panel
```

```
controlPanel = new Panel();
startButton = new Button("Run All");
startButton.addActionListener(this);
resetButton = new Button("Reset");
resetButton.addActionListener(this);
dataSizeLabel = new Label("Size", Label.RIGHT);
dataSizeText = new TextField(fieldSize);

controlPanel.add(dataSizeLabel);
controlPanel.add(dataSizeText);
controlPanel.add(startButton);
controlPanel.add(resetButton);

//create and setup quicksort panel
qSortPanel = new Panel();
qSortPanel.setLayout(new BorderLayout());
qSortPanel.setBackground(backColor);
//create and setup quicksort control panel
qSortCtrlPanel = new Panel();
qLabel = new Label("Quicksort", Label.RIGHT);
qStartButton = new Button("Start");
qStartButton.addActionListener(this);
qSwapLabel = new Label("Swap", Label.RIGHT);
qSwapText = new TextField(fieldSize);
qCompLabel = new Label("Comparison", Label.RIGHT);
qCompText = new TextField(fieldSize);
//add buttons and labels to quicksort control panel
qSortCtrlPanel.add(qLabel);
qSortCtrlPanel.add(qStartButton);
qSortCtrlPanel.add(qSwapLabel);
qSortCtrlPanel.add(qSwapText);
qSortCtrlPanel.add(qCompLabel);
qSortCtrlPanel.add(qCompText);
//create quicksort canvas which handles quickSort animation
qSort = new sortAnimation(new quickSort(this));
//add quicksort control panel and canvas to quicksort panel
qSortPanel.add("South", qSortCtrlPanel);
qSortPanel.add("Center", qSort);

//create and setup heapsort panel
hSortPanel = new Panel();
hSortPanel.setLayout(new BorderLayout());
hSortPanel.setBackground(backColor);
hSortCtrlPanel = new Panel();
hLabel = new Label("Heapsort", Label.RIGHT);
hStartButton = new Button("Start");
hStartButton.addActionListener(this);
hSwapLabel = new Label("Swap", Label.RIGHT);
hSwapText = new TextField(fieldSize);
hCompLabel = new Label("Comparsion", Label.RIGHT);
hCompText = new TextField(fieldSize);
hSortCtrlPanel.add(hLabel);
hSortCtrlPanel.add(hStartButton);
hSortCtrlPanel.add(hSwapLabel);
hSortCtrlPanel.add(hSwapText);
hSortCtrlPanel.add(hCompLabel);
hSortCtrlPanel.add(hCompText);
hSort = new sortAnimation(new heapSort(this));
hSortPanel.add("South", hSortCtrlPanel);
hSortPanel.add("Center", hSort);
```

```
    //create and setup fast heapsort panel
    hSortPanel1 = new Panel();
    hSortPanel1.setLayout(new BorderLayout());
    hSortPanel1.setBackground(backColor);
    hSortCtrlPanel1 = new Panel();
    hLabel1 = new Label("Fast Heapsort", Label.RIGHT);
    hStartButton1 = new Button("Start");
    hStartButton1.addActionListener(this);
    hSwapLabel1 = new Label("Swap", Label.RIGHT);
    hSwapText1 = new TextField(fieldSize);
    hCompLabel1 = new Label("Comparsion", Label.RIGHT);
    hCompText1 = new TextField(fieldSize);
    hMoveLabel1 = new Label("Move", Label.RIGHT);
    hMoveText1 = new TextField(fieldSize);
    hSortCtrlPanel1.add(hLabel1);
    hSortCtrlPanel1.add(hStartButton1);
    hSortCtrlPanel1.add(hSwapLabel1);
    hSortCtrlPanel1.add(hSwapText1);
    hSortCtrlPanel1.add(hCompLabel1);
    hSortCtrlPanel1.add(hCompText1);
    hSortCtrlPanel1.add(hMoveLabel1);
    hSortCtrlPanel1.add(hMoveText1);
    hSort1 = new sortAnimation(new heapSort1(this));
    hSortPanel1.add("South", hSortCtrlPanel1);
    hSortPanel1.add("Center", hSort1);

    //create and setup insertion sort panel
    iSortPanel = new Panel();
    iSortPanel.setLayout(new BorderLayout());
    iSortPanel.setBackground(backColor);
    iSortCtrlPanel = new Panel();
    iLabel = new Label("Insertion sort", Label.RIGHT);
    iStartButton = new Button("Start");
    iStartButton.addActionListener(this);
    iSwapLabel = new Label("Swap", Label.RIGHT);
    iSwapText = new TextField(fieldSize);
    iCompLabel = new Label("Comparsion", Label.RIGHT);
    iCompText = new TextField(fieldSize);
    iSortCtrlPanel.add(iLabel);
    iSortCtrlPanel.add(iStartButton);
    iSortCtrlPanel.add(iSwapLabel);
    iSortCtrlPanel.add(iSwapText);
    iSortCtrlPanel.add(iCompLabel);
    iSortCtrlPanel.add(iCompText);
    iSort = new sortAnimation(new insertSort(this));
    iSortPanel.add("South", iSortCtrlPanel);
    iSortPanel.add("Center", iSort);

    //create display panel which holds four algorithm panels together
    displayPanel = new Panel();
    displayPanel.setLayout(new GridLayout(4, 0));
    displayPanel.add(qSortPanel);
    displayPanel.add(iSortPanel);
    displayPanel.add(hSortPanel);
    displayPanel.add(hSortPanel1);

    //setup screen - arrange overall control panel and display panel
    add("Center", displayPanel);
    add("South", controlPanel);
}
```

```
//*************************************************************************
// start(): Do nothing here since everything is activated by
//          actionPerformed() in this program.
//*************************************************************************
public void start(){
  reset();
}

//*************************************************************************
// reset(): Clears the screen and reset everything.
//*************************************************************************
public void reset(){
  //enable buttons
  startButton.setEnabled(true);
  qStartButton.setEnabled(true);
  hStartButton.setEnabled(true);
  hStartButton1.setEnabled(true);
  iStartButton.setEnabled(true);

  //function call to reset all counts to 0
  qSwapCount(0);
  qCompCount(0);
  hSwapCount(0);
  hCompCount(0);
  hSwapCount1(0);
  hCompCount1(0);
  hMoveCount1(0);
  iSwapCount(0);
  iCompCount(0);

  //clean data array
  for(int i=0; i<=nMax; i++)
    Array[i] = 0;

  //generate random data
  createRandomData();

  //reset individual algorithm by calling its own reset() function
  qSort.reset(Array);
  hSort.reset(Array);
  hSort1.reset(Array);
  iSort.reset(Array);
}

//*************************************************************************
// actionPerformed(): This function handles button pressing which
//                    activates a particular process.
//*************************************************************************
public void actionPerformed(ActionEvent e){
  Object source = e.getSource();
  //if Run All button is pressing
  if(source == startButton){
    //disable all start-buttons while execution is in progress
    startButton.setEnabled(false);
    qStartButton.setEnabled(false);
    hStartButton.setEnabled(false);
    hStartButton1.setEnabled(false);
    iStartButton.setEnabled(false);

    //start all sorting processes
```

```
      qSort.start();
      hSort.start();
      hSort1.start();
      iSort.start();
    }

    //if Start button of quicksort is pressing
    if(source == qStartButton){
      startButton.setEnabled(false);
      qStartButton.setEnabled(false);
      //start quicksort process
      qSort.start();
    }

    //if Start button of heapsort is pressing
    if(source == hStartButton){
      startButton.setEnabled(false);
      hStartButton.setEnabled(false);
      hSort.start();
    }

    //if Start button of fast heapsort is pressing
    if(source == hStartButton1){
      startButton.setEnabled(false);
      hStartButton1.setEnabled(false);
      hSort1.start();
    }

    //if Start button of insertion sort is pressing
    if(source == iStartButton){
      startButton.setEnabled(false);
      iStartButton.setEnabled(false);
      iSort.start();
    }

    //if reset button is pressing, reset everything
    if(source == resetButton){
      //get data size
      nMax = Integer.parseInt(dataSizeText.getText());
      Array = new int[nMax+1];
      reset();
    }
  }

  //************************************************************************
  // stop(): Build-in function: invoked when exits browser.
  //************************************************************************
  public void stop(){
    qSort.stop();
    hSort.stop();
    hSort1.stop();
    iSort.stop();
  }

  //************************************************************************
  // createRandomDate(): Generate random data and put them in array
  //************************************************************************
  public void createRandomData(){
    int flag, theData, i, j;

    i=1;
```

```
    while(i<=nMax){
      theData=(int)(nMax * Math.random())+1;
      flag=0;
      for(j=1; j<=i; j++){
        if(theData==Array[j]){
          flag=1;
        break;
        }
      }
      if(flag==0){
        Array[i++]=theData;
        if(nMax <= upLimit)
          System.out.print(Array[i-1]+" ");
      }
    }
    System.out.println("(presort: size="+nMax+")");
  }

  //************************************************************************
  //Following functions print out the counts of swaps and comparisons.
  //************************************************************************
  public void qSwapCount(int count){
    qSwapText.setText(Integer.toString(count));
  }
  public void qCompCount(int count){
    qCompText.setText(Integer.toString(count));
  }
  public void hSwapCount(int count){
    hSwapText.setText(Integer.toString(count));
  }
  public void hCompCount(int count){
    hCompText.setText(Integer.toString(count));
  }
  public void hSwapCount1(int count){
    hSwapText1.setText(Integer.toString(count));
  }
  public void hCompCount1(int count){
    hCompText1.setText(Integer.toString(count));
  }
  public void hMoveCount1(int count){
    hMoveText1.setText(Integer.toString(count));
  }
  public void iSwapCount(int count){
    iSwapText.setText(Integer.toString(count));
  }
  public void iCompCount(int count){
    iCompText.setText(Integer.toString(count));
  }
}

//************************************************************************
// sortAnimation class: Handles the animation process.
//************************************************************************
class sortAnimation extends Applet implements Runnable{
  int nMax = 20;
  int upLimit = 25;
  int nodeSize = 20;
  private Thread runner;
  private sortGeneral algorithm;
  private Image offscreen;
  private Graphics offgraphics;
```

```
private int pause = 60;
private int step = 20;
private int xMax, yMax, x, space=3;
private Node nodes[];
Color backColor = new Color(0, 140, 140);
Color nodeColor = new Color(255, 255, 0);
Color keyColor = new Color(0, 0, 0);
Color swapColor = new Color(0, 255, 255);
Color compColor = new Color(0, 255, 0);

//************************************************************************
//constructor - bring in a sorting algorithm as parameter
//************************************************************************
public sortAnimation(sortGeneral sortAlgorithm){
  algorithm = sortAlgorithm;
  //put this animation program above the sorting algorithm
  algorithm.setParent(this);
}

//if no thread is running, create a thread
public void start(){
  if(runner == null){
    runner = new Thread(this);
    runner.start();
  }
}

//************************************************************************
// run(): Invoked by start() to execute sorting and animation.
//************************************************************************
public void run(){
  algorithm.sort(nodes);
}

//************************************************************************
// moveNode(): Handle the animation of swapping nodes. The move path
//             of each node is determined by math.sin() function.
//************************************************************************
public void moveNode(int front, int back){
  if(nodes.length -1> upLimit)        //if size>upLimit, no animation
    return;
  int height, base;
  float xMove, yMove;
  double radians = (float)0, radiansInc;

  xMove = (nodes[back].x - nodes[front].x)/step;
  height = getSize().height;
  base = height - height/3;           //lower bound
  height = height/3;                  //upper bound
  radiansInc = 3.14156/step;          //half cycle of sine

  for(int n=0; n<step; n++){
    // update x and y axes
    radians += radiansInc;
    yMove = base - (float)Math.sin(radians) * height-10;
    if(n == (step -1))
      yMove = base;
    nodes[front].x= (float)(nodes[front].x+xMove);
    nodes[back].x= (float)(nodes[back].x-xMove);
    nodes[back].y= (float)(yMove);
    nodes[front].y= (float)(yMove);
```

```
    //paint each move
    Thread thisThread = Thread.currentThread();
    if(thisThread == runner){
      repaint();
    }
    try{
      if((nodes[front].key == -1)||(nodes[back].key == -1))
        Thread.currentThread().sleep(pause/3);
      else
        Thread.currentThread().sleep(pause);
    } catch (InterruptedException e){}
  }
}

//paint the comparison nodes with new color
public void markCompNodes(){
  if(nodes.length > upLimit)
    return;
  Thread thisThread = Thread.currentThread();
  if(thisThread == runner){
    repaint();
  }
  try{
    Thread.currentThread().sleep(pause*step/3);
  } catch (InterruptedException e){}
}

//**********************************************************************
// update(): This function is invoked by reparint() to update the
//           screen. This function uses double-buffer technical which
//           creates an image as a new screen (offscreen) and a graph
//           (offgraphics) in the new screen. Then paint nodes in the
//           graph, and draw the new screen above old screen. Thus,we
//           do not have to erase the old paint during each update.
//**********************************************************************
public void update(Graphics screen){
  int Xsize, Ysize = 15;

  Dimension d = getSize();

  //create an image as a new screen
  offscreen = createImage(d.width, d.height);

  //create a graph in the new screen
  offgraphics = offscreen.getGraphics();
  offgraphics.setColor(this.getBackground());
  offgraphics.fillRect(0, 0, d.width, d.height);

  //if size > 25, paint warning message
  if(nodes.length-1 > upLimit){
    Font newFont = new Font("TimesRoman", Font.BOLD, 17);
    offgraphics.setFont(newFont);
    offgraphics.setColor(Color.white);
    offgraphics.drawString("Size > 25. Run with no animation?", 230,
      (int)nodes[1].y);
    screen.drawImage(offscreen, 0, 0, null);
    return;
  }

  //paint nodes in offgraphics
```

```
    for(int i=1; i<=nMax; i++){
      if(nodes[i]!=null){
        //paint node
        offgraphics.setColor(nodeColor);
        if(nodes[i].swap)
          offgraphics.setColor(swapColor);
        if(nodes[i].comp)
          offgraphics.setColor(compColor);
        if(nodes[i].key == -1)                    //-1 indicates no key
          offgraphics.setColor(backColor);
        offgraphics.fillRect((int)nodes[i].x, (int)nodes[i].y,
          nodeSize, nodeSize);

        // fill key inside node
        if(nodes[i].key!=-1){
          if(nodes[i].key > 9)
            Xsize = 3;
          else
            Xsize = 7;
          offgraphics.setColor(keyColor);
          offgraphics.drawString(Integer.toString(nodes[i].key),
            (int)(nodes[i].x+Xsize), (int)(nodes[i].y+Ysize));
        }
      }
    }
    //draw the new screen on old screen
    screen.drawImage(offscreen, 0, 0, null);
  }

  //************************************************************************
  //stop(): Invoked when exit browser
  //************************************************************************
  public void stop(){
    if(runner != null){
      runner = null;
    }
  }

  //************************************************************************
  //reset(): Resets nodes and screen.
  //************************************************************************
  public void reset(int array[]){
    //make sure that no process is still running
    stop();
    nMax = array.length-1;

    //create a new set of nodes
    nodes = new Node[array.length];
    xMax = getSize().width;
    yMax = getSize().height;
    x = (xMax/2)-((nMax*nodeSize)/2)-((space*nMax-1)/2);

    for(int i=0; i<=nMax; i++){
      Node tmpNode = new Node();
      tmpNode.key = array[i];
      tmpNode.y = yMax - yMax/3;
      tmpNode.x = x;
      tmpNode.swap = false;
      tmpNode.comp = false;
      nodes[i] = tmpNode;
      x += nodeSize + space;
```

```
    }

    //reset variables of individual algorithm
    algorithm.reset();

    //paint the new nodes
    repaint();
  }
}

//**************************************************************************
// sortGeneral: Base class for sorting algorithms. It contains the
//              common methods and variables shared by all algorithms
//**************************************************************************
class sortGeneral{
  int upLimit = 25;
  protected sortAnimation parent;
  protected SortAll ctrlPanel;
  int swapCount, compCount, moveCount;

  //bring in sortAnimation program to handle animation process
  public void setParent(sortAnimation p){
    parent = p;
  }

  //**************************************************************************
  // reset(): Clear counts.
  //**************************************************************************
  public void reset(){
    swapCount = 0;
    compCount = 0;
    moveCount = 0;
  }

  //**************************************************************************
  // sort(): Virtual function - will be implemented by derived classes.
  //**************************************************************************
  public void sort(Node nodes[]){
  }

  //**************************************************************************
  // swap(): This common function will be called by derived classes.
  //**************************************************************************
  public void swap(Node nodes[], int front, int back){
    Node tmpNode = new Node();

    if((nodes[front]!=null)&&(nodes[back]!=null)){
      if((front != back)&&((nodes.length-1)<=upLimit)){
        //animate swap process
        nodes[front].swap = nodes[back].swap = true;
        parent.moveNode(front, back);
        nodes[front].swap = nodes[back].swap = false;
      }
      //swap data after animation
      tmpNode = nodes[front];
      nodes[front] = nodes[back];
      nodes[back] = tmpNode;
    }
  }
```

```
  //*************************************************************************
  //markNodes(): Mark and change the color of comparison nodes.
  //*************************************************************************
  public void markNodes(Node nodes[], int i, int j){
      if((nodes.length-1)>upLimit)
        return;
      nodes[i].comp = nodes[j].comp = true;
      parent.markCompNodes();
      nodes[i].comp = nodes[j].comp = false;
  }
}

//*************************************************************************
// quickSort class: Derived class of sortGeneral to handle quicksort.
//*************************************************************************
class quickSort extends sortGeneral{
  //constructor bringa in SortAll that defines text fields for counts
  public quickSort(SortAll p){
    ctrlPanel = p;
  }

  //*************************************************************************
  // sort(): Implement the virtual function sort() of base class.
  //*************************************************************************
  public void sort(Node nodes[]){
    quicksort(nodes, 1, nodes.length-1);
    parent.repaint();

    //print the results on server side (not the animation screen)
    if(nodes.length<=upLimit){
      for(int i=1; i<nodes.length; i++){
        System.out.print(nodes[i].key +" ");
      }
    }
    System.out.println("Sorted(quicksort): comparison="+compCount +"
      swap="+swapCount);
  }

  //*************************************************************************
  // quicksort(): Quicksort main program
  //*************************************************************************
  public void quicksort(Node nodes[], int left, int right){
    int l = left;
    int r = right;
    int pivot = (l+r)/2;
    int mid = nodes[pivot].key;

    do{
      while(nodes[l].key < mid){
        markNodes(nodes, pivot, l);
        ctrlPanel.qCompCount(++compCount);
        l++;
      }
      while(nodes[r].key > mid){
        markNodes(nodes, pivot, r);
        ctrlPanel.qCompCount(++compCount);
        r--;
      }
      if(l <= r){
        markNodes(nodes, l, r);
        ctrlPanel.qCompCount(++compCount);
```

```
        swap(nodes, l, r);
        ctrlPanel.qSwapCount(++swapCount);
        l++; r--;
      }
    }while(l <= r);

    if(r > left){
      quicksort(nodes, left, r);
    }
    if(l < right){
      quicksort(nodes, l, right);
    }
  }
}

//*************************************************************************
// insertSort class: Derived class of sortGeneral to handle insert sort
//*************************************************************************
class insertSort extends sortGeneral{
  int min = -1000;                    //put min key in front as sentinel

  //constructor
  public insertSort(SortAll p){
    ctrlPanel = p;
  }

  //*************************************************************************
  // sort(): Implement the virtual function sort() of base class.
  //*************************************************************************
  public void sort(Node nodes[]){
    insertsort(nodes, nodes.length-1);
    parent.repaint();
    if(nodes.length<=upLimit){
      for(int i=1; i<nodes.length; i++){
        System.out.print(nodes[i].key +" ");
      }
    }
    System.out.println("Sorted(insertion sort): comparison="+compCount
      +" swap="+swapCount);
  }

  //*************************************************************************
  // insertsort(): Insertion sort main program.
  //*************************************************************************
  public void insertsort(Node nodes[], int n){
    int theKey, p, j;

    nodes[0].key = min;               //sentinel-no element go beyond it
    for (p=2; p<=n; p++){             //from pass p = 2 through n
      theKey = nodes[p].key;
      for(j=p; theKey<nodes[j-1].key; j--){
        markNodes(nodes, j-1, j);
        ctrlPanel.iCompCount(++compCount);
        swap(nodes, j-1, j);
        ctrlPanel.iSwapCount(++swapCount);
      }
      markNodes(nodes, j-1, p);
      ctrlPanel.iCompCount(++compCount);
    }
  }
```

```
}


//*************************************************************************
// heapSort class: Derived class of sortGeneral to handles heapsort.
//*************************************************************************
class heapSort extends sortGeneral{
  Node nodes[];

  //constructor
  public heapSort(SortAll p){
    ctrlPanel = p;
  }

  //*************************************************************************
  // sort(): Implement the virtual function sort() of base class.
  //*************************************************************************
  public void sort(Node nodeArr[]){
    nodes = nodeArr;
    buildHeap();
    delMax(nodes.length-1);
    parent.repaint();
    if(nodes.length<=upLimit){
      for(int i=1; i<nodes.length; i++){
        System.out.print(nodes[i].key +" ");
      }
    }
    System.out.println("Sorted(heapsort): comparison="+compCount +"
          swap="+swapCount);
  }

  //*************************************************************************
  // buildHeap(): Perform build-heap function of heapsort
  //*************************************************************************
  public void buildHeap(){
    int i, n;
    n = nodes.length-1;
    for(i=n/2; i>0; i--)
      perc_down(i,n);
  }

  //*************************************************************************
  // delMax(): This function performs the regular delete-max operations
  //           which put the last leave at the root after the root is
  //           deleted, and then sift it down by swapping it with its
  //           larger child. This method takes two comparisons and one
  //           swap at each level.
  //*************************************************************************
  public void delMax(int size){
    int j, n = size;
    for(j=1; j<=n; j++){
      if(size>1){
        //delete root
        swap(nodes, 1, size);
        ctrlPanel.hSwapCount(++swapCount);

        //rebuild heap after deleting root
        size--;
        //if last two nodes in order, sorting done
        if((size == 2)&&(nodes[1].key<nodes[2].key)){
```

```
            markNodes(nodes, 1, size);          //mark comparison nodes
            ctrlPanel.hCompCount(++compCount);  //update comparison count
            break;
          }
          perc_down(1, size);
        }
      }
    }

    //*************************************************************************
    // perc_down(): Handle perlocate down process of heapsort.
    //*************************************************************************
    public void perc_down(int i, int size){
      int child, tmp;
      for(; i*2<=size; i=child){
        child = 2*i;
        if(child<size){
          markNodes(nodes, child, child+1);
          ctrlPanel.hCompCount(++compCount);
          if(nodes[child].key<nodes[child+1].key)
            child++;
        }

        markNodes(nodes, i, child);
        ctrlPanel.hCompCount(++compCount);
        if(nodes[i].key < nodes[child].key){
          swap(nodes, i, child);
          ctrlPanel.hSwapCount(++swapCount);
        }
        else{
          break;
        }
      }
    }
}

//*************************************************************************
//heapSort1 class: Derived class of sortGeneral to handle fast heapSort
//                 This sorting uses Floyd algorithm for delete-max.
//*************************************************************************
class heapSort1 extends sortGeneral{
  Node nodes[];

  //constructor
  public heapSort1(SortAll p){
    ctrlPanel = p;
  }

  //*************************************************************************
  // sort(): Implement the virtual function sort() of base class.
  //*************************************************************************
  public void sort(Node nodeArr[]){
    nodes = nodeArr;
    buildHeap1();
    delMaxFast(nodes.length-1);
    parent.repaint();
    if(nodes.length<=upLimit){
      for(int i=1; i<nodes.length; i++){
        System.out.print(nodes[i].key +" ");
      }
    }
```

```
  System.out.println("Sorted(fast heapsort): comparison="+compCount +"
    swap="+swapCount +" move="+moveCount);
}

//*********************************************************************
//buildHeap1(): Identical to buildHeap function of regular heapsort.
//*********************************************************************
public void buildHeap1(){
  int i, n;
  n = nodes.length-1;
  for(i=n/2; i>0; i--)
    perc_down1(i,n);
}

//*********************************************************************
// perc_down1(): Identical to perc_down of regular heapsort.
//*********************************************************************
public void perc_down1(int i, int size){
  int child, tmp;
  for(; i*2<=size; i=child){
    child = 2*i;
    if(child<size){
      markNodes(nodes, child, child+1);
      ctrlPanel.hCompCount1(++compCount);
      if(nodes[child].key<nodes[child+1].key)
        child++;
    }

    markNodes(nodes, i, child);
    ctrlPanel.hCompCount1(++compCount);
    if(nodes[i].key < nodes[child].key){
      swap(nodes, i, child);
      ctrlPanel.hSwapCount1(++swapCount);
    }
    else{
      break;
    }
  }
}

//*********************************************************************
// delMaxFast(): Delete-max operation, using Floyd algorithm.  This
//               fast del-max operation starts at the empty root and
//               promotes the larger child, all the way down. This
//               takes only one comparison and one move at each level.
//               After the sift-down process, then sift up the hole
//               before inserting the last leaf, if last leaf has
//               larger value than the parent of the hole. This will
//               avoid swaps during sift up process.
//*********************************************************************
public void delMaxFast(int size){
  int last, parent, maxKey, child;
  parent = 1;
  child = 2;

  Node firstNode = new Node();
  firstNode.x = nodes[1].x-20;
  firstNode.y = nodes[1].y;
  firstNode.swap = true;
  firstNode.comp = false;
  firstNode.key = -1;
```

```
    nodes[0] = firstNode;

    while(size>1){
      if(size!=2){
        oneWayMove(1, 0);
      }

      //sift-down the hole, starting from root
      for(parent=1; parent*2<size; parent=child){
        child = 2*parent;

        if(child<size){
          markNodes(nodes, 1, size);
          ctrlPanel.hCompCount1(++compCount);
          if(nodes[child].key<nodes[child+1].key)
            child++;
        }
        oneWayMove(child, parent);
      }

      //reduce heap size by one
      last = size--;

      //sift-up the last hole
      for(parent=child/2; parent>1; parent=child/2){
        if((child==last/2)||(parent==last/2)||(child==last))
          break;
        markNodes(nodes, 1, size);
        ctrlPanel.hCompCount1(++compCount);
        if(nodes[last].key > nodes[parent].key){
          oneWayMove(parent, child);
          child = parent;
        }
        else
          break;
      }

      //Special case for size=2
      if(size==1){
        markNodes(nodes, 1, size);
        ctrlPanel.hCompCount1(++compCount);
        if(nodes[1].key > nodes[2].key){
          //swap node[1] and nodes[2] using one-way move
          oneWayMove(1, 0);
          oneWayMove(2, 1);
          oneWayMove(0, 2);
        }
      }
      else{
        if(last != child)
          oneWayMove(last, child);
        oneWayMove(0, last);
      }
    }
  }

  //*********************************************************************
  // oneWayMove(): Move one node instead swap two nodes
  //*********************************************************************
  public void oneWayMove(int from, int to){
    Node tmpNode = new Node();
```

```
        nodes[from].swap = true;
        parent.moveNode(from, to);
        nodes[from].swap = false;

        // need swap x, y coordinates for animation
        tmpNode = nodes[to];
        nodes[to] = nodes[from];
        nodes[from] = tmpNode;
        ctrlPanel.hMoveCount1(++moveCount);
      }
    }
```

HTML document

```
//*************************************************************************
//This html program will bring above java applet (SortAll) into browser
//*************************************************************************
<html>
<applet code="SortAll.class" height=400 width=700>
</applet>
</html>
```

VITA

Jian Su

Candidate for the Degree of

Master of Science

Thesis: VISUALIZATION AND ANIMATION OF SORTING ALGORITHMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Beijing, China, daughter of Jiayuan Sun and Peizhi su, married to Sean Zhang.

Education: Received Bachelor of Science degree in Industrial Economics from Chinese People's University, Beijing, China in June 1986. Completed the Requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December 1999.

Experience: Worked as a Research Engineer for Institute of Techno-Economics, State Planning Commission, Beijing, China from 1986 to 1989