3 Principles of Algorithmic Animation

In this chapter we discuss some general issues regarding the production and design of algorithmic animations. A good animation is both a technical production and a work of art. Awareness of the composition principles used by other animators and by visual artists can help programmers to produce better animations and can suggest how to incorporate those ideas in new animation systems.

3.1 Animations and the visual channel

Information is absorbed by humans mainly through two sensory channels – the visual and the audio channel. During a class, both are stimulated by the lecturer. In the social sciences language tends to play the dominant role. In the sciences, the visual channel is as important as the audio channel, or plays even the dominant role. The new field of *computational visualization* studies the role of computer generated illustrations in the sciences and how to improve the information they convey [Strothotte 98].

In computer science and mathematics, concepts are illustrated frequently using pictures. Data structures, especially, are explained using a concrete representation for the data at hand: trees are drawn as nodes connected by arrows, lists as boxes with pointers to sequences of more boxes, arrays as a box with a sequence of numbers inside. Such visualizations, coupled with oral explanations, are the best way to explain the dynamics of algorithms. Psychologists who have studied the way humans build and retain concepts, argue that using two or more different encodings for the same idea helps in remembering and handling concepts creatively [Paivio 90]. Diagrams can influence reasoning [Bauer 93]. This means that a verbal and a visual encoding which mutually reinforce are best for teaching purposes. This is the dual coding hypothesis, already mentioned in Chapter 2.

On the other hand, there have been many efforts directed towards exploring how best to convey information from a machine to a user in order to avoid visual overload. Pilots of airplanes, for example, have experimented with vibrations from a wristwatch. Such mechanical signals are needed because the visual field of pilots is cluttered with too many displays. Therefore, important information can be best given using a vibration alarm which activates a third sensory channel. It is well

known that attention is raised by a signal which clearly pops up against the background [Theeuwes 98]. Such tricks can be applied to algorithmic animations using visual signals that direct the attention of the viewer to the interesting parts of the simulation.

3.1.1 Traditional animations

In an influential article, John Lasseter [1987] reviewed which principles of traditional animation could and should be applied to computer animations, especially in three dimensional simulations. He was not referring to algorithmic animation specifically, but the principles he discusses can also be partially applied to our topic. In traditional animations, the animator has to keep the audience interested in the story's plot and has to drive its attention to certain aspects of the images – it is therefore informative to look at the abstract principles that have been formulated in this field.

The subset of principles for good animations mentioned by Lasseter and relevant for us is the following:

- *Anticipation*. A visual cue can direct the attention to a certain part of the image where an action is about to occur. Timely anticipation allows the viewer to better understand an action which stays in the screen just for a moment. An anticipation action could be making an animation object blink or change color before it starts moving.
- *Timing*. There are three parameters that can be controlled during an animation: the time for the anticipation of an action, the time for the action, and the time for the reaction to the action. If everything is too fast, the viewer might not understand; if it is too slow, the viewer loses interest.
- *Staging*. This means that the attention of the viewer should be directed to where the "action is". No two important actions should occur simultaneously at two points where the user cannot perceive them at the same time. Motion or color can help to highlight the important action. If everything is moving, an object at rest draws the attention. Objects that move in coordination form a unity.
- *Overlapping*. A second action can be started shortly before the first action is completed, in order to keep the interest of the viewer.
- *Slow In and Out*. A moving object is more appealing if it moves faster at the extremes (the turn-around points in paths). This has to do with the physics of movement, but it could be useful also in algorithmic animation, where the animation objects try to resemble physical objects.
- *Secondary action*. An event that reinforces the primary action. In a sorting algorithm, for example, a number which has attained its final position in an array can be colored green to signal completion, while the other numbers are black.
- *Appeal*. The viewer should like what she sees.

For our purposes, the main message that we can extract from this listing, is that the attention of the user should be directed towards the main operations, one at a time. Using anticipation seems to be a good way to prepare the user for an action, by blinking an object, using an arrow, or changing its color (highlighting objects is one of the primitive actions in the animation scripting language to be discussed in Chapter 7). Anticipation works even against the will of the viewer: psychologists know that when a person is given the explicit command to move the eyes towards a specific point on a screen, any object which suddenly appears somewhere else captures the attention of the eyes. The viewer cannot move the eyes voluntarily to the ordered point; his vision is dragged towards the abrupt onset event [Theeuwes 98]. Therefore, anticipation can be produced by releasing the appropriate stimuli at the correct time.

Staging is always the main problem with algorithmic animations. Since we are executing an algorithm, the action can be anywhere on the screen. It is important therefore to try to give the user some orientation. And timing is of course very important: the animation speed should be variable, so that expert and novice users can adjust it to fit their needs. We will return to this problem when we talk about hierarchical stepping.

3.1.2 Visual cues and Gestalt principles

Algorithm animators can take advantage of the automatic pattern recognition abilities of the visual channel if visual perception is exploited effectively [Friedhoff 00]. Visual perception is largely an unconscious process. We perceive unity in a group of objects, even if they are separated, or we perceive differences even if the objects look initially the same.

Phenomena of visual perception were studied by the Gestalt school of Psychology at the turn of the 19th into the 20th century. Working in Germany, Max Wertheimer, Kurt Koffka, and Wolfgang Köhler developed the main Gestalt ideas in the 1920s and 1930s.

Gestalt psychologists have summarized their conclusions about human visual perception in a set of principles that can be illustrated easily. Gestalt principles have been used as a guide for general design [Moore 93], and visual screen design of Web applications [Chang 02]. Over time, several different lists of Gestalt principles have been produced, but the classical reference is [Wertheimer 23], who concentrated on similarity in location, shape, fate, etc.

The classical Gestalt principles are:

Figure and Background. Figures are perceived when they contrast with the background. When the background is ambiguous, as in Fig. 3.1, different interpretations arise.

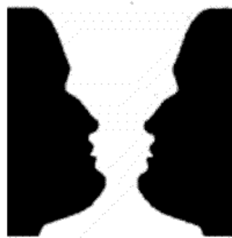

Figure 3.1 A vase or two faces?

Similarity. Objects with a feature in common such as shape, color, size, texture, orientation, etc. are perceived as belonging together; as in the examples shown in Figure 3.2.

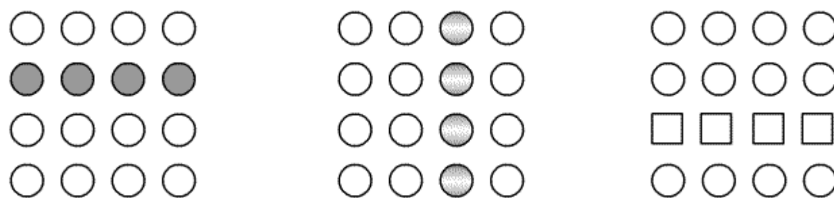

Figure 3.2 Similar objects are grouped by perception

Proximity or Contiguity. We perceive unity if objects are contiguous, especially when compared to other objects. In Figure 3.3 we perceive two columns made of other two columns of circles. The grouping is immediately suggested by the distance between the elements, although this could be considered just a matrix of circles. We cannot avoid perceiving this grouping. In Chapter 4, the history table for the execution of the bubble sort algorithm Figure 4.10 has been produced with this principle in mind. The temporal grouping is made vertically.

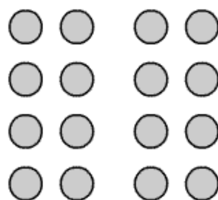

Figure 3.3 Proximity of objects

Continuity. When visually segmenting objects, our brain prefers to deal with continuous objects. The cross below, for example, is perceived as two crossing lines

instead of four lines meeting at a common point. The second sketch is perceived as a line behind a circle, instead as two lines touching the circle.

Figure 3.4 Good continuity

Closure. We tend to complete the shape of figures based on partial information about the boundary. Kanizsa's triangle is the traditional example: we see a triangle where there is none. Or we see a large square, where the boundary is just suggested by a few small circles.

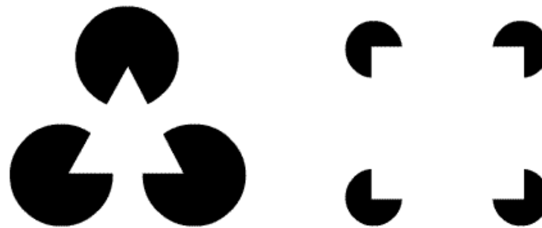

Figure 3.5 Kanizsa's triangle and square

Area. The larger of two figures, where the smaller is inside the larger one, is perceived as the background. The smaller figure seems to lie above the larger one.

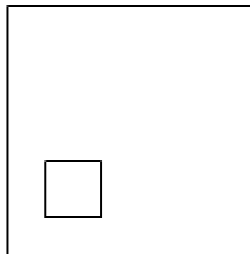

Figure 3.6 The principle of area

Symmetry. We prefer to segment a complex drawing into simple symmetrical objects. Figure 3.6 is perceived as two overlapped squares, not as three pieces in contact. Symmetry may also refer to parallelism. Parallel lines are often grouped together.

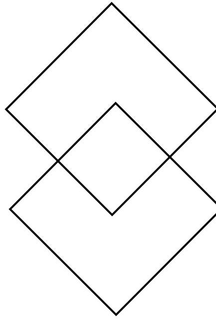

Figure 3.7 Symmetry of objects

3.1.3 Data structures of the mind and Gestalt learning theory

What Gestalt principles make evident is that perceiving the world is not just a passive process. When we are seeing, we are actively testing hypothesis about everything we see and our brain is trying to organize and segment the whole visual impression into known objects or patterns. Seeing is an *active process* and the groupings that we make reflect the kind of "data structures" that our brain uses to organize information. When there is a correspondence between those data structures of the brain and what we see, we can better understand and also better memorize patterns.

One simple example for the difficulty of even seeing information is pattern XIX in Wertheimer's classic paper of 1923.

Figure 3.8 A difficult pattern for the eye [Wertheimer 23]

This pattern is difficult to segment. The principle of proximity tries to pair the nearer points, but this conflict with the similarity of shape, which gives preference to a segmentation in which larger circles are grouped together. Both associations could be possible, but the conflict makes our eyes jump back and forth between the two interpretations. There is a mismatch between our attempts at categorization and what we see.

Sometimes, deciding on an interpretation takes some time, but once we have adopted one, the pattern is easier to see, as shown in Figure 3.9. The mind settles here on an association by shape, rather than proximity.

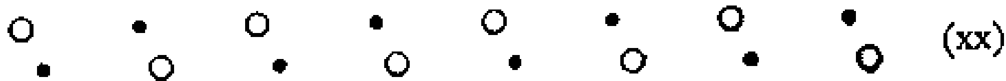

Figure 3.9 Association by shape [Wertheimer 23]

As we see, the way we draw patterns can have a direct correspondence with the effectiveness of learning. Very often, patterns are difficult to interpret because of the mismatch between our unconscious perception and the diagrams or drawings. Not surprisingly, after having started the Gestalt school, Max Wertheimer and other Gestalt theoreticians became interested in the phenomenon of learning. Köhler and Koffka formulated several ideas that directly impinge on learning theories. One such idea is the concept of "Prägnanz" and the concept of transposition. Prägnanz means that we complete patterns and scenes on the mind, to make them fit our expectations and models. Transposition means that we tend to apply successful strategies that worked for one problem to the solution of another problem.

Wertheimer always stressed that when we look, we are looking for *meaning*. We try to understand a scene and the mind settles on that interpretation which makes sense, according to our level of knowledge [Hergenhahn 01]. Learning can be best accomplished when the visual world has a definite meaning

Wertheimer distinguished between "arbitrary" and "meaningful" learning. When we learn a telephone number of a friend there is no meaning we can associate with this number. We will forget it unless we use it repeatedly. Rote memorization is the only way of remembering arbitrary learning elements. But when what we learn is inserted as part of a whole, when it makes sense and can be integrated, then we will remember it and will be able to use it productively.

Thus, according to Wertheimer, learning is more effective, if the elements of a problem can be arranged and rearranged in meaningful ways. There is an exploratory phase to learning, which is called the presolution period. This is the research phase in which the problem is looked at from every possible angle. After a phase of rest, when the mental images organize, suddenly we recognize the solution to a problem or understand the method being used. According to Köhler, this is a kind of qualitative jump which occurs in an intuitive way.

What lessons could we extract from this Gestalt approach to learning for algorithmic animation?

First, I think that we can squarely assign algorithmic animation to that exploratory period which precedes meaningful understanding. An algorithmic animation system makes it possible to look at the problem and the methods from many different angles. The algorithmic animation system should make it possible for the viewer

to try different data sets and so observe the individual steps of an algorithm under different conditions.

Secondly, the visualization of an algorithm should convey meaning. An animation of a sorting algorithm, for example, in which we only see bars change position very fast, does not really help to learn or memorize the learned algorithm. The visualization should provide meaning. The essence of the algorithm should be understandable from the visual representation. An example could be visualizing Quicksort: if we only see the numbers, represented as bars of different lengths, change place rapidly, we will not gain much from the visualization. If, however, we can show that the numbers being moved are being filtered by a pivot element, and the pivot element is being placed in the middle of the two filtered sets, then the visual impression will become lasting and the viewer will have learnt the algorithm possibly for her whole life.

We can therefore think of algorithmic animation as an instrument for *pre-exploratory exploration* and for the *discovery of meaning* in the mechanics of algorithms. Most existing algorithmic animation systems have stressed the first aspect: they allow the viewer to play with many different kinds of data. However, not many existing animation systems have taken care of producing meaningful animations, that is, animations which engrave the meaning of an algorithm on the student's mind in such a way that will not be forgotten. This second aspect, discovery of meaning, is the most difficult. It cannot be automated because teaching meaning is not something computers are good at - meaning is beyond their reach.

With algorithmic animations we strive for *cognitive fidelity*: what we see is what we mean. Achieving this presupposes that we have the adequate visual metaphors so that the data structures of the mind are brought in correspondence with the data structures of the algorithms.

3.2 Pictorial representation of abstract concepts

The main problem in algorithmic animation is that we want to visualize *abstract concepts*, that is, data structures or sequences of steps. Fleischer and Kucera, in their analysis of algorithmic animation for teaching, explain that an algorithmic animation always starts from a model: a certain representation is needed in order to visualize the operations [Fleischer 02]. In computer science, some standard visualizations for the main data structures have been developed over time. They are *visual metaphors* [Jeffery 98]. A tree, for example, is represented using circles from which other circles are hanging. An array is represented by a sequence of square fields inside a rectangle, etc. Some other models have been taken from statistics. Here we look first at the conventions that have been developed over time within the computer science community to deal with primitive and composite data types. Such conventions should be also used in animations [Brown 98c].

3.2.1 Cognition and graphical conventions

Psychologists have studied the cognitive origins of pictorial representations used by humans. Their results show that there is an intrinsic cognitive component involved in the interpretation of graphical representations, as claimed by the Gestalt psychologist, but also that cultural and social conventions play a significant role [Tversky 95].

One important convention, for example, is *schematization*. Human diagrams, pictograms, are a simplification of reality. From the study of the way children represent quantities and from the study of diverse cultures, researchers have noticed that while the first writing systems represented objects directly, later on most of them evolved abstract representations. This is the principle of schematization in action. It operates within a cultural group as an evolutionary process. Sumerian cuneiform writing, for example, became more abstract over the course of several centuries. Schematization is then cemented by education and social interaction. Venn diagrams, another example, were first used by Euler and are now the standard way of introducing sets to pupils in schools [Harel 88].

A second cultural convention is the *direction of axes*. In *xy*-diagrams, the larger values in the *y*-direction are represented upwards from the origin, larger values in the *x*-direction, from left to right. The association of big and dominant with the upward direction seems to be strongly rooted in language and culture. The horizontal direction in which objects and numbers are drawn, on the other hand, is mainly given by the direction in which we write [Tversky 91]. In most Western nations this direction is from left to right. Nowadays the canonical direction of the axes in Cartesian coordinate systems is used in graphs in newspapers and books without having any explanation of the convention used.

Linear arrangements of symbols seem to be another convention deeply rooted in our cultures. Writing is done linearly, in the horizontal or vertical direction, but seldom in any other direction. Linear arrangements are visual cues understood in a natural way by humans. Gestalt theorists would say that the common fate of the same direction of display naturally makes us categorize such linear arrangements as composite objects.

Flow of time in diagrams follows a peculiar cultural convention: in graphs time is almost always assigned to the horizontal axis and time flows from left to right. This is the convention used in musical scores, where the higher notes are also located higher up in the vertical direction. The convention for time flow is so universal, that a person watching a cursor moving from right to left in a time axis will certainly associate this with time going backwards. It seems that when humans scan a painting, this is done from left to right. Most artists paint scenes taking this left to right scanning direction into account.

Whatever conventions are used, what we try to do is to make visual perception help us understand an algorithm. Representing numerical data by the length of boxes helps (as in sorting algorithms), or also representing data by letters (in the BALSA visualization of Heapsort, for example). The right schematization depends on the algorithm being discussed [Fleischer 02]. Such considerations are important because psychologists have found that clever rearrangements of the same information in different kinds of graphs can dramatically affect what the viewer understands [Shah 99].

3.2.2 Modern graphical conventions in computer science

There is by now a long tradition of representing computer science concepts in a certain "canonical" way. Popular books made certain diagrams popular [Martin 85], other representations are the "most natural" and are used by many lecturers when explaining algorithms. Few investigations about the intuitive emergence of pictorial conventions have been done; one of them is [Douglas 94]. The authors of this report let students try to visualize algorithms using scissors and colored papers, as well as XTANGO. The visualizations that emerged represented arrays by linear orderings of boxes, with and without computer. Saliency was represented by changes in color in both cases. However, magnitude was represented by numbers, in the hands-on visualization, and by box length in the computer approach. Viewers of the computer visualization needed some time to understand that the magnitude was being represented by the length of a line.

An interesting experiment is described in [Lohse 94]. Sixty drawings were classified by a group of persons, based on similarity, and eleven specific classes of diagrams emerged: graphs, tables, graphical tables, time charts, networks, structure diagrams, process diagrams, maps, cartograms (maps with added information), icons, and photographs. The authors think that visual representations are "data structures for expressing knowledge", and therefore, it should not be surprising that some standard visualization structures have emerged over time. Moreover, visual representations can facilitate problem solving and discovery, by taking advantage of intuitive "perceptual inference" mechanisms.

What this means for us, is that we should be aware in a software visualization system of the existing standard visualizations of data structures which have emerged in the literature, and also that we should take care of matching the visualization to intuition [Bertin 83]. Getting the graphical representation "right" is the first step towards a useful animation [Douglas 95].

In this section we pass review to some standard graphical representations for data structures and then propose one of our own for heaps, which we think maximizes the information transported to the viewer.

Iconography of Arrays

It is interesting to review the type of data representations for arrays that have been developed over the years. A good starting point are spreadsheet programs, which condense the experience gained by statisticians and graphic specialists over decades. In what follows, we will deal with the sequence of numbers 2, 9, 4, 7, 6, 5 and will illustrate this sequence in different ways.

The direct representation consists in writing this numbers in a box with place for each one of them, as shown below.

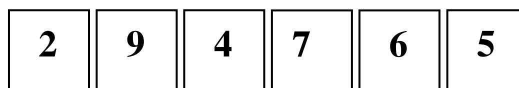

Figure 3.10 Visualizing an array

Contrast this with the main types of abstract representations used for numbers and arrays of numbers in computer graphics such as vertical bar graphs (2D or 3D), horizontal bar graphs (2D or 3D), lines of connected points, or scatter plots. Figure 3.11 shows four examples of bar diagrams (three vertical, one horizontal). The conventions used for the direction of the axes are the usual ones. The background of the display has been shaded to provide good contrast with the data (in color).

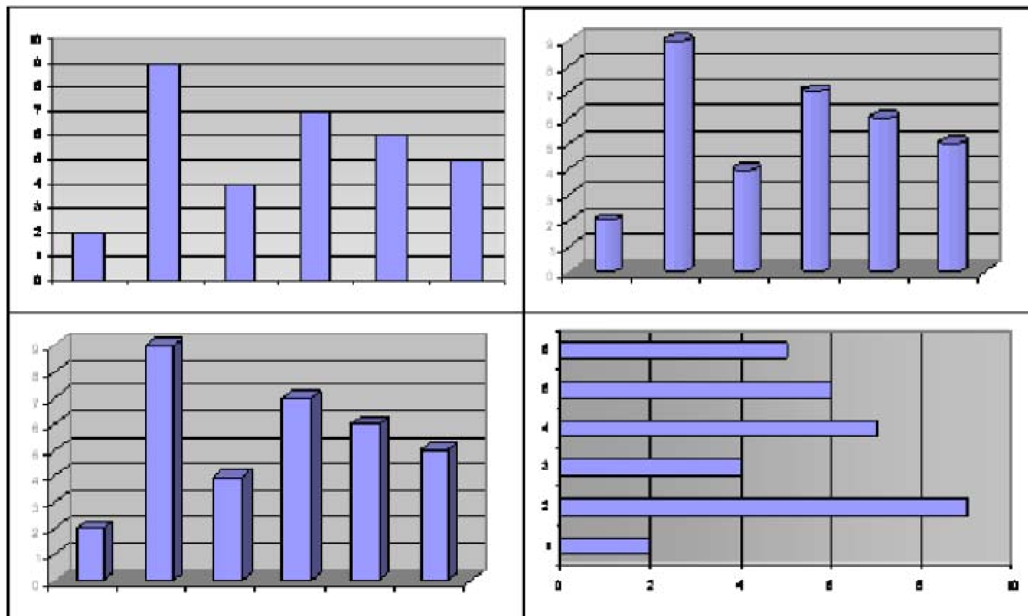

Figure 3.11 visual representations for arrays (in Excel)

A scatterplot is almost useless for visualizing small data sets, as can be seen below, left. Larger data sets, though, are easier to be visualized with a scatterplot. Patterns that are not obvious with a bar plot, become salient when seen as scatterplot.

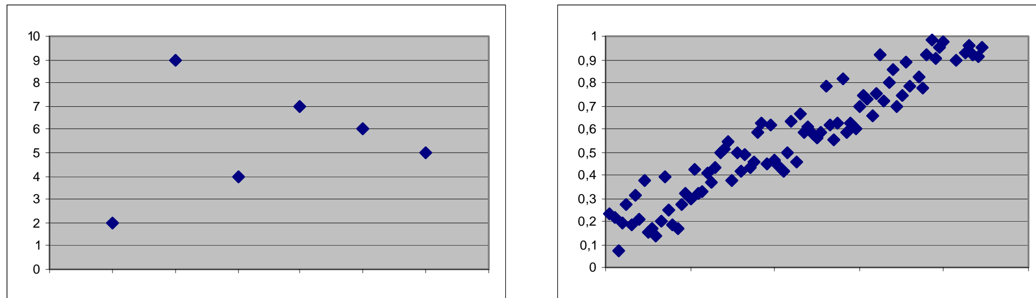

Figure 3.12 A scatterplot is more effective for large arrays

Scatterplots and bar plots have been traditionally used for representing arrays, especially for sorting algorithms. The memory location of the data is shown on the horizontal axis, the value of the data on the vertical axis.

A few animators use a novel circular representation for numbers, as shown in Figure 3.13, which is an animation of the Quicksort algorithm. Here, the magnitude of the data is represented by numbers and the subsets being sorted by their distance from the center.

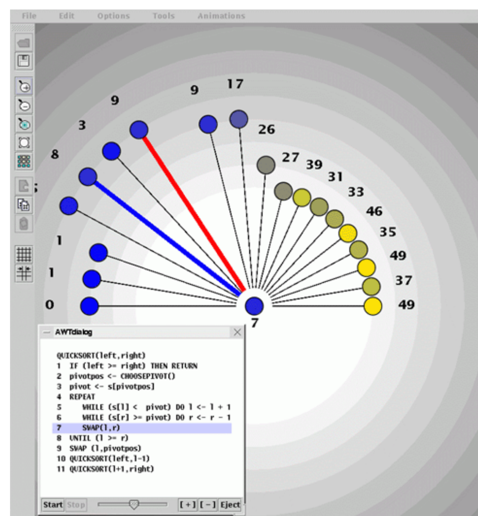

Figure 3.13 A circular visual display of an array

This visualization is artistically appealing but has some shortcomings. The concept of order is not very well represented by arranging numbers around a circle, the magnitude of the numbers has to be read, and only logical groupings are clearly discernible.

Iconography of Lists

The canonical representation for linked lists is to use boxes with two compartments: one compartment for the data, the second for a pointer to the next element in the list. A root pointer provides a reference to the first element in the list. Figure 3.14 shows a graphical representation of a list in LISP. The head of the list is called CAR, the tail the CDR. The CAR points to the numerical data, the CDR to the next element or to NIL.

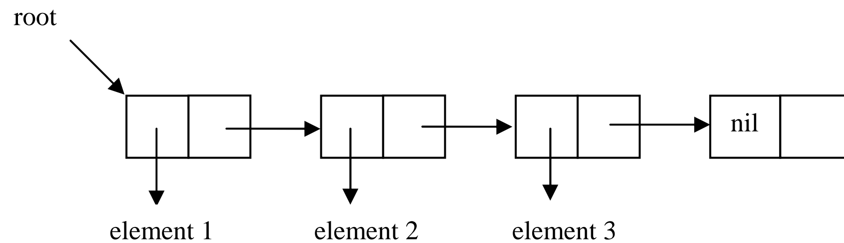

Figure 3.14 Representing a list with boxes and pointers

The illustration shown in Figure 3.15 is a rather more artistic rendering of a list, but the iconography is essentially the same.

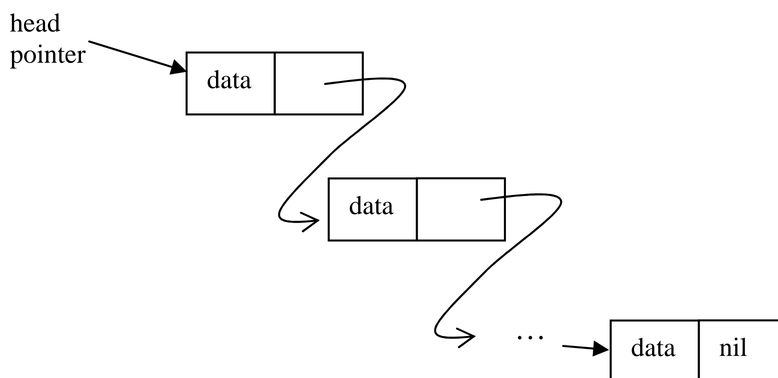

Figure 3.15 A list element consists of data and a pointer to the next element

A textual representation of a list is written using parentheses and separators, as for example in Haskell, where a list is represented by the text [2,9,4,7,6,5]. A textual representation can be used when the internal representation (illustrated with boxes) is not relevant for understanding an algorithm.

Some animations represent the nodes of a list arranging them in a circle. This allows viewing more nodes, but the approach is not very popular.

Iconography of Trees

After lists, trees are one of the most useful data structures in searching, sorting, and for many other applications. In the last years many different types of graphical representations for trees have been developed, mainly for dealing with large amounts of data, as present in the Web.

Our first representation is the canonical: nodes hanging from nodes Figure 3.16. Computer trees grow from top to bottom, reflecting its origins in teaching: we know where a tree can start to grow, but we do not know at the beginning how large the tree will eventually become. Therefore, when using a blackboard the only alternative is to let the tree grow down.

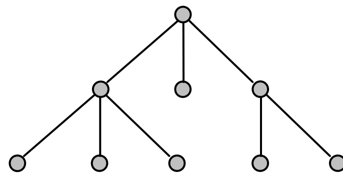

Figure 3.16 A canonical computer science tree

The second representation is very similar to the representation for lists (Figure 3.17). A binary search tree of integer values (where each node has two children), has two pointers to other nodes and contains data. A root pointer provides an entry into the tree. A nil pointer is represented by a crossed box.

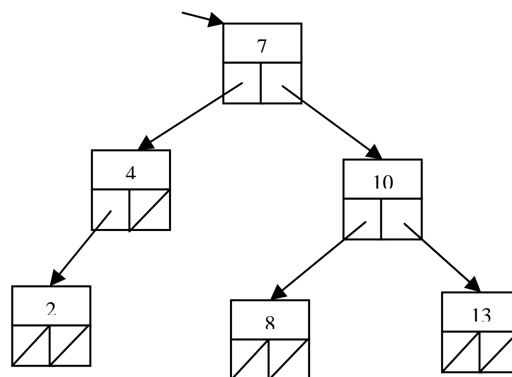

Figure 3.17 A computer science tree with visible pointers to subtrees

The next representation is similar, but the NIL pointers have been represented here by a pointer to ground, to represent an undefined pointer.

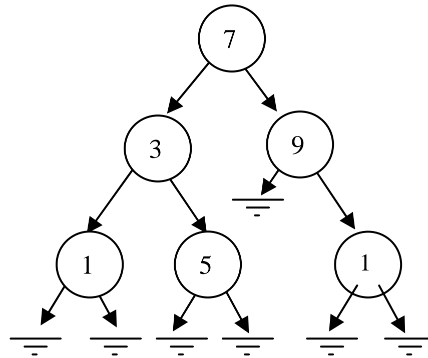

Figure 3.18 computer science tree with visible pointers. Nil is represented by ground

Sometimes we are interested in the relative distance of each node to the root, or the local distance between nodes in the tree, which can be visualized with circular trees, as the ones shown below.

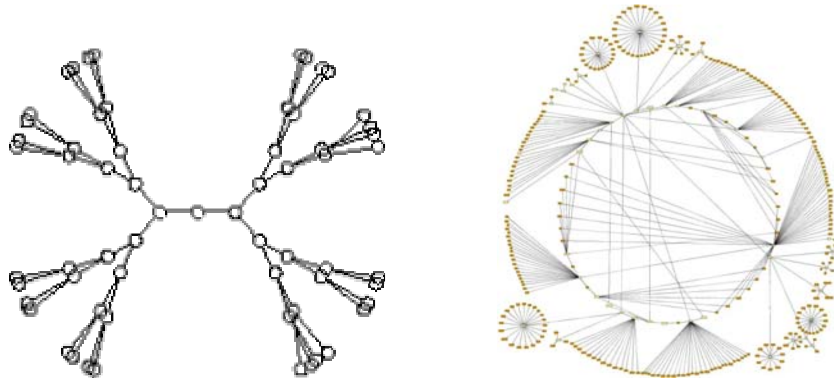

Figure 3.19 Circular trees

Hyperbolic trees are a variation of circular trees which useful when the amount of leaves in the tree is large and is growing [Lamping 94]. The tree gets compressed towards the periphery and the focus of attention lies in the middle, in the root.

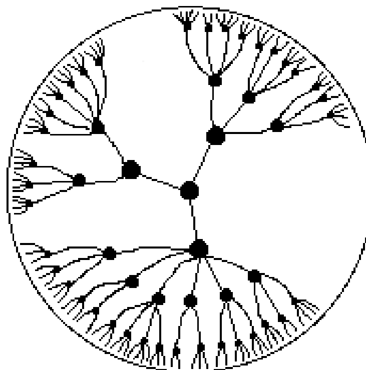

Figure 3.20 Hyperbolic trees, useful for showing complex data trees

Trees can also be shown growing from left to right, as in Figure 3.21. This representation is useful when describing temporal hierarchies of events. As we explained before, there is a natural tendency to associate the x-axis with temporal flow. A tree growing from left to right can be interpreted naturally as a developing process or a sequence of decisions, which must happen one after the other.

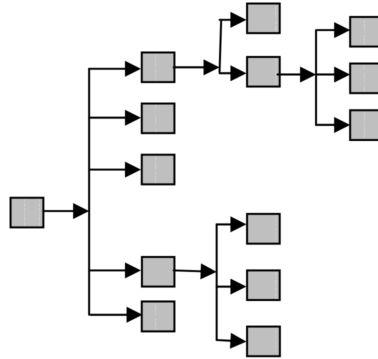

Figure 3.21 A tree of temporal events, or decisions

Cone trees provide a less ordered depiction of the children of nodes. The children hang from the node without any left to right order. Cone trees are used frequently in data representations involving 3D graphics. Figure 3.22 shows a 3D cone tree and its 2D projection, in gray (`www.conal.net/tbag/screenshots.html`).

.

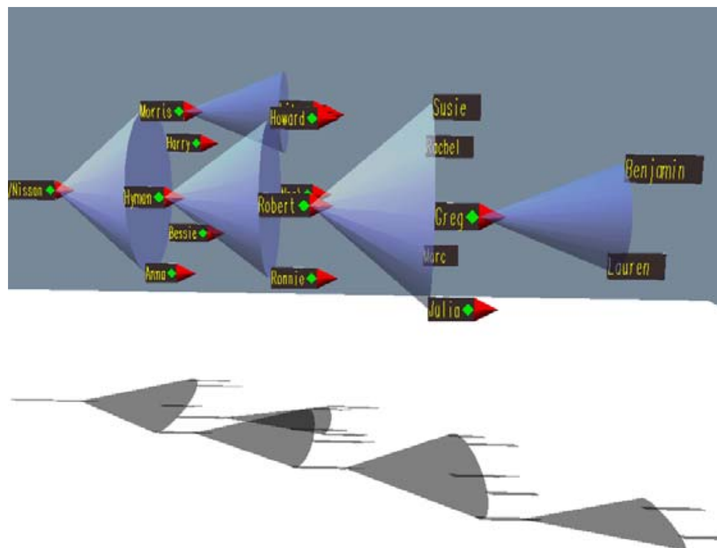

Figure 3.22 Cone trees can be used to visualize information in 3D or 2D

A parsing tree is one of the few examples of trees that grow upwards from the root. It is clear why: in parse trees the words in a sentence are the leaves of the

tree, we know exactly where the leaves are, and in this case it is easier to draw the tree from the leaves to the root.

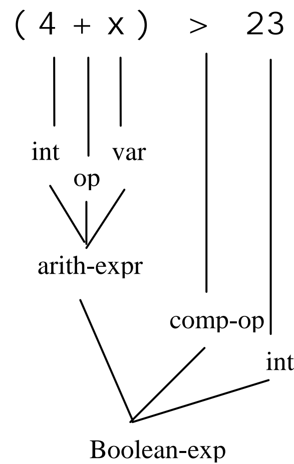

Figure 3.23 A parse tree

Notice that in parse trees the subtrees form part of a category, the root is the final linguistic category that has been found.

Iconography of Phylogenetic trees

In phylogenetic trees, the leaves are the individual species; the nodes bifurcations represent the evolution process. The distance of the bifurcations from other bifurcations, or of the leaves from nodes, has a meaning in this class of trees. The distance is proportional to the time elapsed between bifurcations. Time runs here from bottom to top, the original ancestor is at the root (taken from the Web site about dog evolution at `www.nhm.org/exhibitions/dogs/evolution/`).

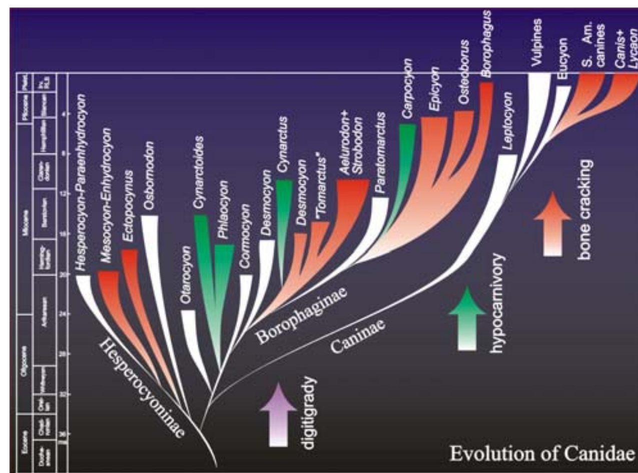

Figure 3.24 A phylogenetic tree

Iconography of tree maps

A novel visualization method for trees, the tree map, was proposed by Shneiderman and Johnson [Johnson 91]. Tree maps represent the whole tree by a square. The first level of the tree (the sons of the root) is represented by vertical divisions of the square, where each rectangle's area is proportional to the value of each node. Each rectangle is subdivided horizontally according to the number of sons of each node, and so on. Figure 3.25 shows an example of a tree and its representation as a tree map. The main advantage of tree maps is that they allow using the whole drawing surface to represent the tree. Tree maps have been used to visualize the allocation of data storage in hard disks and its evolution over time.

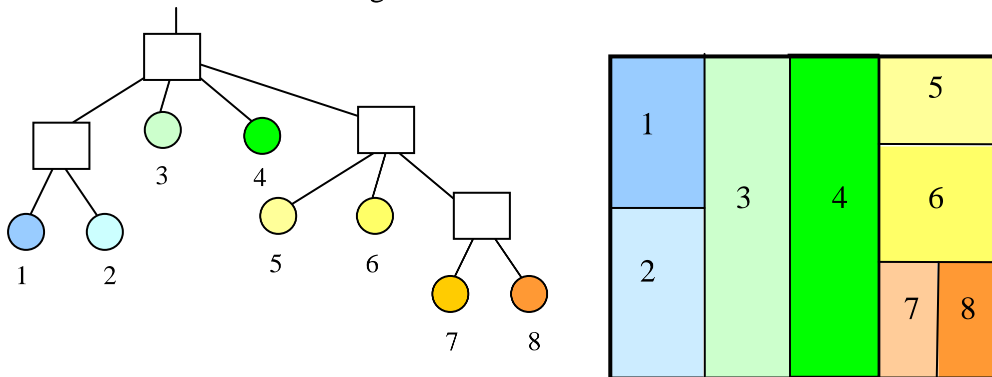

Figure 3.25 Tree maps represent information as areas in rectangles

Iconography of Heaps

The most popular representation for Heaps is just to represent a heap as a tree, as shown in the animation in Figure 3.26 (`www.informatik.uni-trier.de/.../PROJECTS/SortAnim/`). However, a heap is not a classical tree, since no pointers to the children have to be stored in parent nodes. The position of the children is implicit in a data array. A heap is better thought of as an array with an overlaid virtual tree structure.

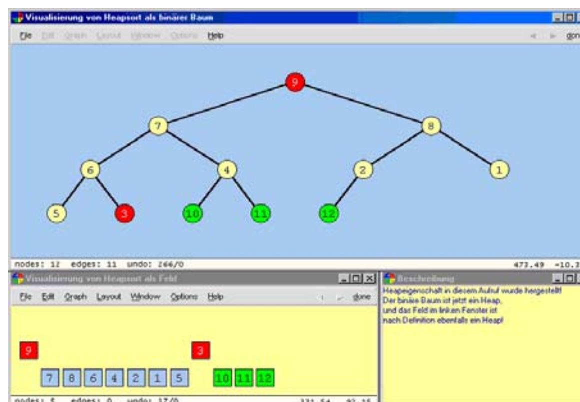

Figure 3.26 A heap, represented as a tree

A search of the available graphical representations in the Internet did not lead to finding any alternative representation for heaps.

Iconography of Graphs

Graphs are usually represented by circular or polygonal nodes and lines as edges. Many methods for drawing planar graphs have been developed, some of which can be used for drawing esthetically pleasing graphs. If the nodes are arranged randomly, the result is something like the graph to the left. If the position is selected carefully, a kind of "organic view" of the graph emerges.

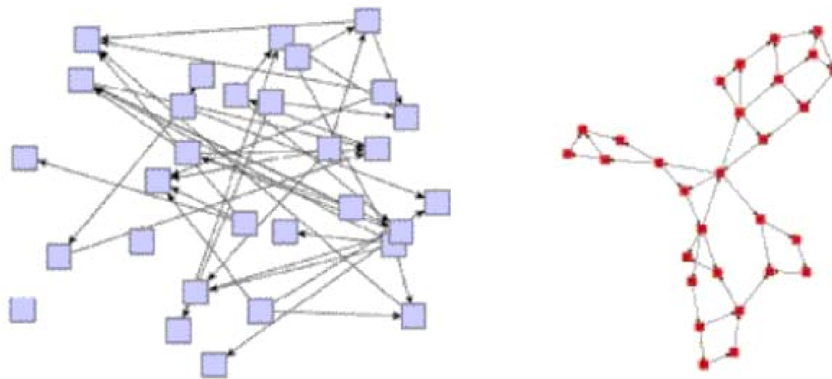

Figure 3.27 Representations for graphs

Iconography of Stacks

Stacks can be represented as objects laid on top of each other. Figure 3.28 below has been taken from a data structures learning unit at the Web site located at `ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/stacks.html`. It shows that objects can be pushed into the stack and can be popped in order, and that there is only one "entrance" to the data structure, namely from the top. This is the way a stack of dishes in a restaurant works.

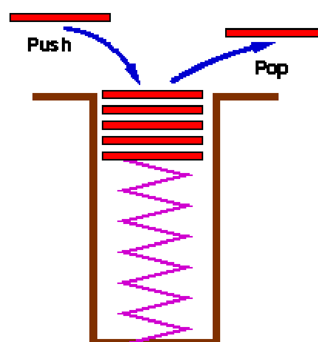

Figure 3.28 stack represented as a receptacle with a single entry

Iconography of Queues

Queues can be represented phenomenologically, that is, in regard to their function and appearance (left). In regard to their implementation, a queue is often represented as a ring. This shows the maximum capacity of the queue, the front of the queue and the rear, and provides an immediate idea of how to implement the queue operations efficiently.

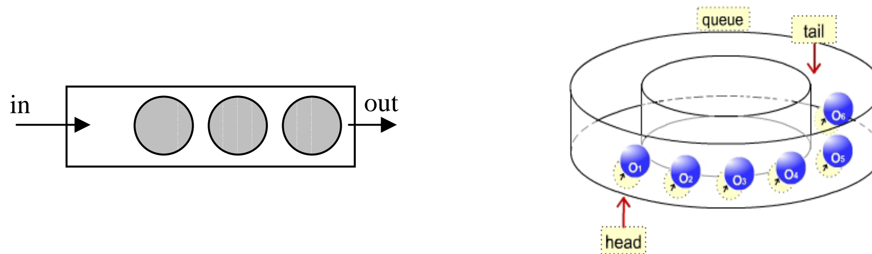

Figure 3.29 A queue, represented as a tube, or as a ring with two pointers

Fisheye views

A relatively new technique for visualizing large data structures, such as long lists, trees or complex graphs, consists in mapping the objects to the screen, but using a local magnification that concentrates the attention on one element and its neighbors [Furnas 86, Sarkar 92]. The kind of mapping used is locally hyperbolic; it resembles a magnifying glass for looking at parts of structures. Figure 3.30 below shows a graph to the left, and a fisheye view of the same graph, with the attention centered in one node, the largest that can be seen. Local operations on this node and its neighbors are now easier to see than in the original graph.

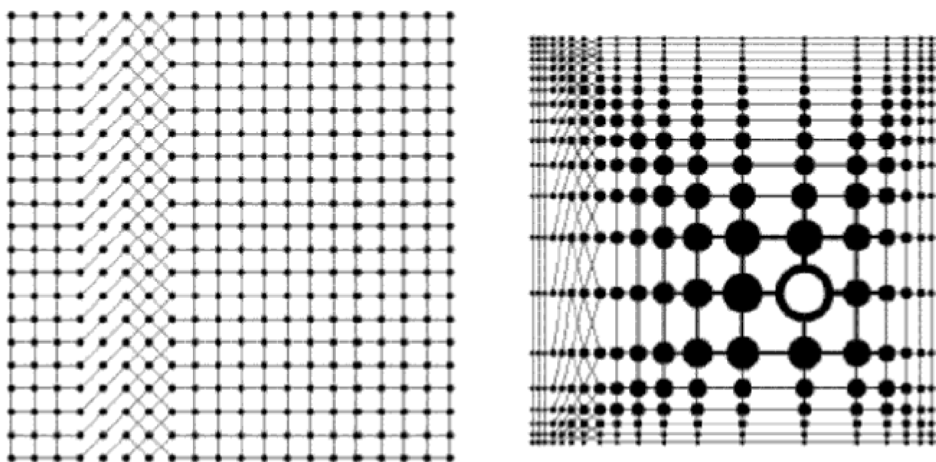

Figure 3.30 A fisheye view (right) centered on a node of the graph to the left [Sarkar 92]

The fisheye view is also useful when dealing with lists or trees, as exemplified by Figure 3.31 below.

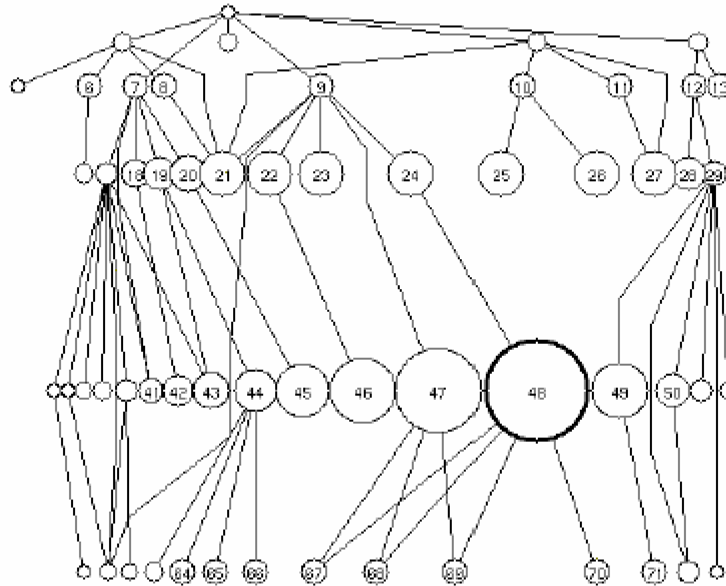

Figure 3.31 Fisheye view of a tree [Sarkar 92]

The fish-eye view can be also applied to code, so that when examining a program, the line being read appears larger and distant lines smaller. In the case of code, it could be possible to have two different magnification points, so that when following a program two distant portions of the code are readable. Computer code tends to have non-local connections, differently from graphs or trees.

```
cross.icn

end

procedure gen(i)
  local tmp, up, down
  tmp := i / 2
  if (i % 2) = 1 then
    tmp +:= 1
  suspend tmp
  up := tmp
  down := tmp
  while (up < i) do {
    suspend up +:= 1
```

Figure 3.32 Fisheye view of computer code

3.3 A new graphical representation for heaps

The purpose of this section is to describe a new way of visualizing heaps. This new visualization technique combines the logical organization of the heap with its physical realization. A simpler predecessor of this idea was used by Brown, but in the context of 3D visualizations for the Zeus system [Brown 91, 93]. His method is not as comprehensive as the one I will explain here.

A disadvantage of traditional visualizations of heaps is that they are simply drawn as binary trees. The fact that data is allocated sequentially in an array and that the father-children relationship is only a virtual one, mediated by the position in the array, is obscured by the tree representation. We can say that a heap is *logically* a binary tree but physically an array. The logical tree view of a heap is overlaid on top of the physical array implementation. Some visualizations of heaps try to deal with this conceptual difficulty by showing the tree representation and the array representation at the same time. This solution is not totally satisfactory. In the Heapsort algorithm, for example, the viewer has to follow changes both in the logical tree and in the array, in order to understand the algorithm. However, we know from section 3.1.1 that no two important actions should occur at the same time.

The solution to this difficulty is to combine the logical with the physical view in a single picture. We start with
Figure 3.33, which shows a heap as a logical tree, but on the left side shows the allocation of the individual elements in an array. The root of the tree is the first element in the array. The two children of the root are allocated in the array elements 2 and 3. The two children of the first root children are stored in the array elements 4 and 5, and so on.

Figure 3.33 shows with the horizontal lines, where each node of the logical view is stored in the array. The numbers to the left are the indices of the array locations (from 1 to 7).

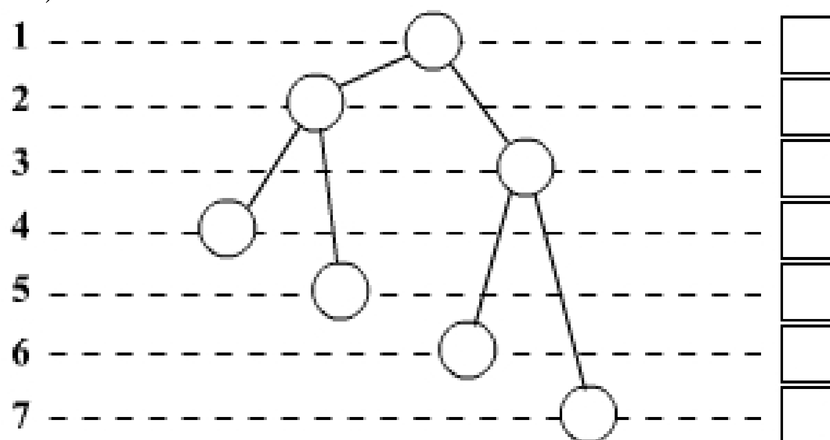

Figure 3.33 Correspondence of the nodes of a heap to the elements of an array

From this visualization it is immediately obvious how to compute the index of a parent node for a node with index *k*: just divide the index by 2 (integer division). It is also obvious that the two children of the node with index *p* are 2*p* and 2*p*+1.

The representation above just shows the allocation of the elements of the heap, but not the heap property. In a max heap, a parent node is not smaller than any of the children nodes. This difficulty is solved with the next visualization. Here the index is shown on the left and the contents of each node horizontally at the bottom. If we only store numbers from 1 to 7 in the heap (for the sake of this example), then the vertical line shows the number stored in each node. Colors have been also used to stress the correspondence of the logical with the physical view. The first node, for example, contains a seven. The children of the first node contain a 6 and 4, respectively, and so on.

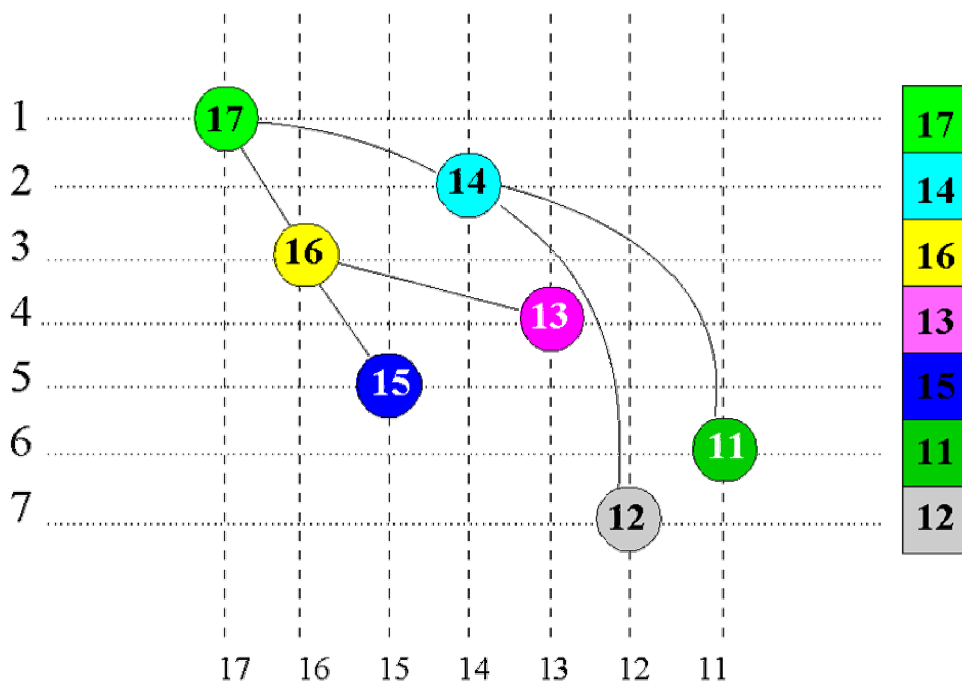

Figure 3.34 Example of a well formed max-heap

The visualization shows that the logical tree leans from the upper left towards the lower right.

The visualization is most useful when the heap property is disrupted. Consider the image below. Since the children nodes of the root do not lay to the right (that is, they are not all smaller or equal), it is obvious that the heap property has been broken and has to be restored by appropriate transformations.

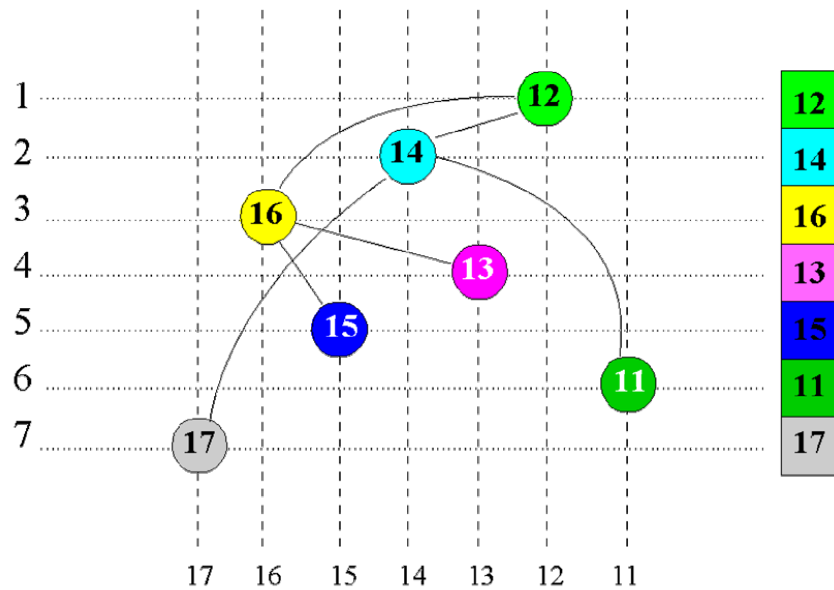

Figure 3.35 A perturbed heap

The visualization can now be improved, if we rotate the diagram by 45 degrees counterclockwise. Figure 3.36 below shows the elements of a heap (left picture) and the heap connections between the elements (right picture). The elements are numbers between 1 and 100, selected randomly with substitution. The index of the array runs from 1 to 100, from the upper right to the lower corner. The value of an element of the heap increases from 1 (lower corner) to 100 (middle left corner).

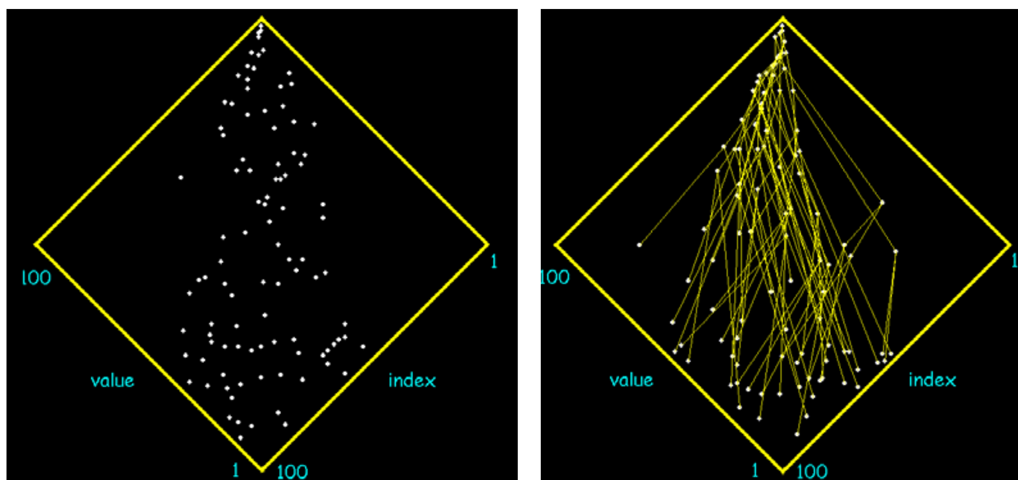

Figure 3.36 Visualization of a heap of 100 numbers. The picture to the right shows the virtual connections between the elements of the heap.

The beauty of this representation is that already the cloud of points gives a very good idea of the virtual tree implicit in a heap array. Value and index of a number can be deduced by projecting normal lines to the sides of the enclosing square. As can be seen on the right picture, a heap admits crossings of the edges of the virtual

tree. A binary search tree does not admit such crossings. The difference between a binary search tree and a heap becomes then visible in this representation.

It is also interesting to look at the kind of heaps produced by Heapsort when the input array is already sorted. The code in [Cormen 90] produces the heap shown in Figure 3.37. It is a kind of regular structure. The pattern shows that the numbers are almost sorted in the heap.

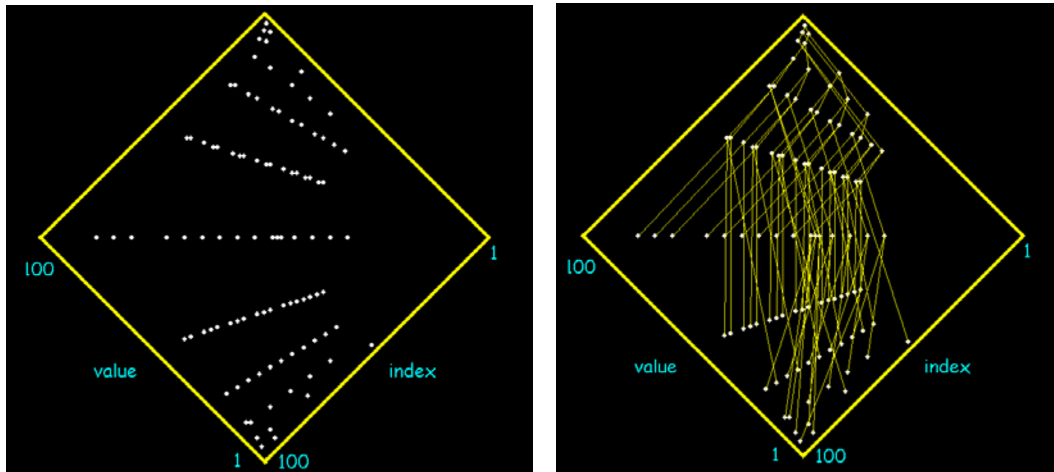

Figure 3.37 The heap when the numbers in the array are already sorted

The diagrams in Figure 3.36 and Figure 3.37 were produced by Chalk Animator, the algorithmic animation system to be discussed in Chapters 4 and 5. The labels for the axis were typed a posteriori. In section 4.12.3 a complete run of the Heapsort algorithm has been illustrated.

3.4 Types of algorithmic animations

From our survey of algorithmic animation systems in Chapter 2, it should be clear that there are many different ways to animate code:

- *animation by handcrafting*, that is, producing the animation by hand, as is done in PowerPoint animations;
- *animation by debugging*, where a development system just produces pictorial representations of variables and data structures and their changes;
- *animation by declaration*, where the interesting state transitions are declared;
- *animation by instrumentation*, where interesting events are flagged.

In all cases, a very important strategy that makes students understand the animation and remember it better, is to try to show not only the mechanics of an algorithm, but also the properties of the data structures it generates. “Show invari-

ants", this seems to be a very important piece of advice for algorithmic animators [Fleischer 02]. By invariants of algorithms, authors mean that important properties of the data should be made visible.

One example of this dictum was used above when my visualization of heaps was introduced: the visualization shows always that the largest element is at the top of the heap and shows how the pattern distorts when this property does not hold. This invariant of heaps is evident in the visualization.

Another example is given by [Fleischer 02]. When discussing binary trees, it is possible to represent the tree structure. If an additional bars diagram is shown below the tree, the ascending order of the data stored in the leaves of the tree becomes visible. When data is being searched, it is easy to see how the search process should proceed. In Figure 3.38 the magnitude of the number being sought is given by the horizontal line; current position in the search is given by the vertical line. Obviously, the search should be continued to the right. This is a very good example of the usefulness of showing the invariant of the data structure together with its logical layout.

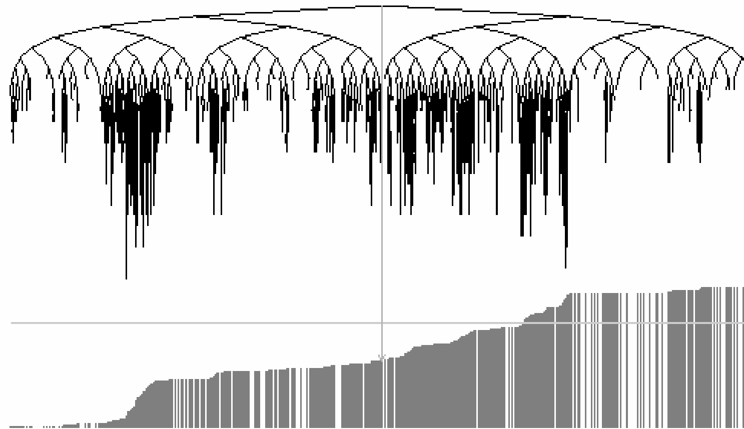

Figure 3.38 A binary search tree and the magnitude of the numbers at each leaf (represented by each bar below a node) [Fleischer 02]

Providing overlaid views of the data structure is another way of providing empirical "visual" proofs that the algorithms work as desired.

Another possibility when dealing with an animation is to increase the size of the problem, in order to see the increase in the number of operations. If the increase in computing size is shown graphically, the viewer can then get an idea of the complexity of the algorithm. The "Sorting Out Sorting" film is a good example of how to gradually increase the size of the problem set.

Another possibility is to show the development of the algorithm's steps over time. The Fourier transform has been traditionally visualized in this way, using a butterfly network. Quicksort can be visualized as a tree of recursive calls, in which each

call produces two recursive calls (in general). If the algorithmic animation system provides multiple views, the best animation will try to present all this information on different views, connected in some pictorial way.

The relationship between changes in the data structures and the pseudocode of the algorithm can be shown in a view that helps students to better understand. This is also the approach that I have followed in section 3.6.2.

3.5 The Esthetics of Algorithmic Animation

In this section we bring life to the abstract principles discussed above. We have implemented some animations of sorting algorithms using the Macromedia Flash animation engine[3]. The animations were designed by hand, trying to make them as compelling and esthetic as possible. Such simulations could be considered the esthetical upper bound of what we can achieve with automatic animation tools.

3.5.1 Resorting Out Sorting

Sorting algorithms are the work horse of the algorithmic animation community. There are more animations for sorting algorithms than for any other topic. However, I want to show here alternative ways to visualize the different sorting methods. The following animations have been done in novel ways trying to provide a memorable visual experience. The six algorithms are: Bubble Sort, Insertion Sort, Quicksort, Heapsort, Merge Sort, and Counting Sort.

Bubble Sort

Bubble sort is a very simple sorting method: numbers are compared pair by pair, starting from the front. When a pair is not in the correct order, the two numbers are swapped. The algorithm finishes when a pass cannot find any non-sorted pair. Bubble sort was initially called "exchange sort" and also "jump sort". The first printed reference to "bubble sort" is from Iverson in 1962 [Astrachan 03]. The name itself is a visualization of the behavior of the larger numbers in the array, which "bubble up" sequentially to the last positions in the array.

The animation is based in this bubble metaphor. The horizontal axis corresponds to the magnitude of the numbers stored in the array. The vertical axis shows the entries in the array. When the algorithm starts, the first two numbers in the array

[3] My Student assistant Chris Bauer produced some of the handcrafted Flash-animations discussed in this section.

are compared and are exchanged, if necessary. When a number is stored in another array location, it changes its position along the vertical direction (from one horizontal dashed line to another).

Figure 3.39 shows, on the left, the original distribution of numbers in the array. Each number has been displaced to the right to better see its movement later. Fig. 3.37 shows, on the right, how the numbers have been displaced by bubble sort. Each “bubble” leaves a yellow trace.

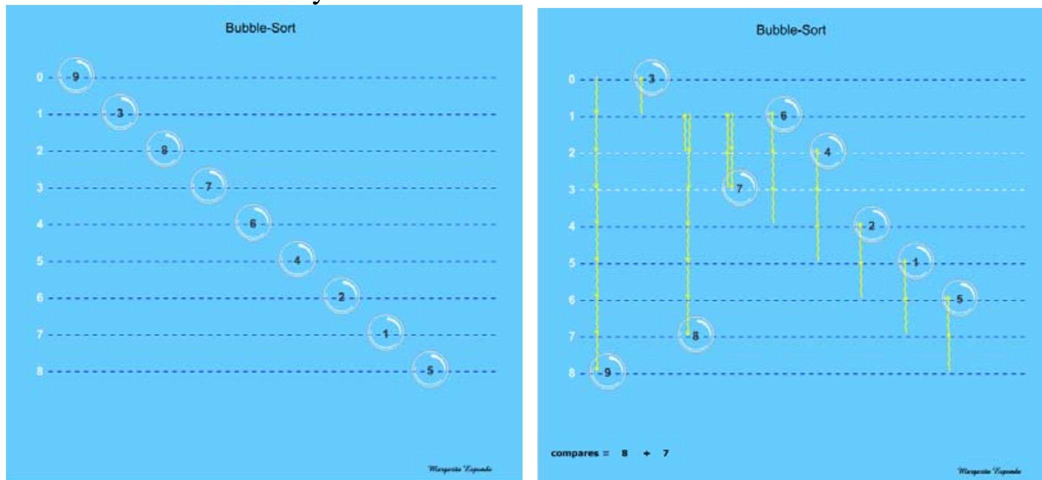

Figure 3.39 Initiation of the bubble sort animation (left), the state after some exchanges (right).

The wiggled lined describing the total displacement of a number across the array being sorted is contained in an overlay. The wiggled line is a trace of the history of the number position. As we can see, some numbers go straight to their position, others go up, and then down again. It is easy to see why. In bubble sort, a number *A* goes up in the array until all numbers larger than *A* have come through. The *A* starts falling, until all numbers smaller than *A* have bubbled up forward in the array.

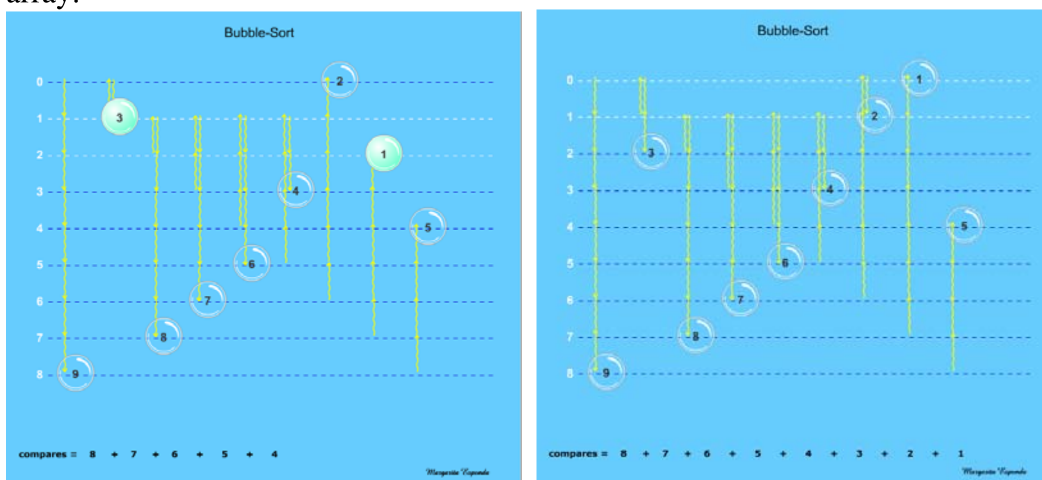

Figure 3.40 The exchange of the numbers 3 and 1 is highlighted (left), the state after number 1 has reached its final position (right)

Figure 3.40 shows how the numbers selected for comparison are highlighted. On the left, the numbers 3 and 1 have been selected (index 1 and 2 in the array). They have changed color.

Figure 3.41 finally shows the end of the sorting process: the sorted numbers move to the right and are arranged in a column. Their vertical position is now easier to compare.

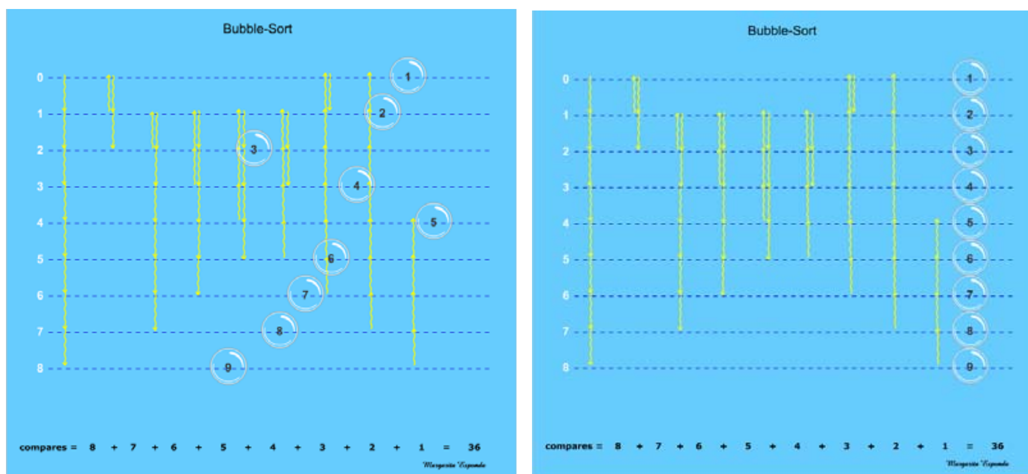

Figure 3.41 The sorted numbers are moved to the right to show their array positions

The animation is reversible. It can be stopped at anytime and can be run backwards, a single or more steps at a time.

The overlay used for the trace of the data movement (the yellow arrows) is a nice illustration of properties of algorithms which become evident once a visualization for data operations is provided. Another overlay could show the number of compare operations used during a run of the algorithm. Then it can be visually verified that the algorithm is of order n^2. Overlays will be discussed in Section 6.9.

Insertion Sort

The next algorithm implemented with Flash is insertion sort, an algorithm better than bubble sort, but still of quadratic complexity. Insertion sort scans an array from beginning to end. If an element *A* is out of order (the element before it is larger), the element is copied to an auxiliary variable and the beginning of the array is examined to make place for *A*, at the correct position, by shifting all elements larger than A one place downward, until the position previously occupied by *A* is filled. A is copied to the "hole" now free in the array.

Figure 3.42 shows the second element of the array (in yellow) taken out and "pushing" the beginning of the array to make place at the first position. A bull-dozer pushes all elements larger than the yellow one, one position to the right.

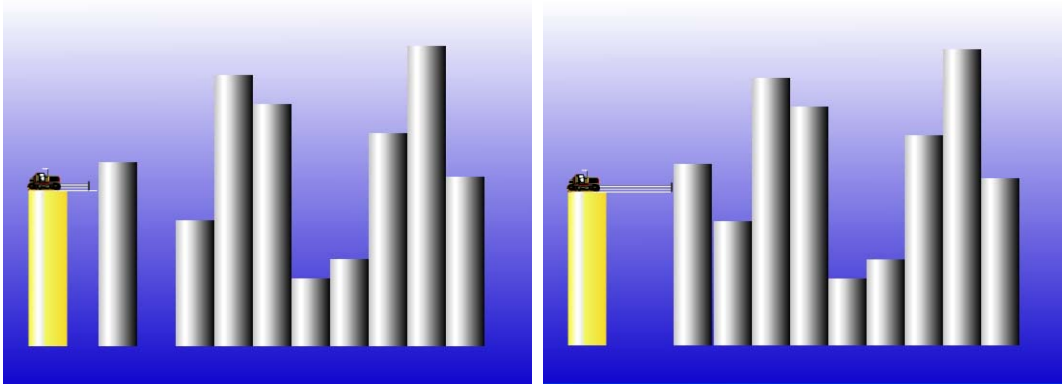

Figure 3.42 A step of insertion sort. The bulldozer pushes the element at index position 1 back, making place for the yellow element.

Figure 3.43 shows insertion sort some steps later. Again, an element has been taken out of the array, the elements larger than it are pushed by the tractor arm, which only reaches those larger than the yellow one, until a new position for the yellow element has been opened.

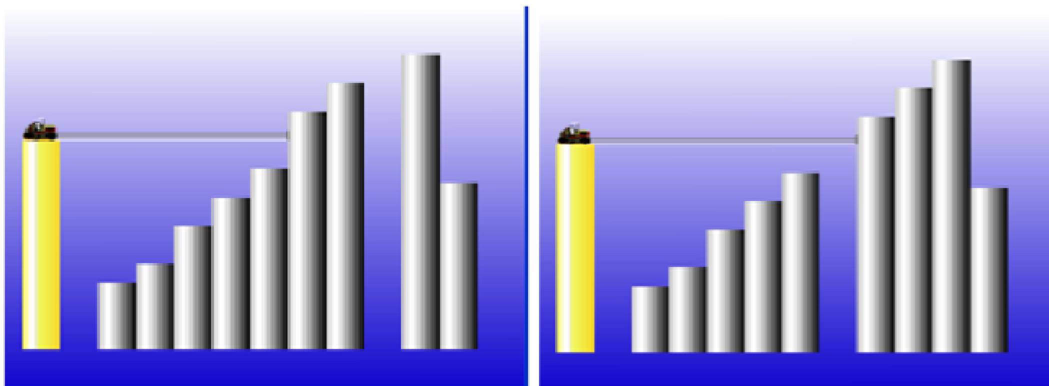

Figure 3.43 Some steps later. The yellow element will be placed in the new empty position.

The animation is more attractive at larger magnifications, because the bulldozer has been drawn in full detail.

Quicksort

Quicksort is a popular sorting algorithm, very easy to code, especially in its functional version. Quicksort was invented in 1962 by Tony Hoare and by now it is cast in stone in the computer science curriculum [Hoare 62].

Quicksort is a recursive algorithm. It takes an element in an array, the so called pivot, and decomposes the original problem into two smaller problems: in order to sort the list, first sort the list of elements smaller than or equal to the pivot, and then sort the list of elements greater than the pivot. Finally, concatenate both sorted lists, with the pivot in the middle.

Figure 3.44 shows the start of the animation with an unsorted array. The numbers are represented as bars of different heights. The first element in the array is selected as the pivot (blue). The pivot shines a “laser” across the array, with a height equal to its own, and any element which is crossed by the laser changes color to orange. This operation partitions the array into the elements smaller (gray) and those greater than or equal to the pivot (orange). The smaller elements are moved to the front, the larger to the back of the array as shown in Figure 3.45. Then the algorithm continues recursively with the two subarrays.

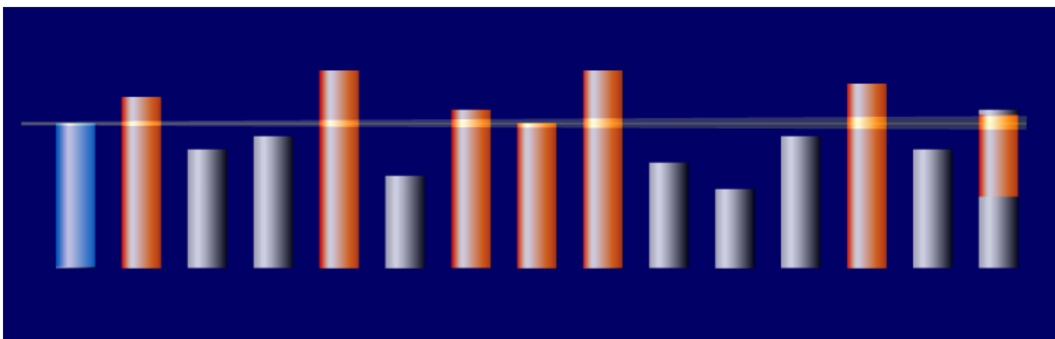

Figure 3.44 The pivot, in blue, shines a laser through all other bars. Those greater than or equal to the pivot are colored orange.

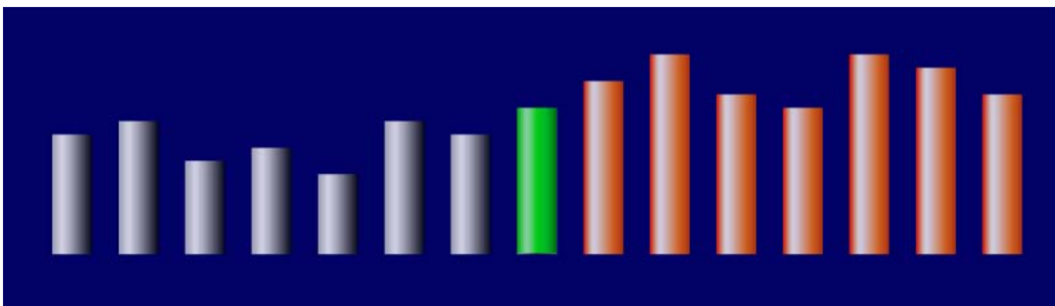

Figure 3.45 The pivot is displaced to the middle, the smaller elements to the left of the pivot, and the larger to the right.

The successive screenshots show the progress of the algorithm and how the first half of the array is processed before continuing sorting the second half.

I produced for Resorting Out Sorting a second handcrafted animation of Quicksort, shown in Figure 3.48. Here the size of the rabbits represents the numbers being sorted and the comparison is made with rulers moving along the array. This animation explains in-place-sorting with more detail. Figure 3.48 illustrates the

partition procedure used in [Cormen 90] as it can be programmed in C. The pseudocode of Quicksort is shown in a small frame, which is animated concurrently with the displacement of the array objects. The animation was used for a course for high-school students at the FU Berlin in 2003.

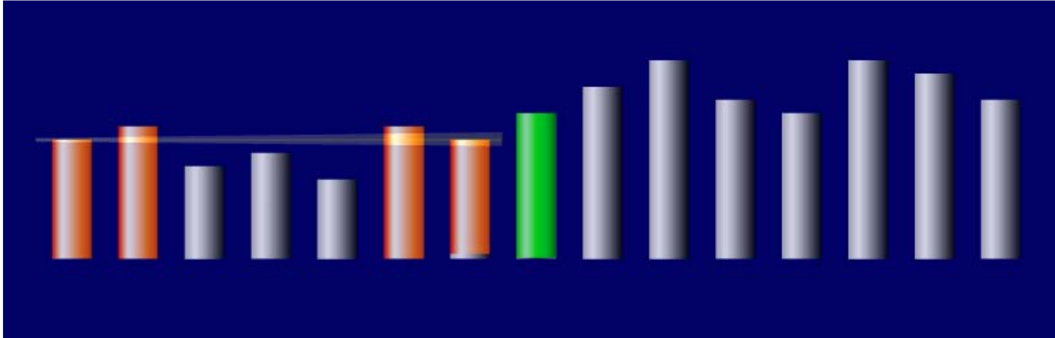

Figure 3.46 The left half of the array is sorted recursively using Quicksort.

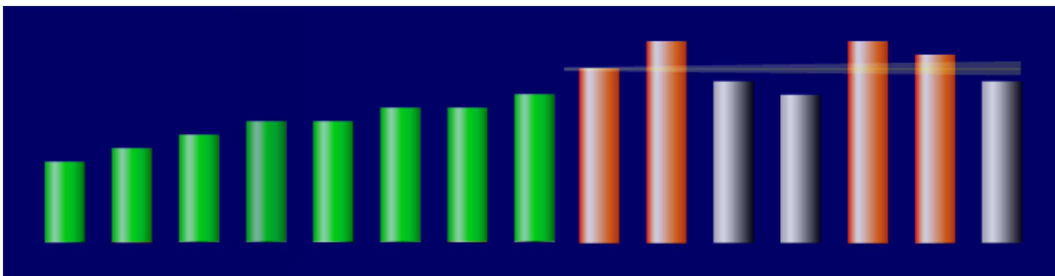

Figure 3.47 The left half of the array has been sorted, the right half is now being sorted.

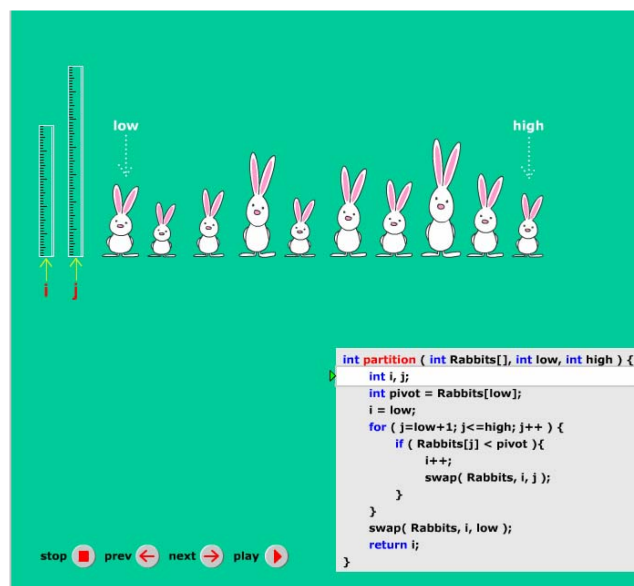

Figure 3.48 Start of the Quicksort algorithm. The pointers *i* and *j* have not yet been initialized. The code is just starting (highlighted in the code window)

Heapsort

Heapsort was invented by Joseph Williams in 1964 [Williams 64]. The array to be sorted is used to build a heap structure, which is a virtual tree superimposed on the array. Once the heap has been built, the first element of the array is taken out of the heap and is exchanged with the last element in the array, which is now considered to be “out” of the heap. The new smaller array is heapified, and the algorithm continues recursively with the smaller array.

Figure 3.49 shows the beginning of Heapsort. First the array (shown on the right, vertically) must be heapified, that is, it must be transformed into a heap. The index for the array position is shown on the left, as a number going down from 1 to 9. In the Figure (left image) the number 3 must be swapped twice in order to build the heap. The animation does this and the final result is shown in Figure 3.49 (right side).

When the heap has been built, the largest element is at the top (the number 9). This is taken out of the heap and positioned at the end of the array. The array is heapified again, and the procedure continues recursively as shown in Figure 3.50.

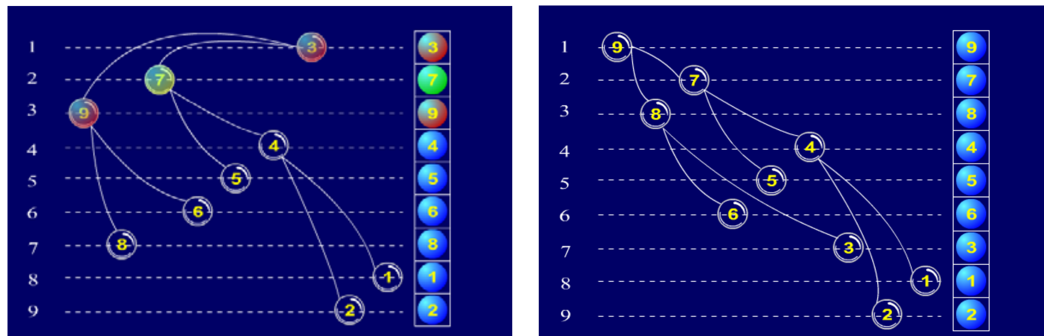

Figure 3.49 The heap is being built (left). The first element in the array (3) does not preserve the heap property and must be swapped. Final heap after two swaps (right). The representation of the heap is the one described in Section 3.4.

The animations show the heap as a virtual tree. Each node’s correspondence to an element in the array is shown using dashed lines. The exchange of nodes is shown both in the virtual tree, as well as in the array. This gives a feeling for the movement of the numbers in the array itself.

With our representation for the heaps, it is very easy to detect visually when an array does not fulfill the max-heap property, as in Figure 3.49, left image.

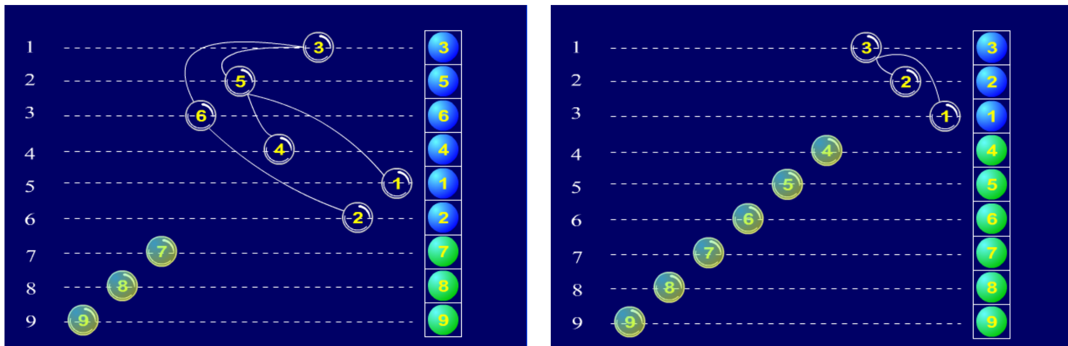

Figure 3.50 The first three numbers have been sorted at the end of the array, they are shown in green (left image). The right image shows six elements already sorted and the final heap of three elements that is yet to be sorted.

Merge Sort

According to Knuth, merge sort is one of the oldest computer sorting algorithms. Its invention is attributed, as so many other things, to John von Neumann. [Knuth 73]. Merge sort divides an array of numbers into two halves. Each half is sorted recursively, and then the two sorted halves are merged into a sorted array.

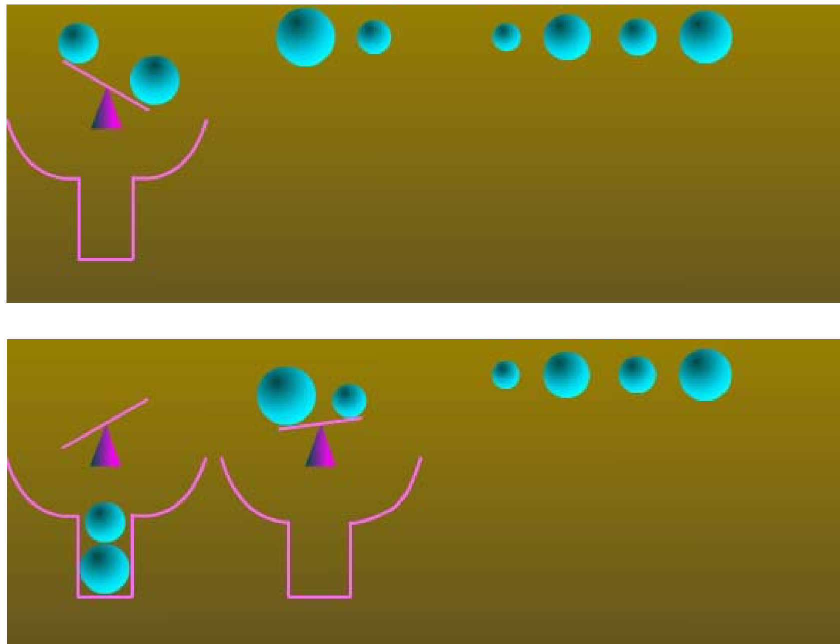

Figure 3.51 Sorting with merge sort. The first two elements of the first subarray are merged using a balance. The larger element falls to the bottom first.

The subdivision process of the array leads to recursive calls until one-element arrays have to be merged. Then, two-element arrays are merged, and so on. The

algorithm can actually start with the merge process, assuming a previous “virtual” subdivision of the original array.

The animations show the successive merge operations working on an unordered array. The screenshot in Figure 3.51 shows the start of the animation. The array (eight elements) has been subdivided into eight virtual single element arrays. The first two subarrays are merged by comparing the two elements using a winged balance. The balance tilts and lets the “heavier” element fall. The second element falls on top. This mechanical visualization helps to understand how the numbers are ordered at the merging points.

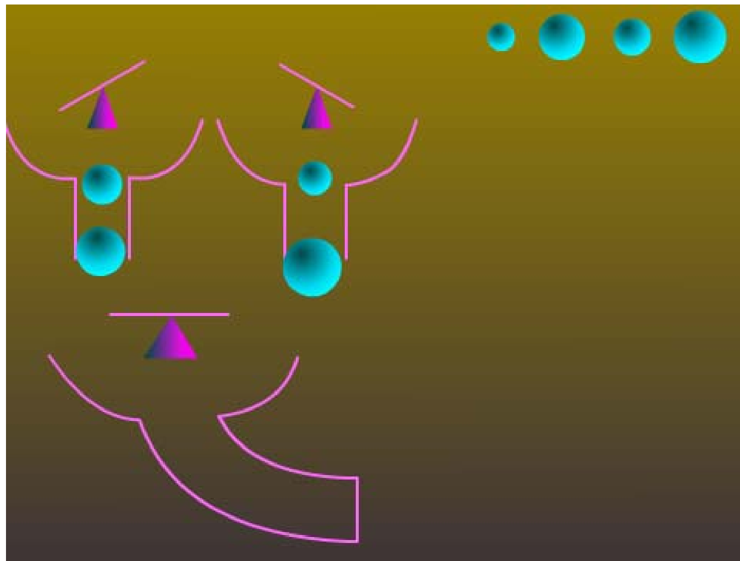

Figure 3.52 The first two-element subarrays are merged next.

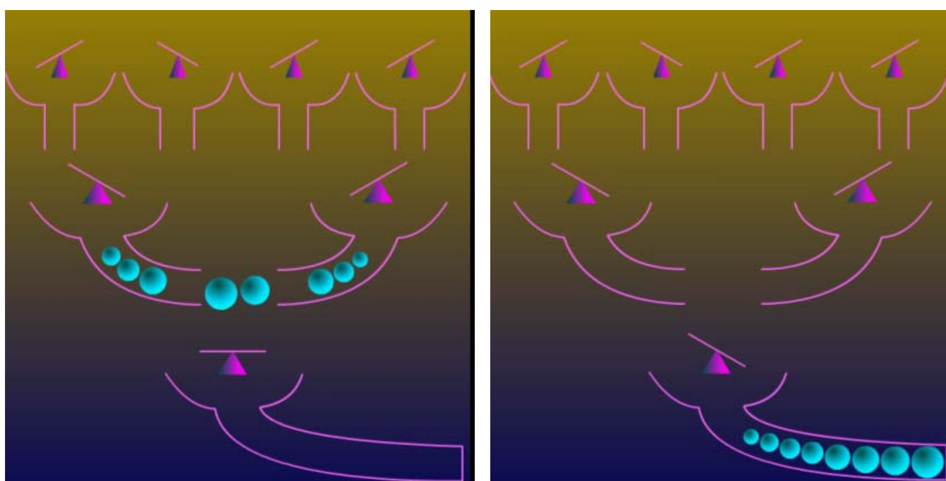

Figure 3.53 Last merge operation of two four-element subarrays (left). Sorted array (right). The numerical weight increases proportionally to the size of each sphere.

Figure 3.52 shows the continuation of the simulation. After the first four one-element subarrays, have been merged into two two-element subarrays, these two are merged again using another balance. The elements fall in order, the heavier first, and we obtain a four element sorted subarray.

Figure 3.53 shows the last merge operation and the result of merge sort. The numbers (represented by the size of the spheres) are now ordered from smaller to largest, in the left-right direction.

The animation provides the user with some level of control. The simulation can be stopped, can be run forward and backwards. Also, the user can zoom-in or zoom-out of the simulation, if desired. This functionality is provided by the Flash player hosting the animation.

Counting Sort

Counting sort is an algorithm which can sort numbers without using pairwise comparison. In the example below, the digits 0 to 5 are sorted. Six boxes are used to accumulate the number of zeroes, ones, twos, threes, fours, and fives present in the input. First the boxes are used for counting the number of times each digit appearance as shown in Figure 3.54. The process has just started (left screenshot). The result of the counting process is shown in the right screenshot. As can be seen, there are two zeroes, three ones, etc.

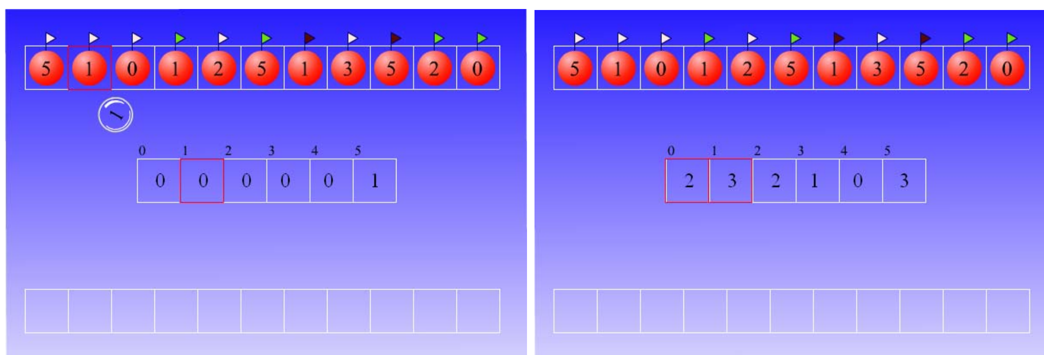

Figure 3.54 Counting sort starting. First the frequency of each digit is counted in the central array of boxes.

Using the information from the count, the position of each subgroup of digits in the sorted array can be computed by summing the contents of the central boxes from left to right. The ones, for example, finish at position 5 because there are two zeroes and three ones. The screenshot in Figure 3.55 shows the accumulation process (left screenshot) and how this accumulated values are indices to the sorted array (virtual pointers, shown in white).

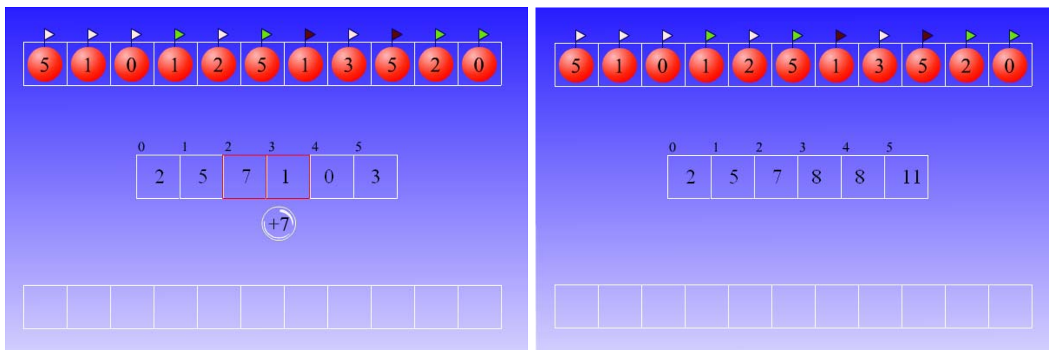

Figure 3.55 The accumulated values (left) left of the bubble can be used as indices (virtual pointers) to the position of the "tail" of each subgroup of digits.

Figure 3.56 shows the final pass through the unsorted array. The numbers are copied, one after the other, from the unsorted to the sorted array. The numbers are copied from the back of the array to the front. Each time a digit is copied to its final position (following the virtual pointer), the index is advanced one position to the front of the array (i.e., it is decreased by one). Figure 3.56 (right) shows the final sorted array.

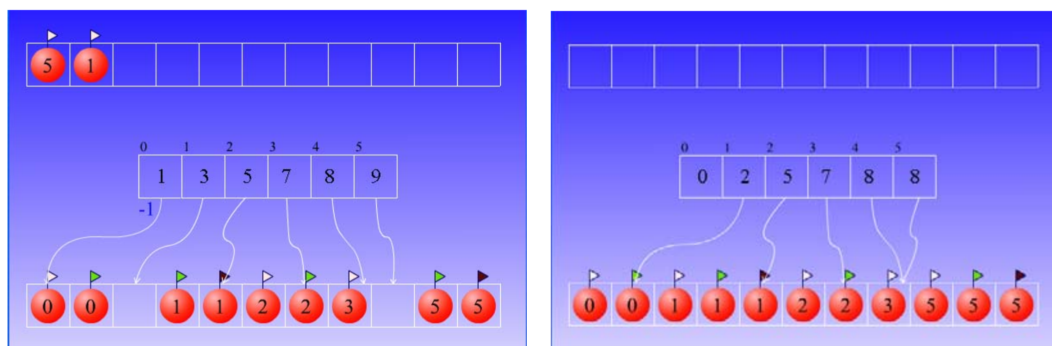

Figure 3.56 The unsorted array being copied to the sorted array (left). The sorted array (right).

In this animation the small flags attached to the digits represent additional information associated with them. Counting sort is "stable", that is, the order of each subgroup of digits is not affected. In the animation, the flags colors go from light to dark, to illustrate that the original order within the subgroup of zeroes, ones, etc. is preserved. This means that numbers with several digits can be sorted by going through each digit, from right to left, and applying counting sort to each digit position. This is the Radix Sort algorithm. The order-preserving property of Counting Sort is essential for Radix Sort.

This concludes our presentation of some handcrafted algorithmic animations. They represent, esthetically, a big improvement over those animations normally found in the internet. These and other sorting algorithms, such as Shellsort and radix sort, were animated also, but with the techniques discussed in Chapter 6. Shellsort was invented by Donald Shell in 1959. It is the most efficient of the

popular n^2 algorithms. Radix sort is indeed an old sorting algorithm. It was used in Hollerith punched card machines even before computers were born [Knuth 73].

3.5.2 Data structures and Algorithms Learning Unit

A complete unit describing some data structures for Java programmers was produced by me within the framework of the national project “Virtuelle Fachhochschule”.[4] The unit was integrated in a comprehensive programming course developed by several colleges in Germany.

The data structures and algorithms unit consists of textual explanations, code and illustrations of data structures and operations performed on them. The textual and graphical explanation resembles a good illustrated book. The user navigates clicking on hyperlinks, from a table of contents, or browsing locally from one page to the next. Figure 3.57 shows the JavaScript navigation elements included on each page. The user can go forward, backward, can start from the first page, go to the table of contents, or print the page.

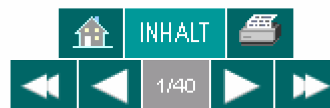

Figure 3.57 The control console of the learning unit.

The unit discusses the following data structures: stacks, queues, lists, double linked lists, queues as double linked lists, and dynamic data types. For each structure an iconographic representation is provided. The operations on each structure are defined textually and with code. Animations placed on several pages of the unit, allow the student to start the code and single-step through it, seeing at the same time the change in configuration of the data structures.

Figure 3.58 below shows an aspect of an animation that connects the intuitive notion of a stack of objects with the stack data structure.

---

[4] According to a press release from the German Ministry of Education and Research of December 2003, the number of students registered for a “Virtuelle Fachhochschule” degree increased from 170 to 600 in the second half-year of 2003 (BMBF-Aktuell Nr. 244/03 of 23rd December, 2003).

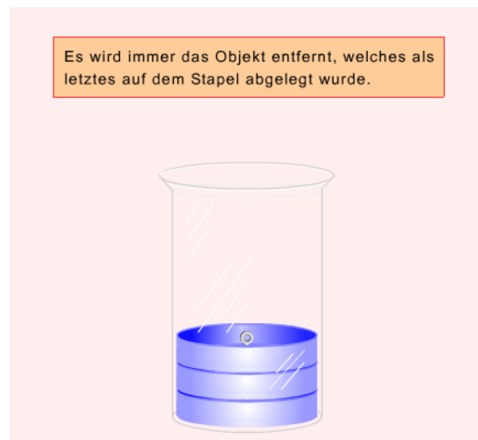

Figure 3.58 A model for a stack

Figure 3.59 shows the representation of a stack of objects. Internally, the stack contains pointers to the objects. From an abstract point of view, we can think of a *stack of objects*, as shown on the right side of Figure 3.59.

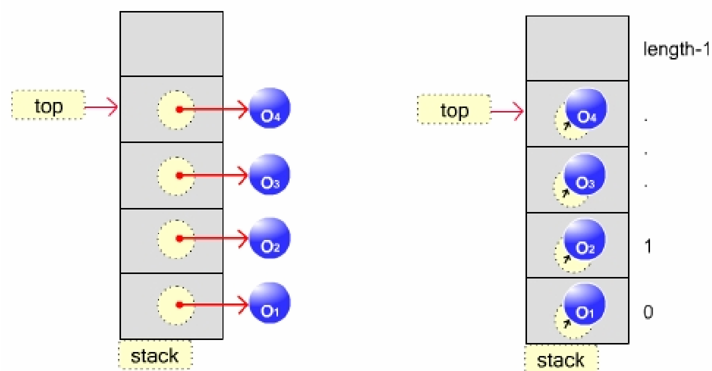

Figure 3.59 Representation of a stack of references to objects.

Figure 3.60 shows a screenshot of an animation running together with the Java code. The current command is highlighted, to draw the attention of the student to the code, and the pointer is then incremented. Notice the forward and backward arrows in the animation. They allow the viewer to proceed forward or backward when playing the animation.

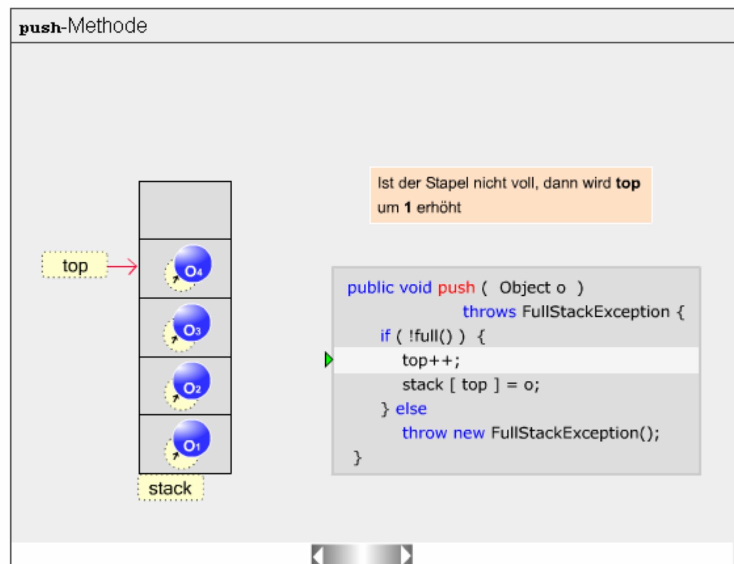

Figure 3.60 Operation of a stack running with the associated Java code.

The last screenshot shows an include operation being performed on a linked list. A small explanation pops up to describe the changes in the linked list and the command being executed.

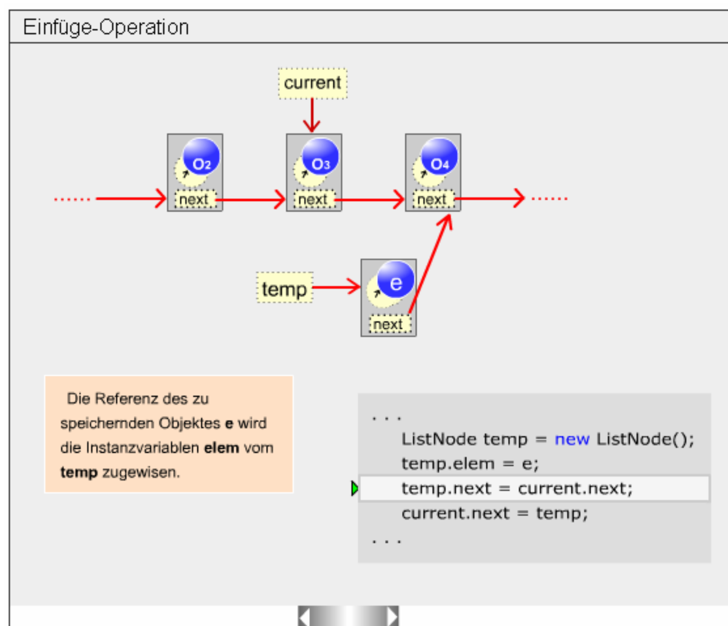

Figure 3.61 Pop-up windows explain each operation.

The sequence of screenshots below shows the execution of the three steps in a portion of code which deletes an element from a double linked list.

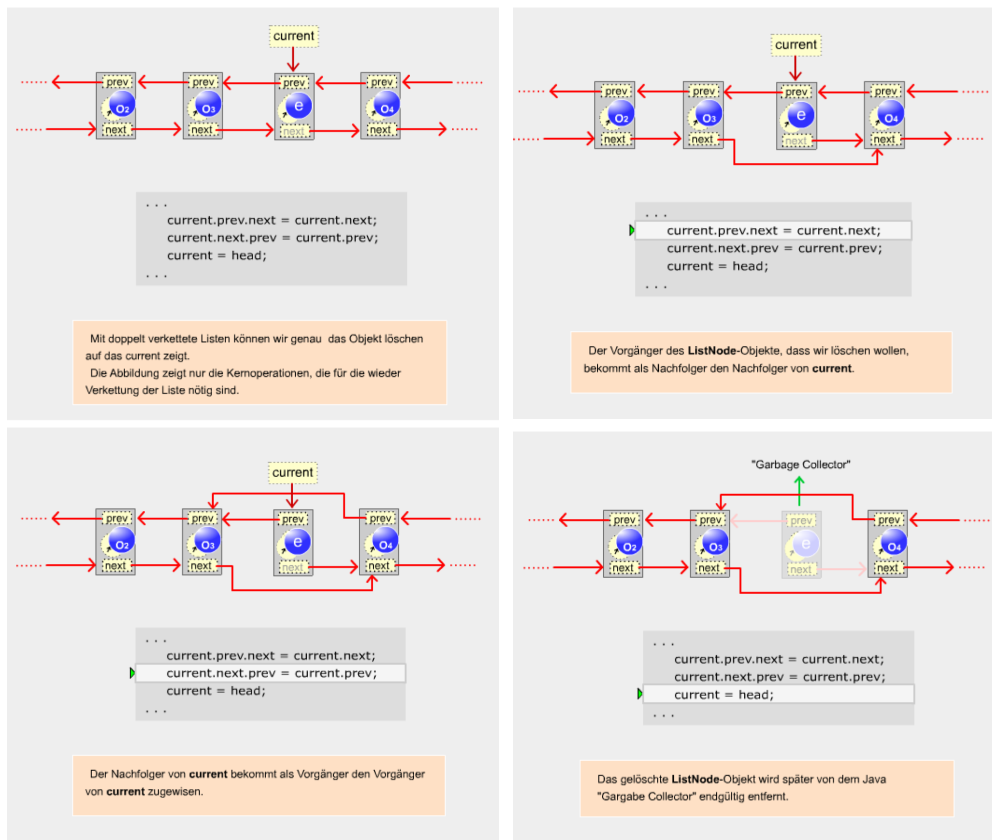

Figure 3.62 An animation sequence for a delete element operation in a double linked list.

The data structures and animations unit has been in the Web since 2002. Students have used it by now, for learning how to program linked data structures in Java. The feedback from the users of the system within the Virtuelle Fachhochschule and at Freie Universität Berlin has been very positive and new productions for other institutions will follow.

3.6 Summary and discussion

This chapter has explored several cognitive issues relating to the way persons perceive and absorb information from pictorial representations and animations. It is far from obvious how to design algorithmic animations in such a way that the amount of information is maximized with the least possible use of virtual “ink”.

In this chapter, we reviewed first the Gestalt approach to visual perception, in order to gain a better understanding of visual design issues. Visual representations can encode much information, as Blackwell found when he asked test subjects to describe in words what other test subjects had rapidly drawn. The average was

84.1 words for a diagram [Blackwell 97]. Some algorithmic animation systems are bad just because they ignore visual design issues, regardless of how computationally powerful they may be. Bad designed systems fall rapidly into oblivion and have no impact on future animation systems.

We delved into Gestalt learning theory, because I think it has something to offer when thinking about the design of algorithmic animations. There must be an adequate mapping between the perceptual intuitive visual machinery and the visual metaphors used. When visual perception and visual inference reinforce the logic of the animation, algorithms can be learned better.

We offered in this chapter an extensive review of the kind of visual metaphors that have become established in the computer science literature. Some of them have been around for decades, others are relatively new (hyperbolic trees or tree maps). A thorough knowledge of the available visual representations is a prerequisite for good algorithmic animations. My review is far from being comprehensive, and far from constituting a "semiology of animations", but is a step in this direction. Authors of algorithmic animations should be aware of the advantages and limitations of each type of visualization method for data structures.

The visualization for heaps discussed in section 3.4 is a novel way of representing the map between the logical and physical structure of a heap. It had not been used before and will be used in section 4.12.3 for the animation of Heapsort.

The two productions discussed at the end of this chapter represent an attempt to give flesh to the abstract ideas discussed here. The animation of sorting algorithms, especially, was done in novel way, trying to provide a fresh and original view of the mechanics of sorting algorithms. It is our answer to Petre's question for the reader in [Petre 98]:

> *"Why has there been no sequel to 'Sorting Out Sorting'?"*

Although Baecker did such a good job with his film, there is still room for improvement, as has been shown here.

All animations produced for this chapter are reversible and make use of overlays, that is, superposed views. Such superposed views are a way of showing the mechanics of an algorithm together with invariants and also traces of the data. In the bubble sort animation, one of the overlays shows a trace of the movement of each element across the array. The overlay can be switched on and off, as desired by the viewer. In the Quicksort animation, one overlay shows the total number of exchanges and the comparisons performed. It is easy to infer from this overlay an estimate of the efficiency of the algorithm. Overlays are an important technique, not available in any other animation system, and which has become an integral part of my own system.

The productions discussed in this chapter were implemented in Flash and Javascript, they are available over the Web, and are a proof of concept for what follows in the next chapters, namely the implementation of a common algorithmic animation system for two different platforms.