Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
Espoo 2003 TKO-A40/03

VISUAL ALGORITHM SIMULATION

Ari Korhonen

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering for public examination and debate in Auditorium TU1 at Helsinki University of Technology (Espoo, Finland) on 21st of November, 2003, at 12 noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science

Helsinki University of Technology
Laboratory of Information Processing Science
P.O.Box 5400
FIN-02015 HUT
Espoo
Finland

ISBN 951–22–6788–8
ISBN 951–22–6795–0 (PDF)
ISSN 1239–6885

Otamedia Oy
Espoo 2003

Abstract

Understanding data structures and algorithms, both of which are abstract concepts, is an integral part of software engineering and elementary computer science education. However, people usually have difficulty in understanding abstract concepts and processes such as procedural encoding of algorithms and data structures. One way to improve their understanding is to provide visualizations to make the abstract concepts more concrete.

This thesis presents the design, implementation and evaluation for the Matrix application framework that occupies a unique niche between the following two domains. In the first domain, called *algorithm animation*, abstractions of the behavior of fundamental computer program operations are visualized. In the second domain, called *algorithm simulation*, the framework for exploring and understanding algorithms and data structures is exhibited.

First, an overview and theoretical basis for the application framework is presented. Second, the different roles are defined and examined for realizing the idea of algorithm simulation. The roles considered includes users (*i.e.*, learners and instructors), visualizers (those who specify the visualizations), programmers (those who wrote the original algorithms to be visualized), and the developers (those who continue to design and implement the Matrix framework). Finally, the effectiveness of the *algorithm simulation exercises*, the main application embodied in the framework, is studied. The current tool is utilized for delivering, representing, solving, and submitting tracing exercises that can be automatically assessed, and thus provides meaningful feedback on learners performance.

Keywords: Software Visualization, Algorithm Animation, Algorithm Simulation Exercises, Automatic Assessment

Preface

This study was carried out at the Laboratory of Information Processing Science, Helsinki University of Technology. The contributions and innovations of this thesis are made in two joint research groups, *Computer Science Education Group* and *Software Visualization Group* during the years 2000 to 2003.

I am sincerely grateful for the kind support of many people. First of all, I wish to thank my supervisor, Professor Lauri Malmi, for his patience and guidance. I also thank Professor Jorma Tarhio, my second reader, for his helpful suggestions and constructive criticism, and my preliminary examiners Professor Rockford J. Ross and Accociate Professor Mordechai Ben-Ari for their valuable comments.

The members of both of the research groups are thanked for their support and involvement in my work. Special thanks go to Ville Karavirta, Pekka Mård, Jussi Nikander, Riku Saikkonen, Harri Salonen, Panu Silvasti, Kimmo Stålnacke, and Petri Tenhunen for their discussions and contributions on the Matrix framework and my thinking. I would also like to thank Professor Patrik Scheinin and B.Sc. Pertti Myllyselkä from the University of Helsinki for their guidance in the educational sciences, and Mrs. Ruth Vilmi for help with the English language.

I wish to thank Professor Eljas Soisalon-Soininen for his guidance during my postgraduate studies. In addition, I thank all my friends in HUT for their friendship and support which made completing my dissertation a positive experience.

Finally, my whole family deserves special thanks for always standing by me. I dedicate this work to them.

Otaniemi, October 2003

Ari Korhonen

Contents

Chapter 1

Introduction

Let us assume we could *visualize* a data structure and have an implemented algorithm to manipulate it. Obviously, the visualization could be used for animating the algorithm in which the visual display contains a sequence of consecutive visual snapshots of the data structure. During the execution of the algorithm, the state of the data structure changes. These states and changes can be animated for the user, one after another. In addition, the user may have some kind of control over the process, so he or she can interact with the system in order to stop and continue or even step through the animation. The user may also have the option to change the values of program variables (or at least the input data for the algorithm), watch these values at certain breakpoints, and so on. This kind of step-by-step animation and visualization of variables could be compared to *visual debugging* [85, 117] of an algorithm.

Moreover, these animation steps could be memorized in order to give the user the control of traversing the animation sequence back and forth. We call such a process *algorithm animation*. Many systems [12, 15, 18, 20, 48, 73, 75, 86, 94, 96, 100, 107] have been developed for this purpose. We will discuss these systems more closely later in this thesis. Figure 1.1 illustrates the general overview of such traditional algorithm animation systems.

Furthermore, we can allow the user to perform his or her own operations. Thus, instead of letting the algorithm execute instructions and manipulate the data structure, we can allow the user to take control over the manipulation process. The user can directly change the data structure on-the-fly through the user interface and build an algorithm animation by demonstration similar to that of Animal [100] or Dance [108, 110]. However, here the user not only has control over the visual representation (direct manipulation in Figure 1.1) but can actually change the underlying data structure in the run-time environment. Thus, the user invokes actual operations by manipulating

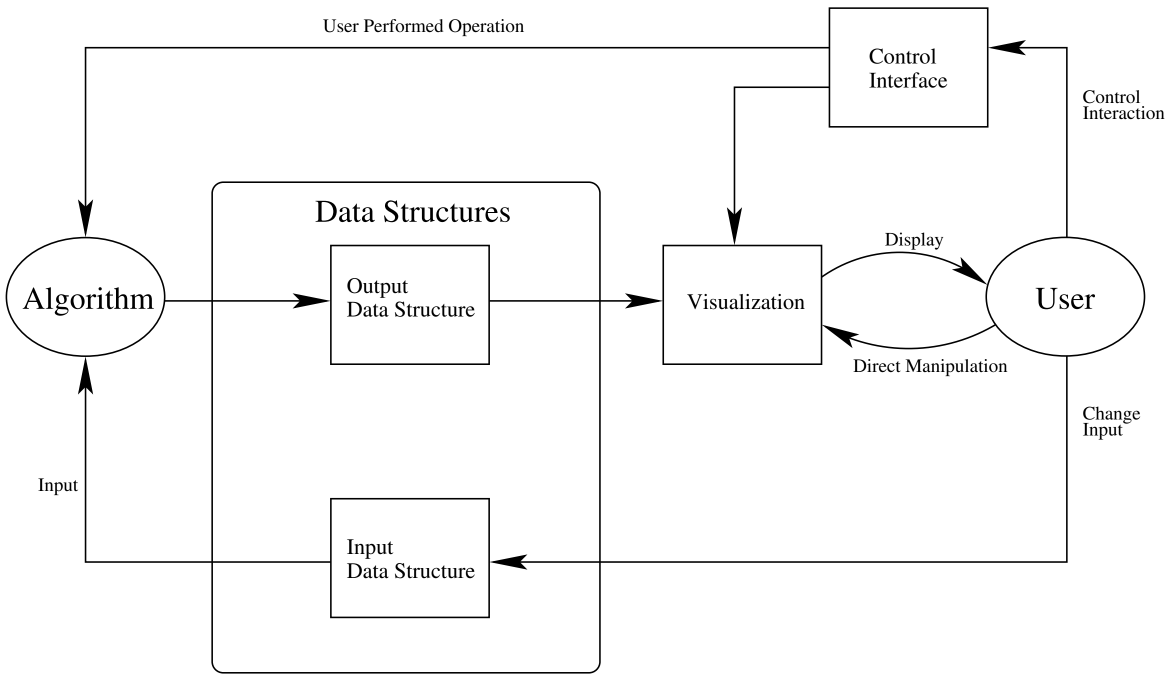


Figure 1.1: General overview of traditional algorithm animation systems. Four interaction cycles can be identified. (1) Control interface allows the user to customize the layout, change animation speed, direction, *etc.* and (2) manipulate the data structures by calling predefined operations. (3) Direct manipulation enables interaction between the user and the graphical entities on the display. (4) The algorithm can be executed with different inputs.

the graphical display and the visualization is automatically updated to match up the changed structure. If the user processes the data structure as the algorithm did in the previous example, we say that the user simulates the algorithm [68, 71]. We refer to such an simulation process simply as *visual algorithm simulation* or *algorithm simulation* (see Figure 1.2).

Figure 1.2 is a simplified version of Figure 1.1 but it still allows more interaction between the user and the system. This is because we do not distinguish between input and output structures, and allow the user directly to interact with any data structure visualization. Thus, an output structure can be used as an input structure for the same or another algorithm. Similarly, the simulation process can always be continued with any data structure, for example, by performing some predefined operation on it.

Naturally, algorithm simulation includes the idea of visual debugging, and it is straightforward to apply the algorithm simulation to produce algorithm animations.

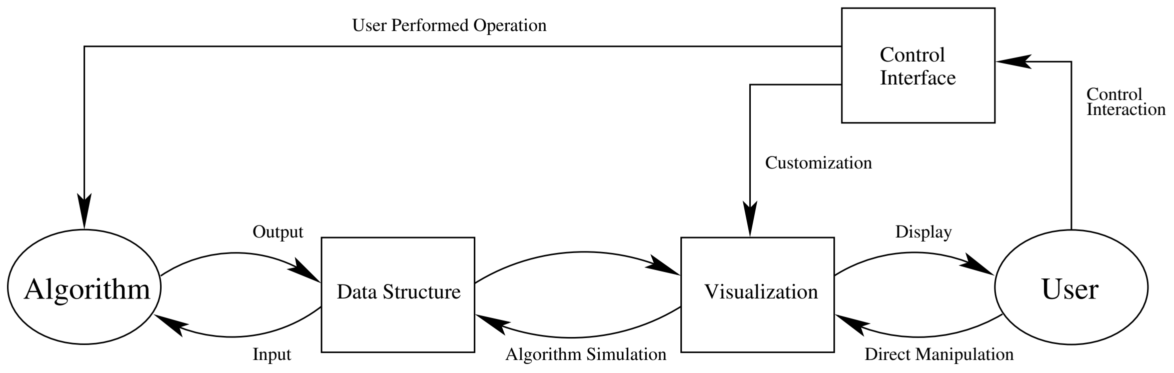


Figure 1.2: General overview of algorithm animation and simulation. Five interaction cycles can be identified. The Control Interface (1) and (2) and its functionality remains the same as in Figure 1.1. (3) In addition, direct manipulation is allowed, but (4) the changes can also be delivered into the underlying data structures in terms of algorithm simulation. Again (5), the underlying data structure can be passed to the algorithm as an input by applying algorithm simulation functionality.

However, algorithm simulation is a much more powerful concept than, for example, visual debugging. One way to distinguish between these two is to consider the source code. Visual debugging intrinsically requires that the source code exists. However, algorithm simulation only requires that the user has a mental model of the algorithm to be simulated. We are not only able to change the values of variables and data structures but also, for example, the order of the operations the algorithm performs.

Another way to look at the situation is to think in terms of performed operations. We can consider an operation to be a primitive operation if we cannot divide it into lower level operations. For example, adding unity into an integer is a primitive operation in this sense. However, the complexity of operations the user can perform during the simulation process has no limitations. Complex operations can be formed out of primitive ones in terms of algorithms. This yields another problem, how to divide the complex operations. Here, the content complexity can be managed by seeing these complex operations as algorithms that we can look into in terms of visual debugging. In the wider context, however, the user performs these operations in terms of algorithm simulation.

1.1 Motivation and Research Problem

Data structures and algorithms are important core issues in computer science education. Because they often form complex concepts, algorithm animation, visual debugging and algorithm simulation are all attractive methods to be considered learning aids. The key question is, however, how we should apply these methods in order to actually help the students to cope with these complex concepts. A lot of research has been carried out to identify the great number of rules that we must take into account while designing and creating effective visualizations and algorithm animation for teaching purposes [6, 21, 43, 45, 83]. See, for example, the techniques developed for using color and sound [21] or hand-made designs [43] to enhance the algorithm animations. We argue, however, that these are only minor details (albeit important ones) in the learning process as a whole. In order to make a real difference here, we should change the point of view and look at the problem from the learner's perspective. How can we make sure the learner actually gets the picture? It is not what the learner sees but what he or she does. In addition, we argue that no matter what fancy visualizations the teacher has available, the tools cannot compete in their effectiveness with environments in which the learner must perform some actions in order to become convinced of his or her own understanding.

From the pedagogical point of view, for example, a plain tool for viewing the execution of an algorithm is not good enough [54]. Even visual debugging cannot cope with the problem because it is always bound to the actual source code. It is still the system that does all the work and the learner only observes its behavior. At least we should ensure that a level of progress in learning has taken place. This requires an environment where we can give and obtain feedback on the student's performance. Many ideas and systems have been introduced to enhance the interaction, assignments, mark-up facilities, and so on, including [3, 22, 47, 49, 81, 97, 109]. On the other hand, the vast masses of students in basic computer science classes have led us into the situation in which giving individual guidance for a single student is impossible even with semi-automated systems. Thus, a kind of fully automatic instructor would be useful such as [7, 13, 16, 40, 51, 56, 57, 97, 102]. However, the topics of data structures and algorithm are more abstract than those introduced on basic programming courses. Therefore, systems that grade programming exercises are not suitable here. We are more interested in the logic and behavior of an algorithm than its implementation details. The problem is to find a suitable application framework for a system that is capable of interacting with the user through general purpose data structure abstractions in this logical level and giving feedback on his or her performance. Quite close to this idea comes PILOT [16] in which the learner solves problems related to graph algorithms and receives graphical illustration of the correctness of the solution, along with a score and an explanation of the errors made. However, the current tool covers only graph algorithms, and especially the minimum spanning tree problem. Hence, there is no underlying general purpose application framework that can be extended to other concepts and problem types without creating a new interface.

In this thesis, we will apply the concept of algorithm simulation to reach the goal set and fill the gap between visual debuggers and real-time algorithm simulation. The idea is to develop a general purpose platform for illustrating all the common data structure abstractions applied regularly to illustrate the logic and behavior of algorithms. Moreover, the platform should be able to allow user interaction in terms of algorithm simulation. As an application, we support exercises in which automatically generated visual feedback is possible for algorithm simulation exercises. We call such a process *automatic assessment* of algorithm simulation exercises [66].

1.2 TRAKLA

The history of this thesis goes back as far as the early 1990's. At that time, the TRAKLA project was launched to look into the possibility of automatically evaluating students' answers to exercises for the course on Data Structures and Algorithms [56]. At the beginning, the exercises were solved by manually simulating algorithms with paper and pencil, thus having neither graphical representations nor algorithm animations. The answer was formed into a specific format to be turned in via email and the answer was assessed by the system. There was, however, a natural way to improve the concept by developing a separate graphical user interface called *Tred* [64]. The user interface was able to portray several data structures graphically that the learner could directly manipulate. It also had the capability of hiding unnecessary format details and to turn in the exercise after the algorithm simulation process was performed by the student. Moreover, it was capable of representing the simulation sequence as an algorithm animation.

TRAKLA has evolved to be a crucial part of a more complex open[1] distributed[2] learning environment called *WWW-TRAKLA*, which is a distributed learning environment in which the whole set of common telematic tools (such as email, newsgroups, WWW forms, and such special purpose tools as TRAKLA) are combined. The system

[1]This is both a pedagogical and a technical term, since from the pedagogical point of view, open is roughly defined as an environment in which usage is independent of time and place, and from the technical point of view, open refers to systems that conform to well-defined interfaces.

[2]This is purely a technical term which refers to the concept of autonomous processing elements which cooperate by sending messages to each other.

has proven to be a valuable teaching and learning tool during the past decade [69, 80]. We found, among other things, that there was no significant difference in the final examination results between students doing instructive simulation exercises in classroom sessions and learners using our web-based learning environment.

The environment has been in production use since 1991 in a course with yearly enrollment of 400-700 students each solving 25-30 exercises. From the student's point of view, TRAKLA has been a meaningful environment for learning the functionality of data structures and algorithms. To highlight a few of TRAKLA's best characteristics, we argue that it is possible to

1. produce personally tailored assignments for each student,
2. provide most of the exercises with graphical visualizations,
3. allow direct manipulation of graphical display to specify a solution,
4. allow on-line submission of solutions,
5. perform automatic assessment for a solution,
6. provide immediate feedback for a solution,
7. provide the opportunity to revise incorrect answers,
8. generate and provide a model solution for an assignment,
9. produce statistics about students' overall performance,
10. research the overall performance of a given class, and
11. encourage natural discussion about topics between students.

The immediate feedback is a particularly important feature here and it is based on the automatic assessment of the submitted exercises. Students receive an email for each submission within a few minutes, containing additional information about their performance. We stress here that a student must think the revised solution through for each submission, since the solution space for exercises is simply too large for using a simple trial-and-error method[3] without the rethinking and analysis phase of the outcome.

It should be noted, however, that most of these features cannot be obtained without a computer aided learning environment. For example, there is a huge difference

[3]Or any bricolage extreme [115] as some authors refer to this.

between feedback provided within a few seconds or minutes instead of within a few days or even weeks. Unfortunately, this is often the case on many courses. Alternatively, if there is no automatic assessment involved, the possibility of allowing students to revise their partially correct solutions is probably not even considered.

1.2.1 Problems with TRAKLA

Despite the fact that many new features have been included in the TRAKLA system during the past decade, a lot of new ideas remain to be added and a number of problems are still to be solved. First, development of new exercises is currently rather a slow process and requires the technical skills of a *system developer*[4]. This is rooted in the fact that every single exercise has its own way to produce the model solution, and to provide the feedback. This could be avoided by generalizing the automatic assessment procedure. The overall goal is to provide a system in which the set of exercises could be easily extended by the teacher.

Second, the automatic assessment raises the questions in which circumstances it is safe to perform the assessment procedure automatically and how to avoid the possible pitfalls. The old system provides "closed" exercises that usually have only one correct solution (one-one mapping between the input set and the correct output set). We would like to extend this scope of automatically assessed exercises by additionally allowing more open questions in which an exercise may have several correct solutions.

Third, the model solutions for assignments are only available in non-graphical (verbal) form. However, different learners exhibit different learning styles [42] that may affect on their performance. For example, verbal learners could perform better using tools such as TRAKLA whereas visual learners could perform better if we could provide the model solution also in graphical form, *i.e.*, as an algorithm animation.

Unfortunately, the design and architecture of the TRAKLA system did not meet the requirements for adding on these new features. Since similar systems did not exist, we decided to develop a new system having the advantages of TRAKLA and Tred in one single package called TRAKLA2. The first phase of this work is presented in this thesis and describes the design of the application framework called *Matrix*. Moreover, a prototype implementation of the new TRAKLA2 environment has been build on top of Matrix.

[4]By a system developer we mean a person who is familiar with all technical details of the overall system and the framework.

1.2.2 Matrix

Our goal has been to develop an application framework for algorithm simulation tools. On top of the framework, it should be possible rapidly to build applications such as TRAKLA. The applications should be capable of automatically assessing exercises and giving immediate feedback on students' performance as described on page 13. The further development of the system should require only programming skills (and not any technical skills of system developers) to produce a new exercise. The framework should also provide a meaningful starting point for developing similar environments for other related areas such as software visualization in general, other courses in computer science, and software engineering.

The first challenge is to determine a visual representation for all possible abstractions needed, in order to manage the algorithm animation. It should be noted, however, that we have been developing a prototype and therefore have possibly omitted, for example, some customization details. We want to answer the question *how we could* represent a data structure rather than *how we should* represent a data structure. To provide powerful tools for interaction between the system and a student, we also need a way to simulate these algorithms. Thus, we must define the *simulation model* for algorithms and data structures, based on the formal framework. The simulation model consists of formal definitions of the visual data types needed by the algorithm in order to function properly. This model is also used as the framework for the whole design and architecture of the system, thereby making components created during the development of the environment reusable.

Since we are dealing with a complex system, it is adequate to use formal methods for the definitions of the most crucial parts of the system. This has led to several new ideas of how to describe the elements of data structures and algorithms compared to textbooks. However, most of the terms and notions are more-or-less adopted from literature, as discussed later in the text. Moreover, object-oriented analysis and design methods, together with *de-facto* UML standard (Unified Modeling Language) and its (mainly graphical) notation to express designs, have been used [67]. In particular, the Java programming language has been selected as the primary implementation language because of its many good properties.

1.3 Contributions

In this thesis, we argue that software visualization tools and techniques can provide a valuable learning aid for building viable mental models for several concepts in algorithmics. Learning environments based on such tools can result in as good learning as traditional teaching methods can but requires less time and effort from instructors [69, 80]. In addition, these environments can provide new opportunities for the learner, which could not be achieved by means of traditional teaching methods. For example, the learner can receive immediate feedback on his or her performance at any place or time while interacting with the environment.

A purely pedagogical approach to develop such learning environments is not sufficient. Computer science education has its special characteristics that general purpose environments cannot facilitate [9]. For example, there exists a number of general purpose tools and environments (that are too numerous to be cited here) which employ simulation as a technique to foster learning. However, what is common to all of them is that most of them do not support automatic assessment in which the learner can receive accurate feedback on his or her performance. By focusing on the context of computer science education and especially to data structures and algorithms, however, we can improve the concept of simulation even further. Thus, in this sense, we see that our research on computer science education is more related to computer science than educational sciences. We summarize the contributions as follows:

1. The theoretical simulation model for data structures and algorithms consisting of well-defined mathematical models for some elementary data types together with two new concepts of abstractions called fundamental data type [65] and conceptual data type.

2. A novel way to specify algorithm animations in terms of algorithm simulation [67, 71]. This new approach can be applied to building a framework that is capable of delivering special purpose algorithm simulation exercises in which the learner simulates an algorithm by directly manipulating the visual objects on display [70]. In addition, the framework is capable of grading and giving immediate feedback on the correctness of such simulations. Moreover, the model solutions for the exercises can be expressed as algorithm animations [66].

3. Simulation Exercise Taxonomy characterizing different aspects of such exercises.

4. Learning environment based on the method of algorithm simulation exercises that is proven to be as effective as traditional teaching methods [69].

5. Research carried out to identify several aspects of effective use of algorithm animations and algorithm simulations in education [80, 89].

6. Freely available application framework [72] (proof of concept)[5] for creating

[5]at http://www.cs.hut.fi/Research/Matrix/

visualizations, animations and simulations of user-made algorithms in a conceptual level [68] together with several applications based on this.

1.4 Organization of this Thesis

This thesis consists of 10 chapters, which are organized as follows. In Chapter 1, we give the motivation for the thesis and introduce a brief overview of the research topic and its history. In Chapter 2, we broaden this view by introducing the key concepts needed to understand the different aspects of this thesis. The discussion is continued by creating the formal theoretical construction, which forms the basis for the rest of the thesis in Chapter 3.

The theory is applied to produce a prototype implementation that serves as the proof of concept. We take several perspectives to introduce the concepts of visual algorithm simulation within this prototype. In Chapter 4, we emphasize the difference between traditional software visualization systems and our approach by introducing a novel way to specify new algorithm animations from the user's perspective. Chapter 5 gives deeper understanding of the techniques used to achieve the algorithm simulation functionality. This point of view is extended even further in Chapter 6 by introducing the ready-made library components to be reused in order to create new data structures more rapidly and to allow the full power of algorithm animation features for user made data structures. In Chapter 7, we revisit the theoretical framework and address all the fundamental design principles that should be taken into account when implementing new features to this prototype system.

Chapter 8 gives a brief overview of how to build an application on top of the framework and especially how to apply the framework in educational context. The properties of such applications are discussed in Chapter 9. Finally, in Chapter 10, we give the summary and conclusions of the thesis.

Chapter 2

Algorithm Animation and Simulation Techniques

Software visualization (SV) [17, 38, 95] has been polarized toward two opposite domains. In one domain that we call *program visualization* (PV), views of program structures are generated automatically. These types of views, which refer to "debugging" of an algorithm by tracing its execution step by step, are generic, low-level views and not expressive enough to convey adequately how an algorithm functions.

In the second domain, called *algorithm visualization* (AV), we are interested in visualizing all the states of the data structures during the execution of an algorithm. Moreover, we want explicitly to show the object structures (components, subcomponents, and their values) that are required to understand fully the logic and the behavior of the algorithm. We might omit some trivial data types or variables which are implicitly present in the visualization or which otherwise do not offer any additional information about the behavior. However, the visualization as a discrete sequence of views of data structures should include enough frames fully to explain the operations the algorithm performs. Great care must be taken to preserve only those characteristics of a data structure that are essential. This again depends on the data structure, since these characteristics and their meanings are always relative to some metaphor level concepts.

2.1 Algorithm Visualization

Some authors distinguish between *static algorithm visualizations* and *dynamic algorithm visualizations* in which the first ones are more like a "series of still images" while the second ones refer to the algorithm animations. We, however, would like to see the whole field of visualizations and different kinds of graphical symbol systems in terms of a continuum as follows.

An advantage of graphical representations and their dynamics is that structural connectivity and symmetry are emphasized. Things that are conceptually related appear to be close. They also provide a rich syntax to define visually such concepts as links, flows, and directions. For most people, this seems to be easier to grasp than procedural encodings. The reason seems to lie in the inherent parallelism of the human visual system and the corresponding importance of visualization as a problem-solving tool.

Managing complexity is at the heart of programming and has guided the development of its tools. The repertoire of specialized graphical symbol systems and tools makes it possible to bridge, in a reliable fashion, large conceptual distances between a modeler, his or her mental model of a system and a computer executing its computational image. By means of *algorithm visualizations*, it is possible to provide conceptual displays not only to concretize these mental models but also for executing the model in terms of algorithm animation and simulation.

2.2 Algorithm Animation

Software Visualization Technology includes all the tools and methods to illustrate how algorithms work by graphically representing the algorithms in action. From our point of view, the dynamics of the visualization plays the major role here. Thus, another way to distinguish between the two extremes of Software Visualization is to consider the phenomena as an animation.

We adopt the definition of *Software Visualization* introduced by Price *et al.* [95]. They make the distinction between *Algorithm Animation* and *Program Animation* where the first one is considered to be dynamic visualizations of algorithms that are *implemented later* while the latter is considered to be dynamic visualizations to enhance human understanding of the *actual implementation* of programs. However, it should be noted that many systems can be subsumed by the terms algorithm and program animation [60].

2.3 Algorithm Simulation

There are two different styles of simulation by Kreutzer [74]. On the one hand, there is its classical application as a vehicle for numerical predictions (which is not discussed further in this thesis). On the other hand, simulation could be used for *exploring and understanding symbolic models* of complex systems. This kind of descriptive model is mainly speculative. It offers symbolic representations for some problem space, without guidance on how to search it. Use of this kind of model is therefore an experimental technique for exploring "possible worlds" through simulation processes. Any exploration of states of behavior under a chosen setup is a simulation experiment in this sense.

"On a rational basis validity can be proved through correspondence with the theory a model is designated to instantiate: that is, by showing how its behavior can be logically derived from purely formal properties of its representation." [74]

In this respect, we will give a formal representations for the chosen *discrete-event simulation model* [77] in the next chapter. The *challenge is to construct a simulation tool based on this theoretical framework.* A user could then use this simulation tool for "exploration of states of behavior" of algorithms and data structures.

The goal of *Algorithm Simulation* is to further the understanding of algorithms and data structures inductively, based on observations of an algorithm in operation. The setup defines the set of circumstances - observed data structures, initial state, termination conditions - under which an algorithm will be observed. Using a descriptive model involves searching for a solution through a finite number of states to which the data structures can be subjected. Therefore, running an algorithm under a given setup corresponds to one such experimental simulation.

There is also another possible view: "Simulation may serve as a vehicle to gain sufficient insight to construct simpler analytical models, focusing on those aspects of a system that prove to be relevant." [74] In this way, we can see that the construction of simulation tools is justified also from the researcher's point of view. A different facet of the simulation method is rooted in the fact that it is an experimental technique, comprising such aspects as model calibration and data collection as well as experimental design and output analysis.

Representation will always be needed for simulation, since it is by definition an experimental technique where *data generated by each experiment must be collected, summarized and reported in a meaningful way.* While exploring states of behavior, a representation for each state is needed. In algorithm visualization, these representations are visualizations of the data structures needed by an algorithm. Hence, all the three major aspects defined above, algorithm visualization, animation, and simulation, seem to be seamlessly integrated in this work.

Direct Manipulation

Direct Manipulation is a method that requires human interaction for monitoring and executing the simulation model, e.g., a sequence of visualized data structures. In order to simulate a system, an approximate mathematical model is needed. Here, we can see the visual representation to act as the model of the underlying physical data structure. In fact, it is an approximation of the real physical data structure because its operational interface is only limited to those operations explicitly implemented.

The manipulation process is conceptually the opposite of the algorithm animation with respect to the information flow. Where algorithm animation visually delivers the information from the system to the user, direct manipulation delivers the input from the user to the system through a graphical user interface (see (3) in Figure 1.2 on page 10). Generally, if human interaction is allowed between the visual demonstration of an algorithm and the user in such a way that the user can directly manipulate the data structure representation, the process is called direct manipulation. However, algorithm simulation allows the user directly to manipulate not only the representation but also the underlying implemented data structure (see (4) and (5) in Figure 1.2). Thus, algorithm simulation is generalization of direct manipulation in this sense.

2.4 Animation Techniques

Algorithm animation maps the current state of an executed algorithm into the visualization that is shown to the user. The visualization can be animated based on the executed operations between two consecutive states. Several techniques exist to specify how the visualization is connected to the algorithm. In this section, we classify these connection techniques and give examples of each of them. However, the list of actual systems employing the techniques is too large to be listed here. A number of new systems have been presented recently at workshops and conferences, including Dagstuhl Seminar on Software Visualization 2001 [38] and the Second International Program Visualization Workshop 2002 [10]. The first one is also a good starting point to get familiar with the classical systems and concepts as well as the state-of-the-art in Algorithm Animation. Finally, it should be noted that many of the systems belong to several categories. Thus, the following examples are only representatives of the applied techniques.

2.4.1 Annotation

Annotation is one of the classic methods of producing algorithm animations. For example, in Polka [111], C++ code is augmented with *Algorithm operations* that establish the connection between the annotated algorithm and the animation scene. In general, the original source code can be transformed or augmented to include the annotated code lines that produce the animation.

Automatic annotation is a particularly interesting approach in which the system automatically adds necessary animation calls into the source code. Systems employing this technique to program animation include UWPI [50], Eliot [76], and its followers, Jeliot [48], and Jeliot 2000 [11, 12]. For example, Eliot is based on self-animating classes that use the services of the Polka package. The system automatically extracts all the data objects used in the original source code and makes use of the self-animating components. The self-animation relies on the C++ *operator overloading* ability. Some operators in the self-animating classes have been redefined to produce animation as a side-effect.

2.4.2 Scripting Languages

One approach to applying annotation are those of employed by the scripting languages such as BALSA-II [19] and JAWAA [94]. The JAWAA animation system, for example, is written in Java, but the JAWAA commands can be added to a program written in any programming language. The system animates the sequence of commands (script language) with any Java compatible web browser. There is also an editor available that allows one to lay out animations graphically. The edited animation can be combined with animations hand-written in JAWAA, and reduces the time needed to create complex animations.

2.4.3 Aspects of Invasiveness

Non-invasive visualization specification styles for altering the behavior of algorithms can be used to represent the algorithm in action without modification. These methods allow a variety of useful functionality including changing the input, scope, and layout of interactive algorithm animation.

Annotation is inherently an invasive method. However, other methods also have – more or less – a level of *invasiveness* involved[1]. We can consider the visualization

[1]Even if this is untrue for some limited scope linear code examples the concurrent systems, as a more general example, raises the question again.

specification style "an act of measurement" in a dynamic context. The original phenomenon (an algorithm in action) and the corresponding result (the visualization) is therefore always affected by the measurement. This question is generally included under the heading of *the measurement problem* [2].

However, the invasive character of the specification style can be limited (at least in a limited scope, *i.e.*, from the user's point of view) if the algorithm can be executed, for example, with different input data without need of recompiling. Therefore, we can see two instances of an inherently invasive method to be less and more invasive in contrast to each other depending on the implementation. Alternatively, even within one particular system there might be actions that can be considered to be invasive or non-invasive depending on the scope. See, for example, Jeliot above which is based on annotation that is an invasive method; but from the user's point of view, automatic annotation does not require any source level intervention at all.

2.4.4 Event Driven vs. State Driven Approach

One of the fundamental ways to classify systems is to determine whether they represent an *event driven* or a *state driven* approach [37]. Within the event driven approach, the visualizer annotates the key points in the program code to dispatch *interesting events* that change the display. The method allows the production of highly customized and intuitive animations. The customization, however, requires that the source code be available and the visualizer have a deep understanding of its operations. Moreover, the manual annotation procedure is always an invasive method. Systems based on interesting events include pioneering BALSA [18, 23], its successors BALSA-II [19], Zeus [20], and Polka [111]. Newer systems include the GANIMAL framework [39] and ANIMAL [100].

An alternative approach is to monitor the data structures during the execution of an underlying algorithm. This state driven approach observes the changes in *interesting variables* and creates the mapping between the state of the computation and the corresponding visualization. The method requires little or no knowledge of the source code. On the other hand, pure state mapping limits the changes on the display to those that actually modify the underlying data structure. Thus, customizing the visualization to include, let us say, a comparison between two variables, requires an additional mechanism. Two well-known systems based on state mapping are PAVANE [98], and LEONARDO [35, 36].

2.4.5 Animation Frameworks

A *framework* is a set of generic reference data types for a given problem domain including a control flow system that performs interaction among the objects instantiated from it. Frameworks are widely used because it is easier to modify and customize a framework than to write an application from scratch. An application programming interface (API) is usually provided in order to allow the reuse of the reference data types in terms of *composition* and *class inheritance* (see, for example, JDSL Visualizer [7]). We call such frameworks *application frameworks* as opposed to *conceptual frameworks* that are usually more loosely tied to the target program (see, for example, Tango [107]).

Here, we can distinguish between specification styles by looking at the level in which a framework interacts with the target program. Application frameworks are closely tied to the programs they are visualizing. Thus, such animation libraries (see, for example [76, 96]) should fall in this category. However, pure conceptual frameworks do not require any system-code coupling at all. Here, Salsa and Alvis [53] adopts the broadest interpretation of what conceptual framework could be by supporting learner costructed pen-and-paper visualizations.

Of course, systems also exist that are neither application nor conceptual frameworks in this sense. For example, many debuggers [117] fall into this category even though they might have some connections to software visualization. The aim of these systems is something other than the production of algorithm animations.

2.4.6 Top Down vs. Bottom Up Visualization

Of course, many application frameworks also exist for software visualization domain. One such framework is Tarraingím by Noble *et al.* [90, 91]. The idea is to monitor the target program to gather information on the program's actions. In particular, the target program's implementation is encapsulated behind interfaces. Tarraingím produces visualizations *top down* — working from abstractions rather than their implementations. This is in contrast to *bottom up* techniques — using annotations, procedures [86] or mapping rules [32, 31, 35, 114] to extract abstractions from their implementations. The target program's design abstractions can be visualized without the program being modified or annotated, or the implementation details of abstractions being exposed to the visualization system.

Top-down visualization is a particularly interesting idea because the conceptual displays shown to the user may be abstractions of algorithms, data structures, or any combination of the two. The concepts are high-level views because the connection technique is based on monitoring the target program's public operations. However, the visualization system only needs to know how to *use* the abstractions it is visualizing, not how to *implement* them. Moreover, the different views can be reused to display any object that implements the required interfaces.

2.4.7 Other Aspects

There are many other important aspects to be taken into account while designing a new visualization system. For example, in many case, the programming language might allow or restrict some specific connection technique to be applied. Different language paradigms, such as imperative, functional, object oriented, and logical programming paradigm, may need different techniques due to their unique styles of computation. On the other hand, many domain specific animation systems exists, for example, for computational geometry, concurrent programs, real-time animation, computational models, and animation of proofs. However, for our purposes a slightly larger granularity helps to emphasize the major conceptual differences among various connection techniques. Hence, we are going to address these issues only very briefly. Better surveys exist to get familiar with these topics (see, for example, Dagstuhl Seminar on Software Visualization 2001 [38]).

2.5 Simulation Techniques

We do not know any other system that is capable of direct manipulation in terms of algorithm simulation similar to Matrix. Astrachan *et al.* discuss simulation exercises while introducing the Lambada system [3, 4]. However, their context is completely different, because the students simulate models of practical applications, partly coded by themselves, and the system is used only for illustrating the use of primitive data structures without much interaction with the user.

On the other hand, GeoWin [24] is a visualization tool for geometric algorithms in which the user can manipulate a set of actual geometric objects (e.g. geometric attributes of points in a plane) through the interactive interface. However, the scope of the system is quite different from Matrix. While all the relations between objects in GeoWin are determined by their coordinates in some geometric space, the relations between the underlying object instances in Matrix are determined by their interconnected references.

More close to Matrix, in this sense, comes CATAI [26] and JDSL Visualizer [7]. They allow the user to invoke some methods on the running algorithm. In terms of method invocations it is possible directly to access the content of the data structures or to execute a piece of code encapsulated in an ordinary method call. However, Matrix also provides a user interface and an environment for this task in terms of direct manipulation. Thus, Matrix not only allows method invocations but also the facility to simulate an algorithm in a more abstract level by drag & dropping new input data at any time to the corresponding data structure. Moreover, CATAI is targeted at distributed multi-user environment and is dependent on CORBA (Common Object Request Broker Architecture). Thus, the animated algorithms are implemented as CORBA objects at the server side. A remote animation client then instantiates the proper animation by invoking remote method calls.

Moreover, Matrix is designed for students working on a higher level of abstraction than, for example, JDSL Visualizer. In other words, JDSL Visualizer is designed for a *programming course* to provide interactive debugging tools for educational purposes (Program Visualization and Animation) while Matrix is intended for a *data structures and algorithms course* to illustrate and grasp the logic and concepts of data structures and algorithms (Algorithm Visualization and Animation). Of course, both kinds of tools are fit for use.

In Matrix, the user can directly change the underlying data structure on-the-fly through the user interface. Also, for example, Animal [100] and Dance [108, 110] both have the look and feel of building an algorithm animation by demonstration. In addition, the JAWAA [94] editor is capable of producing scripting language commands that can be animated. However, within these systems the user does not manipulate an actual data structure but only a visualization of an alleged structure. The system produces the algorithm animation sequence based on the direct manipulation. However, while creating, for example, an AVL tree demonstration, it is the user's concern to maintain the tree balanced. In Matrix, several levels of interaction are possible: one can manipulate a tree as with Animal or it is also possible to invoke an actual insert method for the AVL tree that inserts an element into the appropriate position. The actual underlying structure is updated and the new state of the structure is visualized for the user.

Finally, Matrix is implemented in Java, which gives us more flexibility in terms of platform independence, compared to older systems such as Amethyst [87] and UWPI [50]. Of course, Java has its own restrictions, but Java together with WWW has given a new impetus to algorithm visualization techniques.

Visual Programming

Visual programming is one of the particularly interesting ways to look at the domain of simulation techniques. One of the most important categories of purely visual languages can be characterized by their reliance on visual techniques throughout the programming process. The programmer manipulates a graphical display to create a program that is also compiled, debugged and executed in the same visual environment. Two examples of this idea are VPL [27] and FORMS/3 [25].

As a whole, visual programming is not the same as program visualization. However, both domains share a mutual interest in visualizing the many well known concepts regularly used in algorithmics. Moreover, visual programming seems to be more closely related to algorithm simulation than algorithm animation. See also, for example, "programming by demonstration" [34, 84, 108, 110]. One way to look at this is to see algorithm simulation as a niche between algorithm animation and visual programming.

Some authors [60] classify Visual Programming (VP) as an algorithm animation specification technique. This has led to such characterizations as follows: "such a graphical notation itself is a kind of static code/data visualization." However, VP is much more than that. Its whole power is undervalued if it is considered to be only a technique to produce visualizations and animations. Actually, its contributions are completely in the opposite direction, namely in its ability to allow the user to manipulate the visualization in such a way that the logic of the user's actions also has a valuable interpretation. Perhaps this undervalued classification is a consequence of lack of awareness of such manipulation techniques in general. Visual programming, however, is beyond the scope of this thesis.

2.6 Summary

To highlight some of the categories, the *connection techniques and specification styles* have their pros and cons depending on the educational purposes from the pedagogical point of view. For example, pioneering visualizers such as BALSA [23], Tango [107], Polka [111], and Polka's application (front-end) Samba [109] are powerful tools for animating algorithms by using the *signaling interesting event* approach. Thus, one possible pedagogical approach is to foster student creation of algorithm animations for learning purposes as done by Stasko [109]. On the other hand, a system without any intervention requirements makes it possible to the programmer to create limited algorithm animations with no prior knowledge of the actual animating system. For example, Jeliot algorithm animation package [76] describes an approach to array algorithm animation that requires no manual intervention into the algorithm itself. The automatic animation helps the student to understand what the programmed algorithms are doing and to trace errors in terms of visual debugging. From the research point of view, however, it is possible to animate, for example, certain low level utilities through application program interfaces (API), thus learning how they are function without even looking into the source code.

As we can see, there are efforts to develop systems in which the programmer's *manual intervention* to produce an animation is limited to a minimum. For example, the Astrachan group [3, 4] are developing an "interesting event approach" based on the pioneering systems BALSA, Tango, Polka, and Samba. Another characterizing property is whether a system is intended to be used for program animation as a visual debugger or for concept animation in order to grasp the logic of a visualized concept. The Eliot algorithm animation package [76] does both of these and completely automates the intervention process.

The other issue is the level of interaction. Program animation naturally requires a "debugging functionality" [48, 73]. On the other hand, concept animation seems to provide more sophisticated displays with which to interact. The student may change the behavior of the system by programming new methods as with JDSL Visualizer [7] or by simulating a sequence of primitive operations as is the case with TRAKLA and TRAKLA2. Generally speaking, we could say that in program animation we are interested in algorithms by focusing on the data structure visualization, and in concept animation we are interested in the data structures by focusing on the algorithms and their behavior.

Much of the recent work has focused on developing more interactive systems to teach and to learn data structures and algorithms. Most of the systems are programming language dependent and intended for a programming class as tools for improving the quality and functionality of computer programs. In this kind of interactive process, the feedback is essential and thus the quality of the feedback is becoming increasingly important. However, most of the systems give no guidance on how to give or to obtain feedback on students' performance. On the other hand, there is only a very small number of systems in which automatic evaluation of students' performance is possible [7, 13, 51, 102]. Most of these systems are intended for testing whether the students' program is producing correct outputs for a given input or whether errors were encountered. In this respect, TRAKLA and TRAKLA2 represents a unique genre by providing the automatic assessment of algorithmic assignments.

Even though feedback to students is the most essential form of feedback, very few of the systems are also capable of assessing students' work and giving continuous feedback on their performance to the teacher. This is very important in improving the teaching process. Another concern is whether the student is actually doing something useful with these tools or merely playing and enjoying, for example, the animation without any actual learning. As reported by Baker [7], some students have difficulties seeing the systems as complementary tools, and therefore some of them do not use them in the most effective manner.

Traditionally, grading has been based on final examination, but a variety of weekly homework assignments are also widely used. Thus, automatic assessment of these kinds of assignments has been seen to provide great value to large-scale courses. However, our latest experiments seem to support the view that automatic assessment also has a role to play with small student groups. For example, learners studying while they have a steady job seem to benefit a great deal from environments that can be accessed at any place or time.

User Roles

As we have pointed out, the different uses of simulation tools can be separated into several roles. In the following, we adopt the roles identified by Price *et al.*[Price] in their principled taxonomy. The roles of programmer and developer are already mentioned. The *programmer* implements the actual data structures, and algorithms ought to be visualized. On the other hand, the underlying simulation tool is implemented and documented by *developers*. Finally it is the *visualizer* that specifies the visualization the *user* sees and interacts with in terms of simulation tool operations.

The separation of roles of programmer and visualizer is essential here because the programmer might be unaware of the simulation tool *before* making the decision to visualize the piece of code. Similarly, the visualizer might want to provide the visualization for third party use, *i.e.*, users and not for himself at all.

In the following chapters, we will present the implemented Matrix framework for algorithm animation and simulation from different perspectives, *i.e.*, roles. However, we start by introducing the theoretical background which provides a deeper insight into the phenomena by formally defining the basic components needed to build a suitable model for an algorithm simulation system.

Chapter 3

Theoretical Point of View

In this chapter, we give definitions of all the necessary terms and notions in order to understand the design and architecture of the implemented Matrix system. We use several different kinds of techniques to illustrate our point of view.

From the theoretical point of view, it is natural formally to define those concepts and examples used in this thesis. On the other hand, from the technical point of view, we have decided to apply object-oriented modeling and design, and therefore it is natural to use real engineering tools, diagrams and methods to illustrate those same examples in the context of how this theory is applied in practice. We also include the basic definitions and semantics for such concepts as algorithms, data types, data structures, and abstract data types that are more or less adopted from the literature. However, some of these concepts are reconstructed in terms of discrete mathematics by the author. What we mean by *discrete mathematics* here is the selection of topics from set theory, combinatorics, graph theory, and algebra (see, *e.g.*, [44, 79]).

On the other hand, this chapter also defines two completely new generalizations called conceptual data type and fundamental data type. These concepts do not refer, despite the names, to any concepts identified so far. Finally, at the end of this chapter, we apply the constructed theoretical model for the design of binary search trees, as described in the case study.

3.1 Prerequisites

In this section, we collect some basic terminology and notations concerning sets, relations, and graphs that will be used throughout.

3.1.1 Basics of Set Theory

We use standard, somewhat axiomatic, notation for sets. Thus, the following notions should be familiar to the reader. We write

1. $A \subseteq B$, if A is a subset of B,

2. $A \subset B$, if A is a proper subset of B,

3. $|A|$ as the cardinality of a set A,

4. $\overline{A}$ as the complement of A,

5. $R \subseteq A \times A$ as a binary relation on the set A,

6. R^{-1} as the inverse of relation R, and

7. $\bigcup F$ as the union (sum-set) of all sets belonging to F.

For a binary relation $R \subseteq A \times A$ we shall also use infix-notation and write xRy instead of $(x,y) \in R$. In addition, we define the following notions concerning partial orders that will be used later in the text.

Definition 3.1.1 *Let $R \subseteq A \times A$. Then R is said to be an* ordering relation *or simply an* order *on the set A, if*

1. $(x,y) \in R \wedge (y,z) \in R \rightarrow (x,z) \in R$ *(transitivity),*

2. $\neg \exists x,\ (x,x) \in R$ *(irreflexivity), and*

3. $\forall x,y \in A,\ (x,y) \in R \vee (y,x) \in R$ *(comparability).*

An ordered set *is an ordered pair (A,R) in which R is an order on set A. In addition, if the third constraint is omitted, R is said to be a partial order on the set A.*

Let R be a binary relation and let R_1 be the *transitive extension* of R such that R_1 contains R, and moreover, if $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R_1$. Let, in general, R_{i+1} denote the transitive extension of R_i. We are now ready to define the *transitive closure* as follows.

Definition 3.1.2 *The set $R^* = \bigcup_{i \geq 0} R_i$ in which $R_0 = R$ is said to be the* transitive closure of *R.*

One of the most fundamental notions in mathematics is that of *function*. The function F can be seen as a binary relation R in which $(x,y) \in R$ if and only if $F(x) = y$. Thus, for every element x there exists at most one element y in mapping R.

Definition 3.1.3 *A set F is said to be a* function*, if F is a relation and $(a,b_1) \in F \wedge (a,b_2) \in F \rightarrow b_1 = b_2 = F(a)$. In addition, if $(a_1,b) \in F \wedge (a_2,b) \in F \rightarrow a_1 = a_2$ then F is said to be a* one-one *function.*

For a function f we often write $f : A \rightarrow B$; where $a \in A, b \in B, (a,b) \in f \rightarrow b = f(a)$. The set A is then called the *domain* and the set B is called the *range* of f.

3.1.2 Basics of Graph Theory

Graphs are natural models for many problems arising in computer science, mathematics, and engineering. Thus, some formal framework is needed to represent the relationships among data objects in graphs, determine connectivity of graphs, name crucial parts of graphs, *etc*. Thus, we give here a very brief introduction to graphs and the underlying theory. Most of these terms and notions are used further in the text in case we want to formally represent some crucial parts of our construction.

Definition 3.1.4 *Let V be a finite set, and let $E \subseteq V \times V$. Then $G = (V,E)$ is called a (directed) graph.*

Directed graphs are often called digraphs. In an undirected graph each edge in E is an unordered pair of vertices.

The *vertices* of a graph can be used for representing objects, and the *edges* for relations between these objects. A *path* is a sequence of vertices $(v_1, v_2, v_3, \ldots, v_k)$, such that $\forall v_i, v_{i+1}$ in the sequence $(v_i, v_{i+1}) \in E$.

Vertices v_1 and v_k are *connected* if there exists a path $v_1 \rightarrow v_k$ on edges $E \cup E^{-1}$. In addition, a graph is connected if every pair of its vertices is connected.

A path is said to be *simple* if all its vertices are distinct. There is, however, one exception. A *simple cycle* is said to be a simple path of at least one edge that begins and ends at the same vertex. A graph without simple cycles is *acyclic*.

3.2 Algorithms

Informally, an algorithm is a well-defined procedure that solves some well-specified computational problem. More formally, an algorithm can be defined as follows [1].

Definition 3.2.1 *An algorithm is a finite sequence A of instructions to compute function $f : I \rightarrow O$, where*

1. *$p \subseteq I$ is the set of input values,*
2. *$r \subseteq O$ is the set of output values of algorithm A, and*
3. *for all instances of p the algorithm A terminates with $r = f(p)$.*

In addition, we require that in A each instruction

1. *has a clear meaning,*
2. *can be performed with a finite amount of effort in a finite length of time.*

We shall present algorithms using a *pseudo-language* that is a combination of the constructs of some programming languages, such as Java, together with informal English statements.

In order to compute the function $f : I \rightarrow O$ efficiently, any set of input values $p \subseteq I$ should be well organized. Hence, we need a degree of ordering and structure for the elements. In general, we refer to such a structure as a *data type*.

3.3 Data Types and Dynamic Sets

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms may change over time, and are thus *dynamic*. In computer science, *dynamic sets* are special cases of *data types*. This is where dynamic sets come into the wider picture, as they are also generalizations of mathematical sets.

Each element of a dynamic set is represented by an object whose attributes can be examined and manipulated if we have a pointer to the object. Traditionally, we assume that one of the object's attributes is an identifying *key attribute*. On the other hand, we can generalize this notion by assuming that at least one or more attribute exists that forms a unique *key*. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may also have other attributes that are manipulated by the set operations. Some of these attributes have special purpose and they may contain pointers to other objects in the set. We refer to the pointer attributes as *references*.

Data type is a particularly important concept because it determines the input and output sets for an algorithm. A data type is the set of values that a variable, constant, function, or other expression may assume together with a set of operations to manipulate the type. Furthermore, *primitive data types* are built-in to a programming language. In the Java programming language, they are *boolean*, *char*, *byte*, *short*, *int*, *long*, *float*, and *double*. For example, a variable b of type *boolean* is defined as $value(b) \in \{true, false\}$. Thus, it can assume either the value *true* or the value *false*, but no other values. For integers (char, byte, int, short, long) and real numbers (float, double), we adopt the standard notation for the set of their values $\mathbb{I}$ and $\mathbb{R}$, respectively. Finally, non-primitive data types are called *Complex Data Types*, and are created by giving names to aggregates of data types. The values of the type may be primitive like integers and characters, or they may be complex, composed of an aggregations of other primitive or (recursively) complex values. In this case, it is convenient to say that a complex data type is composed of an aggregations of the types of those values. Finally, *Generic Data Types* are generalizations of Complex Data Types for which we know how to operate them but the types of the aggregates are not defined.

3.3.1 Abstract Data Types

In the previous section, we argued that algorithms are well-defined procedures that solve some computational problems. On the other hand, procedures and functions are essential tools in programming and they generalize the notion of an *operator*. Thus, we can think of an algorithm as a generalized operator of a programming language and data types as generalized *operands*.

For primitive data types, there is a set of built-in operators. For example primitive data type *int* has its built-in operators $\{+, -, *, /\}$. For an *Abstract Data Type* we can define a set of operators in a similar manner. In addition, the range of data values and associated operators should be specified independent of any particular data structure. Thus, we give the following definition.

An **Abstract Data Type** (ADT) is a set of (abstract) items with a collection of operations defined on them.

Abstract data types can be defined in terms of discrete mathematics. The idea of using formal methods is rooted in the fact that formally defined models should include only the definition and no implementation. This is essential while defining and designing abstract data types.

An example of an abstract data type is *dictionary*. The operations search(x,D), insert(x,D), and delete(x,D) may be defined as follows.

Definition 3.3.1 Dictionary *D is an abstract data type that has the operations* search, insert, *and* delete *on variables x as follows:*

1. *search(x, insert(x,D)) = x*
2. *delete(x, insert(x,D)) = D*

   Moreover, from the axioms above we can define that if the search fails the abstraction should return null value.

3. *search(x, delete(x,D)) = ∅*

3.3.2 Data Structures

In order to implement an abstract data type, we need a data structure. Thus, the abstract data type is the logical model of given physical data structure, and the data structure is the physical implementation of the data type. We adopt the definition given in [1].

A **Data Structure** is a collection of variables, possibly of several different data types connected in various ways.

Separate definitions for the logical (abstract data type) world and for the physical (data structure) world are essential for many reasons. First, it makes the distinction between design and implementation. The definition of an ADT does not specify how the data type is implemented. Implementation level details are hidden from the end user of the ADT. This hiding of the implementation details is known as *encapsulation*. Second, the concept of an ADT is an important principle used for managing complexity through abstraction. The irrelevant implementation details can be ignored in a safe way. This is also the way humans deal with complexity. We use metaphors to assign a label to an assembly of concepts and then manipulate the label in place of the assembly. Third, reusability is one of the key principles in software engineering that can be promoted by designing general purpose data structures suitable for many tasks, *i.e.*, by implementing data types suited for several algorithms.

In this thesis, we will fine tune the concept of data types in order to create a model that supports both white-box and black-box reuse in a natural way. This is done by introducing the two new abstractions which we call conceptual data type and fundamental data type.

3.3.3 Conceptual Data Types

One particular ADT can have several implementations. However, a particular *conceptual model* is needed to define an implementation.

A **Conceptual Model** is a collection of abstract items connected with (binary) relations together with a set of type constraints that lays the foundation for the implementation.

As an example, consider the concept of balanced binary search trees and especially AVL Trees. These concepts conform to the definition of dictionary in Definition 3.3.1. The nodes of any balanced binary search tree are connected with binary relations in order to satisfy Definition 3.4.5 for binary search trees discussed more thoroughly in Section 3.4. Specific implementations of this concept, however, differ by their definition of balance. Here the conceptual model of balanced binary search trees can be seen as an abstraction for the more concrete concept AVL Tree. We refer to any *instance* of such conceptual model as a *conceptual data type*.

A **Conceptual Data Type** (CDT) is an implementation for a conceptual model that encapsulates a particular ADT, *i.e.*, a CDT is a pair $C = (M, A)$ such that M and A refer to the conceptual model and abstract data type of C, respectively.

Example 3.3.1 *Let us consider the abstract data type* dictionary *again. One natural choice for the CDT in this particular abstraction is* Binary Search Tree*. On the other hand, we can choose to implement the binary search tree as an AVL tree. Thus, both implementations share the conceptual model of binary search trees. Moreover, the mutual conceptual model presumes a type constraint in which the keys are drawn from a totally ordered set of keys (keys should be comparable*[1]*).*

As we can see, one particular abstract data type can be implemented in terms of several conceptual data types. And even more precisely, these concepts produce hierarchies in which one concept can be considered to be more abstract than the other. Usually this means that one conceptual data type is defined in terms of another, as is the case in the following definition of AVL tree.

Definition 3.3.2 *An* AVL tree *is a balanced binary search tree in which the difference between the heights of the left and right subtrees for each node is* $-1 \leq d \leq 1$.

[1]We could omit the comparability, for example, by using hash tables instead of binary search trees.

The conceptual model for binary search trees is defined more thoroughly in the case study later on this thesis. However, we will illustrate the different abstractions with the following Table 3.1. The level of abstraction lowers as we go from top to bottom.

Table 3.1: *The level of abstraction lowers from top to bottom.*

ADT	dictionary	
CDT	binary search trees	hash tables
CDT	AVL tree	hashing methods

Perhaps the most natural way to define a conceptual data type, however, is to use a programming language and its syntax. For example, the *Java programming language* has its own ways to declare not only *interface types* but also *abstract classes*, which are partially design and partially implementation. Unfortunately, most object-oriented methods have very little rigor or the underlying conceptual model can be represented only implicitly. However, these methods may be informal, but many people still find them useful. The notation appeals to intuition better than formal definitions. In this thesis, we will use all of these methods to illustrate our examples. The hierarchy above, for example, could be defined in terms of *inheritance*, in which the AVL tree can be inherited from the more abstract conceptual model binary search tree. We will revisit these issues in Chapter 6.

On the other hand, a binary search tree is a special case of the even more generic data type called binary tree. However, the problem is where to put binary trees in this hierarchical model. Obviously, binary search trees can be inherited from binary trees. However, this would imply that either dictionary would have to be inherited from binary tree or vice versa, both of which will result an incorrect choice! Thus, we need another dimension to describe this relationship.

3.3.4 Fundamental Data Types

Usually, when dealing with a certain conceptual data type, we need a well-defined *data model* to describe the behavior of the data type. Portions of these models, however, can be shared among several conceptual data types. This idea of sharing leads to the abstraction called fundamental data type (FDT), as described in this section.

FDTs form the basis of abstractions that serve as “archetypes” for data models. At least most of the data models used in computer science education, could be implemented in terms of these “archetypes”. Examples of such FDTs are array, linked list, binary tree, tree, and graph.

For instance, *array* is a commonly used term in computer science to denote a contiguous block of memory locations, where each memory location stores one fixed-length, and fixed-type variable. However, an array can also refer to the FDT composed of a (homogeneous) collection of variables, each variable identified by a particular index number. Most programming languages do not support this latter form of definition, thus making the term somewhat ambiguous. Nevertheless, in this thesis, we employ this definition. From this point of view, it is possible to *implement* arrays in many different ways. However, the data model behind each implementation is common for all of the implementations.

Data Model defines the connections between the aggregate components for a generic data type.

The idea behind fundamental data types is to have implementations for the data models in such a way that there is no type constraints involved. This implies two consequences. First, we are not interested in the concrete types of FDTs, but consider them all to be generic data types. In the object-oriented programming paradigm, this kind of type definition can be seen as an interface type that has no semantics involved in it. Second, an FDT can store an element of any type. Thus, for generic FDTs, only operations for reading and writing the stored element are defined. Although this requirement is essential, it is not sufficient. Let us consider the abstract data types *stack* and *queue*, which do not have any type constraints for elements to be stored.

Definition 3.3.3 *Stack is an abstract data type in which an element may be inserted or deleted only at one end, called the top of stack. In particular, it follows the last-in-first-out (LIFO) queue discipline. The set of operations defined on this model are* push, pop, *and* isEmpty *on elements x.*

1. *push(x) – insert x into the stack,*
2. *pop() – delete and return an element from the stack, and*
3. *isEmpty() – return TRUE if the stack is empty, FALSE otherwise.*

Definition 3.3.4 *Queue is an abstract data type in which an element may be deleted only at one end, called the front, and inserted only at the other end, called the rear. In particular, it follows the first-in-first-out (FIFO) queue discipline. The set of operations defined on this model are the following.*

1. *put(x) – insert x into the queue,*
2. *get() – delete and return an element from the queue, and*
3. *isEmpty() – return TRUE if the queue is empty, FALSE otherwise.*

We do not want to view, for example, the stack as an FDT in a similar manner as, for example, the linked list. This is because of the nature of the stack (it is somewhat too abstract): it is always defined in terms of operations on that type, namely push and pop, and has its chronological partial ordering of elements stored in it (LIFO), giving it a particular semantics. Thus, some authors refer to the data types that satisfy the second constraint but not the first one as *basic data types*. We are now ready to define the concept of fundamental data type as follows.

Fundamental data type (FDT) is a generic data type that is used to implement a particular data model.

This definition has major implications. This is because any FDT may store any other FDT as the key attribute. This gives us a natural way to extend our set of FDTs, namely through *composition*[2]. In other words, FDTs can be nested for creating more complex structures. An example of a *composite FDT* is an *adjacency list*, which is the composition of an array and a linked list, where every element of an array stores a linked list as its key attribute. This kind of structure can be used, for example, to represent graphs. The nodes of the graph are enumerated, so that for any given node there exists the index *i*, and the adjacent nodes can be found from the linked list headed at array position *i*.

Furthermore, abstract data types can be defined in terms of conceptual data types, and conceptual data types in terms of fundamental data types, as we will demonstrate in the next example.

Example 3.3.2 *Let us continue the discussion of Example 3.3.1. The abstract data type dictionary was defined in terms of conceptual data type binary search tree. However, a binary search tree is a special case of even more generic concept called* binary tree*. Hence, a natural choice for the FDT for this particular concept could be an ordinary binary tree.*

As we can see, data models can be piled on top of each other to construct more abstract data types. The original abstract data type can be implemented by several

[2]Object composition is a technique in which new functionality can be obtained by assembling or *composing* objects to produce more complex functionality.

conceptual data types. On the other hand, the CDTs can be defined in terms of FDTs. The lower the abstraction is in this pile, the less semantics is involved in it. However, at some point we end up at a level where the concept includes no semantics at all.

On the other hand, while implementing a data structure we can employ inheritance but at the same time, we must turn the pile other way around. The most generic data type is at the top of the pile and the semantics grows top to bottom. Table 3.2 illustrates this phenomenon.

Table 3.2: *The level of generality lowers from top to bottom.*

FDT	binary tree	
CDT	binary search tree	array representation of binary tree
CDT	AVL tree	binary heap

Two orthogonal dimensions are identified, abstraction and generality, each illustrating a different aspect of conceptual data types. This leads us to the idea of *multiple inheritance*. The conceptual model of an AVL tree can be defined both in terms of the abstract data type dictionary and the fundamental data type binary tree. In both cases, the binary search tree behaves as a generalization of the AVL tree.

On the other hand, because some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the integers, we have a natural way to make a distinction between FDTs and CDTs. If the dynamic set has characteristics of manipulating objects which can be compared to each other, we are always dealing with a CDT, not an FDT.

3.4 Case Study: Binary Search Tree

Dictionary is an interface that characterizes the structure in which we can store, delete and retrieve elements of a certain type. Binary search tree is a conceptual data type that implements the interface dictionary. We say that the binary search tree is a *subtype* of dictionary, because its interface contains the interface of its *supertype* dictionary. There are several obvious ways to implement a binary search tree. We could simply reuse the implementation of a binary tree. Thus, we say that the binary search tree is a *subclass* of the binary tree, because its body contains the methods of its *superclass* binary tree.

Priority queue is an interface that allows at least the following two operations: *Insert*, which stores a new element into the structure, and *DeleteMin*, which removes and returns the minimum element in the priority queue. *Heap* is a conceptual data type that implements this interface. However, heaps are also binary trees, with the exception that they are complete[3].

At this point, it should be clear to the reader that both of these conceptual data types are special types of binary trees. Furthermore, if we have a binary tree construction, it should be possible to define each of these concepts separately in terms of a binary tree. However, a binary tree is also a special type of even more generic FDT *directed rooted tree*. Thus, we start building our construction by first defining the directed rooted tree.

In graph theory, a rooted tree is a connected, acyclic, and undirected graph in which one of the nodes is distinguished as the root of the tree. In order to extend this definition to directed trees, we define the directed rooted tree formally in terms of set theory as follows.

Definition 3.4.1 *Let S be a finite set, and let $R \subseteq S \times S$. Then $T = (S,R)$ is called a* directed rooted tree *rooted at (distinct) r, if $\forall a,b,x \in S$, $\exists r$,*

1. $b \neq r \leftrightarrow \exists a' \in S, (a',b) \in R$,
2. $(a,a) \notin R$,
3. $(a,b) \in R^* \rightarrow (b,a) \notin R$, *and*
4. $(a,x) \in R \wedge (b,x) \in R \rightarrow a = b$.

We use the shorthand notation $a \in T$ to denote $a \in S$, where $T = (S,R)$, and $R \subseteq S \times S$.

Especially, when dealing with the implementation of a directed rooted tree, it is obvious that the tree may possibly have some kind of ordering between its subtrees. This kind of directed rooted tree is called an *ordered tree* in which the subtrees of each node are ordered. The ordered tree is the FDT that allows hierarchical relationships between various objects in such a manner that it is either empty or contains a distinguished *root node*, and the remaining nodes form an ordered tuple of disjoint ordered trees, called *subtrees*. In particular, for a given node, the root of the nonempty subtree is called a *child node*, and the node itself is referred to as the *father* of the child node.

Definition 3.4.2 *Let $T = (S,R)$ be a directed rooted tree, and let R' be a partial order of S. Then, $T' = (S,R,R')$ is called an* ordered tree*, if $\forall s \in S$, there exists total order $P_s = \{(a,b) | a,b \in C_s\} \subseteq R'$, where $C_s = \{c | (s,c) \in R\}$.*

[3]A binary tree is complete if all possible levels are filled, with the possible exception of the bottom level, which is filled from the left to the right.

We are now almost ready to define formally a very important special case of ordered trees called the *binary tree*. The binary tree is an ordered tree in which nodes may have no more than two subtrees so that each node has *degree d* at most two. In addition, if for any node $d = 1$, the subtree is either the left or the right subtree, and this choice may not be ambiguous. This is why we need the notion of *positional tree*, to handle comprehensively the case in which some of the subtrees may be missing.

Definition 3.4.3 *Let* $T = (S,R,R')$ *be an ordered tree and let* $f\colon S \times S \rightarrow \{1,2,3,\ldots\}$ *be a* positioning function. *Then,* $T' = (S,R,f)$ *is called a* positional tree, *if*

1. $\forall (r,c) \in R\ \exists i,\ f(r,c) = i$, *and*

2. $\forall r, c_1, c_2 \in S,\ f(r,c_1) = f(r,c_2) \rightarrow c_1 = c_2$.

If no child is labeled with integer i, then the ith child of a node is *absent*. Furthermore, by a *k-ary tree* we understand any positional tree in which for each node, all children with label greater than k are absent. Thus, the binary tree is the k-ary tree with $k = 2$.

Definition 3.4.4 *Let* $T = (S,R,f)$ *be a positional tree. In addition, let us denote* $R_a = \{(a,b) | (a,b) \in R\}$. *Then* $T = (S,R,f)$ *is called a* binary tree, *if* $\forall a \in S,\ 0 \leq |R_a| \leq 2$. *The children* c_1 *and* c_2 *labeled with* $f(r,c_1) = 1$ *and* $f(r,c_2) = 2$ *are called the left and the right child of r, respectively. In addition, the subtrees,* T_1 *and* T_2*, rooted at* c_1 *and* c_2 *are called the left and the right subtrees of r, respectively.*

Extending this definition in such a way that, for all nodes in subtree T_1 the keys should be less than the minimum key of nodes in subtree T_2, gives us the model for the binary search trees. Of course, we presuppose here that the keys are drawn from a totally ordered set of elements. In the following definition, we denote the key of the node p by p_{key}.

Definition 3.4.5 *Let* $T = (S,R,f)$ *be a binary tree rooted at r and let* $T_1 = (S_1,R_1,f_1)$ *and* $T_2 = (S_2,R_2,f_2)$ *be T's left and right subtrees rooted at* r_1 *and* r_2*, respectively. Thus,* $f(r,r_1) = 1$ *and* $f(r,r_2) = 2$. *In addition, let* $<$ *be a total order of keys. Then T is called a binary search tree, and* $M = (T,<)$ *is called a binary search tree model, if*

1. $\forall x \in T_1 \forall y \in T_2,\ x_{key} < r_{key} < y_{key}$.

2. T_1 *and* T_2 *are binary search trees.*

Thus, the binary search tree is a special case for the binary tree. On the other hand, as mentioned before, the binary search tree implements the supertype *dictionary*. The dictionary is an interface that defines the operations in Definition 3.3.1.

A **binary search tree** is a conceptual data type $B = (M, A)$, where M denotes the binary search tree model and A denotes the abstract data type dictionary.

3.5 Summary

In this chapter, we have introduced the concepts abstract data type, conceptual data type, and fundamental data type. By contrast, in the previous chapter, we introduced the term algorithm visualization that raises the question of how to illustrate and allow interaction with several different kinds of data structures. The following Table 3.3 illustrates the abstract relationships among these concepts and gives a couple of examples of each.

Table 3.3: *Concepts of different levels of abstractions.*

concepts	examples
ADT	dictionary, priority queue
CDT	binary search trees, AVL tree, hash tables
FDT	array, linked list, binary tree, graph
data structure	array representation of binary tree, dynamic binary tree

Abstract data types generalize the problem domains in which we are most interested. For example, searching is one particularly important problem domain and many techniques exist to apply. Thus, the question is how to visualize the different techniques in such a way that they are as intuitive as possible. We need representations that appeal to one's own mental model of the phenomena. Here, FDTs and CDTs come into the picture. The idea is to construct a model for software visualization system that not only portrays different kinds of well known FDTs and CDTs but also allows one to interact with these visualizations and hence become familiar with the functionality these concepts encapsulate.

Chapter 4

User's Point of View

Simulation is by definition an experimental technique where the data generated by each experiment must be collected, analyzed and represented in a meaningful way [74, 77]. Similarly, any visual algorithm simulation setup constitutes a *simulation model* that is used to gain some understanding of how the corresponding system behaves. We will demonstrate the use of Matrix framework for representing fundamental data types to report and update their values in terms of *algorithm animation and simulation*. The data are gathered to illustrate the desired true characteristics of the algorithm through event procedures performed by the user. In addition, a number of visualizations will be introduced for representing the model components.

Two kinds of functionality are provided for interaction with the system. First, *control over the visualization* is required, for example, in order to adjust the amount of detail presented in the display, to navigate through large object graphs, or to control the speed and direction of animations (see *Control Interface* in Figure 1.1 on page 9). A considerable number of systems exist providing miscellaneous sets of functionality for these purposes. For example, Dynalab [15] supports the flexible animation control for executing animations forward and backward. On the other hand, Brown [18] was the first one to introduce the support for custom input data sets. These are also representatives of systems that apply one of the *bottom up visualization techniques*. However, only few systems allow conceptual visualization of the actual underlying object graph *top down* (the underlying data structure in Figure 1.2 is automatically drawn into the display through a visualization interface after each change in the structure). One such system, however, is Arma-Dino, introduced by Hill, Noble, and Potter [52], who present a technique for dynamically visualizing the structure of object graphs. Their key contribution is the use of *object ownership trees* to illustrate the objects and their inter-object references. Similarly, Matrix can also visualize the actual run-time topology of an object graph, and automatically provide layouts with different levels of details.

Second, some meaningful ways to make experiments are needed in order to explore the behavior of the underlying structure. Here, Matrix allows the user to *change the state of the underlying object graph* in terms of direct manipulation (see *algorithm simulation* in Figure 1.2). The manipulation events are targeted to the visualization and correspond to the changes mentioned earlier. Again, the display is automatically updated to match the current state after each change. Moreover, all the changes are recorded in order to be reversed through the control interface. Therefore, this second item is virtually our primary contribution and is the type of interaction we mean by algorithm simulation. Of course, both kinds of functionality are needed for exploring the underlying structure.

Another system that allows the user to simulate algorithms in this sense is Pilot [16]. However, it is targeted only to graph algorithm. Moreover, the user is only allowed to interact with some attributes of edges and vertices (*e.g.*, change the color) and not the structure itself. In the following, we introduce the four basic entities that all can be subjected to changes in Matrix application framework.

4.1 Elements of Visualization

From the user's point of view, Matrix operates on a number of *visual concepts* which include arrays, linked lists, binary trees, common trees, and graphs, as depicted in Figure 4.1. Many of the basic layouts are based on the algorithms introduced in the literature of information visualization (see, for example, Di Battista, *et al.* [8]).

Conceptual data types can be visualized using different visual concepts. For example, a heap can be represented as a binary tree or as an array. Moreover, both concepts can be visible at the same time and simulation operations performed on either of them are visualized simultaneously in the other one. This is a result of the selected connection technique and no additional effort is needed to exploit this feature.

All the visual concepts can be assembled by integrating the following four different kinds of parts into a whole. First, a *visual container* is a graphical entity that corresponds to the overall visual structure and may contain any other visual entities. Second, *visual components* are parts of visual containers that are connected to each other by *visual references*. Finally, the visual components can hold another visual container, recursively, or a *visual key*, which has no internal structure.

For example, in Figure 4.1, the Tree layout is composed of 9 visual components, each holding a visual key (denoted by letters A, N, E, X, A, M, P, L, and E). The components are connected to each other by 8 visual references to form a tree like structure

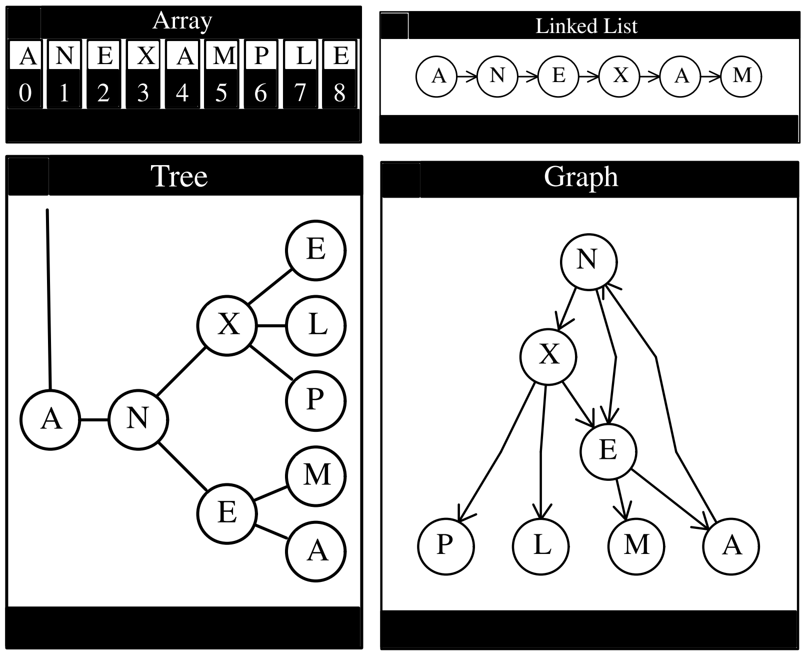


Figure 4.1: Fundamental data types, such as arrays, lists, trees, and graphs are important reusable abstractions regularly used in computer science. The figures above are printed from the Matrix system.

drawn from left to right (the default orientation, however, is top to bottom). Finally, the frame around the structure corresponds to the visual container that responds to events targeted to whole structure (for example, in a binary search tree, the user can insert new keys into the structure by drag & dropping the keys into the frame).

The user can interact with all the entities described above as far as the underlying structure allows the changes. For example, the user may insert or delete components, replace any key with another one (again, simple drag & drop of any visible key onto the target key is sufficient) or rename keys. On the other hand, the structure can similarly be changed by setting a visual reference to point to another visual component. Finally, the visual container (whole structure) can be attached to a visual component, resulting in nested visualization of two or more concepts. We will show an example of this later in this chapter (see Figure 4.3).

4.2 Algorithm Simulation

In algorithm simulation, we are interested in the detailed study of the dynamic behavior of data structures. Sequences of data structures are directly represented through event procedures. This paradigm permits the representation of systems at an essentially unlimited level of detail[1]. Simulation experiments are performed under the control of an *animator* entity with event process orientation. Model executions are guarded by an algorithm or by human interaction.

Context sensitive drag-and-drop operations provide versatile opportunities to interact on different abstraction levels. Consider a student learning basic search trees. He or she can do exercises on a binary search tree (BST) by dragging new keys into the correct leaf positions in the tree, simulating the insertion algorithm. After mastering this, one can switch to work on the conceptual level and drag the keys into the title bar of the representation. The keys are inserted into the tree by using the pre-implemented BST insertion routine. Now, one faces the question of balancing the tree. First, rotations can be studied on the detailed level by dragging edges into the new positions and by letting the system redraw the tree. Second, after mastering this, the available rotation commands can be invoked from menus directly. Finally, one can choose to work on AVL trees, create a new tree from the menu, and use the AVL-tree insertion routine to add new elements into the search tree. In addition, one can experiment on the behavior of AVL trees by creating an array of keys and by dragging the whole array into the title bar. The system inserts the keys from the array one by one into the tree using the implemented insertion routine. Moreover, the result is not a static tree, but a sequence of states of the tree between the insertions, which can be stepped back and forth to examine the working of the algorithm more closely.

An important feature providing new opportunities is that visual concepts can be nested to create complex structures, for example, adjacency lists or B-trees. Matrix does not restrict the level of nesting. Moreover, context sensitive operations can be invoked for such structures.

Object-oriented modeling seems to offer a natural way to implement these kinds of systems. The entity descriptions encapsulate the label, the structure and its behavior into one organizational unit. An object's behavior can then be illustrated by a sequence of discrete states or by a continuous process. Either the algorithm or the human controlling the model can be the dominant participant. In order to integrate all

[1]For example, Stern *et al.* [112] describe a strategy for managing content complexity in algorithm animation by stepwise refinement. She suggests a lecturing style in which she gives the basic idea first. When teaching algorithms, this means a top-down approach in which the major components of a program are blocked out first, and then progressively elaborated, until finally a working computer program is obtained in details.

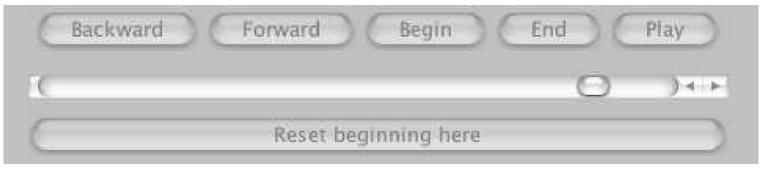


Figure 4.2: Matrix control panel.

of this, the animator must continually check for both the *state* and the *control* events. These are defined in the following.

4.2.1 Control Over the Visualization

The *control events* are trivially obtained by implementing a control panel in which the control operations are supported. The basic set of control operations in Figure 4.2 include the following list of animation control operations. The actions these operations take are obvious.

1. move one step backward
2. move one step forward
3. move to the beginning of animation
4. move to the end of animation, and
5. play animation.

Moreover, the user has several other control operations to perform. These operations influence the layout of the visualization and are implemented by most of the modern algorithm animation systems. Thus, we only summarize these in briefly.

1. *Multiple views*: User can open structure or part of it in the same or a new window.
2. *Multiple layouts*: User can change the layout of the structure.
3. *Orientation*: User can change the orientation of layout (rotate, mirror horizontally, and mirror vertically).
4. *Granularity and Elision control*: User can minimize structure or part of it; hide unnecessary details such as title, indices of arrays, direction of references, back, forward, or cross edges in graphs, and empty subtrees in trees.

5. *Labeling*: User can name or rename entities.

6. *Security*: User can disable or enable entities in order to allow or prevent them to respond a particular event.

7. *Scaling*: User can change the font type or font size.

Finally, the system provides several additional features that could be used to finalize a task. First, the current prototype implementation includes an extensive set of ready-made data structures and algorithms that could be directly summoned from the menu bar. The implemented fundamental data types are array, linked list, several tree structures, and graphs. Moreover, also included are many CDTs such as Binary Search Tree, 2-3-4-Tree, Red-Black-Tree, Digital Search Tree, Radix Search Trie, Binary Heap, AVL Tree, and Splay Tree. In addition, the prototype includes several non-empty structures that are already initialized to contain useful data for simulation purposes, for example, a set of keys in alphabetical or in random order. Second, the produced algorithm animation can be exported in several formats to other targets. For example, as a smooth SVG (Scalable Vector Graphics) animation to be embedded into a Web page or a series of still images in *TeXdraw format* [59] that could be embedded in a LaTeX source file. Matrix also supports text formats to import and automatically create object graphs from edge lists and adjacency lists. Third, the current job can be saved into the file system and loaded back into the Matrix system. Thus, the whole animation sequence, *i.e.*, the invoked operations and states of the underlying object graph, could be stored and restored by means of Java serialization.

4.2.2 State of the Underlying Object Graph

The set of *state events* may vary among the models to be simulated as with that of CDT, in which the conceptual model can be subjected to several operations. On the other hand, there exists only a limited set of *actions* in the GUI that a user may perform during the execution of a simulation. Thus, the simulation environment should be responsible for mapping these actions to particular state events. The same action might lead to a very different result depending on the model to be simulated. On the other hand, there might be a situation in which some actions are not allowed, and thus the model should be aware of this and throw an exception.

The very basic mouse driven graphical user interface contains at least the following *actions* and corresponding default events.

1. click an object – read this object; set temporary reference to point into the clicked object

2. double click an object – write[2] object into this; replace the current object at this position with the temporary object

3. drag() & drop an object – insert object into this; the target object in which the drop occurs is replaced with the dragged source object

4. popUpMenu – delete, customize, change object's representation, *etc*.

Naturally, the set of actions above could be targeted to any of the visual objects (visual container, visual component, visual reference, and visual key). Nevertheless, the functionality may differ from object to object. However, the simulation model gives a meta-level proposition for the actions as described above. We discuss this general set of actions here. Furthermore, a number of exceptions are discussed briefly.

Read and Write Operations. From the animator's point of view, the only operations we are interested in are those that in some way manipulate the data structure to be visualized. In procedural programming languages, this manipulation takes the form of *assignments*. An assignment is the operation that stores the value of an expression in a variable. Thus, a visual component is considered to be the expression above; the variable refers to some other visual components key attribute. Moreover, we may assign any FDT into the key because the data structure the visual component points to (represents) is by definition a fundamental data type.

A single *click* on an object assigns an internal reference to point into the object. Thus, the object could be rewritten to some other place by *double clicking* on the target object. The object read can be a single piece of data (datum) as well as, for example, a subtree of a binary search tree. A double click on a visual data object causes the action to be delivered to the component holding the data object. This is because the data object is considered to be the *value* of an expression, and thus cannot possibly change or replace itself. If the component similarly cannot handle the write operation, the action is forwarded to the container holding the component with the additional information where the write operation occurred (especially if the container is an array, the *array position component* invokes the write operation of the array with the additional information of the actual index where the action took place). Usually, the object read is rewritten as the key of the target object. Thus, we may end up having very complex structures if, for example, we read a binary tree and rewrite it to some position of an array and then again read this array and write it into some other tree structure. It is possible to end up with this kind of complex structures, for example, while illustrating compiler techniques, as demonstrated in the following example.

[2]The read and write operations could also be defined the other way around. However, for security reasons we have settled on this choice because, if the double click fails and the user performs single click, no harm is done since the operation does not change the structure.

Example 4.2.1 *In Figure 4.3 a set of basic blocks in linear code are illustrated graphically in the* dominator tree*. The basic blocks are derived from the code of the* binary search example *in Appendix A. The dominator tree contains the basic blocks $B_0 \dots B_6$ that are represented as the nodes of the tree. The node is decorated with a label if the corresponding block has the label $L_0 \dots L_3$. The basic block B_1 is represented in detail by nesting the corresponding linear code. The block contains two expressions that are represented in the array B_1. Furthermore, the first code line is visualized by the expression tree in the array at position* 0*. The second code line is elided by minimizing the corresponding expression tree.*

The example above can be constructed by first creating the four containers (the Dominator Tree, the array B_1, and the two expression trees). After that, the two expression trees can be read and rewritten into the array one at a time. Elision control is possible by minimizing the second tree as illustrated in the figure. Finally, the array can be read and rewritten into the node B_1 of the Dominator Tree. Note, however, that any object itself is responsible for taking care of the validity of its keys, and no new objects (other than visual objects) are created during the read and write actions.

For example, if we attempt to write some complex structure into a key of a binary search tree, it should be the binary search tree in which the exception occurs to reject the operation. Thus, the binary search tree does the type checking for all the keys inserted into the underlying binary tree. In addition, if the new key is accepted, it might be the case that the expansion of the tree (a new key was inserted either at the leaf node or as the key of an empty subtree) is triggered and thus new, possibly empty, subtrees are created. On the other hand, if the action is allowed, we do not create any duplicates of the original structure but only a new dual visualization for it. In other words, from now on, any action on the original visualization will cause the same change in the dual visualization and vice versa.

Insert Operation. The drag-and-drop action is sometimes referred as an *insert* operation. This is especially true in the case of conceptual data types such as binary search trees and heaps. Thus, the insert operation is executed by dragging and dropping the object to be inserted (source) into a container (target), for example to its label top of the frame. Thus, the target must implement a special insert method that takes the source to be inserted as an argument. This method is declared in the special interface *CDT* that indicates that the visualized structure is a conceptual data type and capable of handling such an action. This interface declares other methods, too, such as delete and search. If the target is not a CDT, the drag & drop action is interpreted as the write action.

Pointer Manipulation. The visual references have also functionalities of their own. Especially if it is implemented for the underlying structure. However, such pointer

```
LineNo  Basic Blocks in Linear Code:
        ------------------------- B0 -------------------------
   1    L0: if low >= high goto L3
        ------------------------- B1 -------------------------
   2        try = (low + high)/2
   3        if key >= table[try] goto L1
        ------------------------- B2 -------------------------
   4        high = try - 1
   5        goto L0
        ------------------------- B3 -------------------------
   6    L1: if key <= table[try] goto L2
        ------------------------- B4 -------------------------
   7        low = try + 1
   8        goto L0
        ------------------------- B5 -------------------------
   9    L2: return try
        ------------------------- B6 -------------------------
  10    L3: return low
```

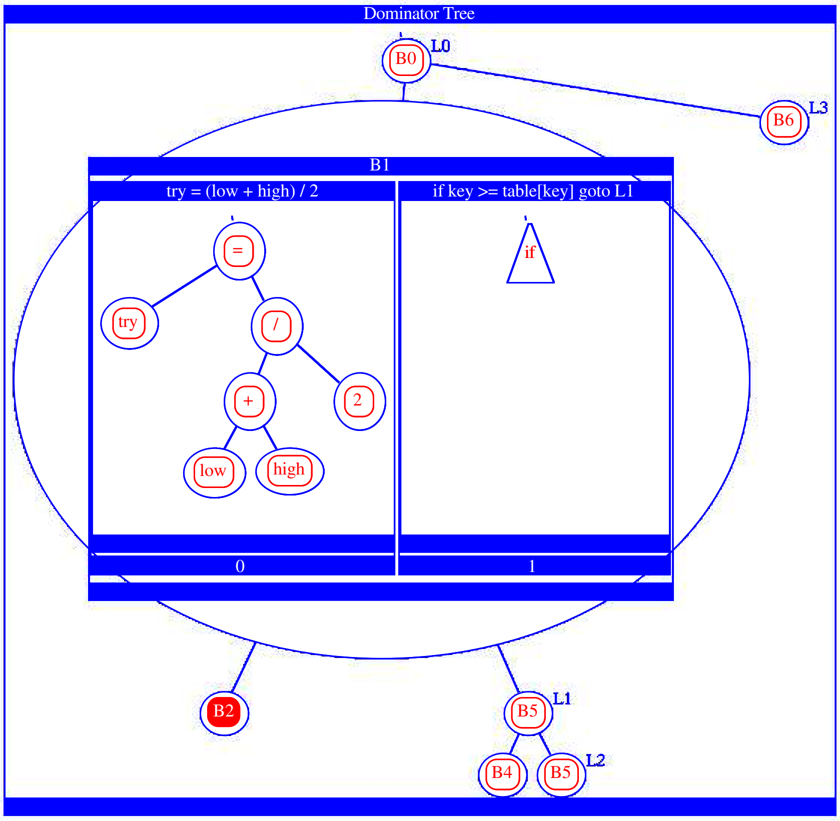


Figure 4.3: Complex composition of visual representations Array and Tree. The Basic Blocks in Linear Code are represented in the Dominator Tree. The two expressions of the Basic Block B1 are represented in the array B1. The first expression (line 2) is illustrated in the expression tree at array position 0 while the second expression tree (line 3) at array position 1 is minimized.

manipulation requires a different approach because changing a reference to point into another object may influence the whole structure to become inconsistent. Parts of the whole structure may become unreachable for a moment until a whole set of pointer operations are performed. Thus, pointer operations are queued to be invoked only when all the corresponding operations are completed. This kind of encapsulation ensures that the user can perform any combination of pointer manipulations for the underlying structure.

Instrumental Operations. The GUI can also provide several instrumental operations that help the user to perform several complex tasks more easily. For example, the user can turn the drag & drop facility to *Swap* mode, in which not only is the source structure inserted into the target structure, but also vice versa. Thus, the GUI swaps the corresponding structures without any visible temporal variables. Another example is the *insertion of multiple keys* with only one single GUI operation by dragging & dropping the whole array of keys into the title bar of the target structure visualization. The GUI inserts the keys in the array one at a time.

Other Operations. The default behavior of the drag & drop functionality can be changed. For example, it can be interconnected to the *delete* operation declared in the interface CDT. In addition, such miscellaneous operations can be declared in pop-up menu attached to all visual objects. By default, the delete operation is included there. Similarly, almost any kind of additional operations could be implemented. Thus, the actual set of possible simulation operations depend heavily on those procedures allowed by the visualizer. For example, the AVL tree visualization may include a special set of *rotate* operations to illustrate the behavior of this particular balanced search tree.

Decoration. There exist also certain predefined decoration operations in the default pop-up menu. For example, the *label* (name) of a component could be enabled, disabled and set by selecting the proper operation. Note, however, that in this case the label is not necessarily a property of the actual visualized data structure but rather a property of the visualization, and thus does not appear in every visual instance of the underlying structure.

We discuss the theme of extending simulation functionality from the visualizer's perspective even further in the next chapter.

Chapter 5

Visualizer's Point of View

The role of the visualizer is to specify the visualization for the user. However, the Matrix framework provides several ways to specify a visualization. Thus, we first look at the different possibilities before proceeding into the details.

Algorithms can be simulated by directly interacting with the GUI. This approach requires the use of ready-made data structures and the corresponding visualizations. The visualizer selects the ready-made data structure to be visualized from the menu bar, opens an existing simulation from file, or imports the structure in ASCII file format. In all of these cases, the structure is automatically visualized. The visualizer may continue to create an animation by interacting with the visualization, as explained in the previous section. Finally, the animation can be saved or exported to several file formats. However, since the role of the visualizer does not differ from that of user, we do not discuss further the simulation process here.

The problem is how to create a visualization for an arbitrary object graph with unlimited functionality. We have already mentioned one option, the ASCII file format in which the structure to be visualized can be defined in terms of *adjacency list* or *edge list*. In this case, however, the simulation functionality in Matrix (the user is able to invoke algorithms from the GUI) is limited to that provided by the system developers. In order to visualize how one's own algorithms change the object graph, the corresponding implementation must conform to a Matrix *concept interface*.

In this chapter, we discuss the ASCII file formats and describe the concept interfaces regularly applied by the visualizer. We also give an example of each. In addition, we illustrate how to apply the framework to produce arbitrary complex visualizations by nesting the visual concepts within each other. Moreover, customization issues are discussed by representing how to decorate a visualization. Finally, we summarize the chapter by giving formal definitions for some regularly used notions.

5.1 ASCII file formats

The simplest task for a visualizer is to produce a static algorithm visualization, *i.e.*, a figure for a text book or static illustration for a lecture. Of course general purpose drawing tools and formats can be applied, but they lack the ability to automate the process of creating conceptual displays directly from their definitions. For example, the TeXdraw [59] macros mentioned in the previous section (an export option in Matrix) can be considered to be such a format that the user or a tool can produce. However, creating a complex conceptual visualization (for example, a red-black tree with dozens of nodes and edges or even a directed graph) is a very time consuming process without a tool that can be automated (*i.e.* programmed) to produce valid displays.

Matrix supports several ASCII file formats to import a data structure. The following Example 5.1.1 illustrates the idea. In addition, one can continue to interact with the visualization and customize the display to be more suitable for the purpose. Moreover, the final display can be exported, for example, to TeXdraw format as mentioned in the previous chapter.

Example 5.1.1 *Figure 5.2 shows the visualization of the example graph represented as an adjacency list in Figure 5.1. The header line in Figure 5.1 indicates that the format describes an adjacency list where there is a row for each node containing the key of the node separated with a colon, and the list of adjacent nodes separated with white space.*

```
#matrix graph adjacency-list
A:B C D
B:C
C:E
D:E
E:B
F:G
G:H
H:F
```

Figure 5.1: Adjacency list representation for the graph in Figure 5.2.

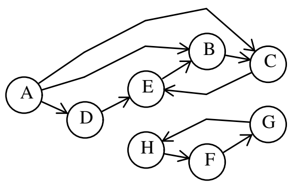


Figure 5.2: Example graph imported from the ASCII file in Figure 5.1. The graph is then exported to the TeXdraw format to be included in this paper.

One possible use for the ASCII file formats is the ability to import data generated by third party applications. The current Matrix framework, however, cannot import *animations* in ASCII format, but only visualizations as we demonstrated in the previous example. Of course, the visualization loaded from file can be animated further in terms of algorithm simulation. Nevertheless, one interesting future interest could be to combine the abilities of several animation and visualization tools that share a common import and export format. For example, JAWAA [94] has its own scripting language that is capable of determining animations. The field, however, lacks mutual agreement on using a general purpose common animation description language that many systems could support.

However, even though scripting language approach can be applied to produce highly visual algorith animations, algorithm simulation requires an additional approach as argued in the following section.

5.2 Interfaces

The first step towards algorithm animation of user made code is the ability to produce a visualization for arbitrary data structure. The following Java interfaces define methods a visualizer should implement in Matrix framework in order to gain a visualization for an arbitrary Java class instance. We have deliberately chosen not to use annotation as our primary animation technique for two main reasons. First, our aim is to provide an environment where the algorithm simulation is possible. As we can recall from Chapter 2, annotation is closely tied with event driven techniques where the visualizer annotates the key points in the program code to dispatch interesting events to change the display. However, in visual algorithm simulation we are interested in the changes in variables that the user originates during the simulation process. Thus, we want to provide a framework to support a state driven rather than event driven approach to promote this task. The interfaces seem to be a suitable approach as they provide the mapping between the data structures and its visualization. Second, annotation is an invasive method that intrinsically leads to bottom up visualization. We see, however, that the top down approach has its benefits as we can, for example, easily support context sensitive drag and drop operations illustrated in the previous chapter. Basically, our intention is to support both approaches. Again, the top down approach is something promoted by concept interfaces. The bottom up approach can be included by implementing ready-made basic data structures that conform to these concept interfaces and allow the user apply them as self animating components [48] in his or her program code.

Note, however, that the customization issues are discussed here only briefly. For example, one might argue that the tree in Figure 4.1 (on page 46) should be drawn top-to-bottom instead of left-to-right, or the integer values to be sorted in a sorting animation should always be represented as bar charts instead of numerical values. We consider these questions to be customization issues that require no interference aside from the visualizer [1]. The user (discusses in Chapter 4) may customize the display by selecting better orientation for the structure. On the other hand, if the wanted aspect is not available (*i.e.* bar charts at the current prototype), it should be the developer's task (discussed in Chapter 7) to improve the system to support the feature. For any representations here, a skilled programmer could easily construct a dual representation in which such properties are taken into account, and are thus not discussed any further.

5.2.1 FDT Interface

FDT is the superclass for all the concept interfaces and includes the following two methods.

1. public Object getElement();

2. public void setElement(Object o);

Any non-FDT element (Java Object but not an FDT) stored at this particular component can be trivially retrieved and visualized as a primitive object (string representation of the corresponding element). Usually, however, this is avoided by making use of the ready-made primitive FDTs (keys) that can be visualized in better ways. Furthermore, if the element retrieved is another non-primitive FDT, the arbitrary complex object graph can be recursively visualized as a nested structure.

The layout of the structure depends on the particular subclass the FDT implements. Moreover, the interface enables *primitive simulation* with any fundamental data type, *i.e.*, the object stored at this component can be replaced with any other FDT. Alternatively, this arbitrary complex object graph can be positioned in place of any other FDT. The corresponding structure, however, may reject the operation if other type constraints are not satisfied. We return to these subjects again in Section 6.

[1]Except in cases where the visualizer exports the outcome as a stand alone animation, for example, in SVG format in which case the decoration should be done beforehand. However, in such cases the tools available to decorate the animation are the same as those discussed in the previous section.

5.2.2 Concept Interfaces

While visualizing a piece of Java code written by a third party programmer, there are basically two approaches to visualizing it. First, the original structure can be replaced by the reference implementation provided by the framework. This also has the advantage that the structure can be fully animated back and forth without any additional coding. This approach, however, usually leads to the time consuming substitutions of several other code lines where the structure is accessed. Fortunately, the process can be automated, at least in some cases (see, for example, Jeliot [48] in which the substitution is automated and the system can replace all the basic data types with their corresponding self animating counterparts). Alternatively, the visualizer may implement all the methods the corresponding conceptual model requires and leave the original code untouched. There exist conceptual interfaces for arrays, lists, trees, and graphs. Moreover, several concept interfaces can be implemented for one single data structure, one of which is configured to be the default and chosen by the application. If the end user wishes to change the layout, he or she can do it through the graphical interface. There is no longer a need to modify or write any code. Typically, in order to conform the user made code to one of these concept interfaces, however, implementation of a couple of methods is required to allow the framework to retrieve and store elements to the particular structure.

Similar to the above, Tarraingím [91] uses collection interfaces in the SELF programming language library [116] to produce top down visualizations. We have, however, decided to introduce a concept interface hierarchy of our own in order to reduce the number of methods the visualizer must implement. This does not even deny the visualizer the opportunity to visualize standard programming language library structures. The library structure can be extended by subclassing it and implementing the corresponding concept interface. As an example of a concept interface, we will introduce the following interface for dynamic trees.

5.2.3 Example Concept Interface Tree

In Figure 5.4, we have an example of a dynamic tree that is represented by exploiting the concept of tree representation. Any object structure conforming to the following interface *Tree* can have this outcome.

1. public int getSubTreeCount();
2. public Tree[] getSubTrees();

As we can see, a node may have as many subtrees as needed without restrictions. On the other hand, a subtree must be of type tree. However, even graphs may conform

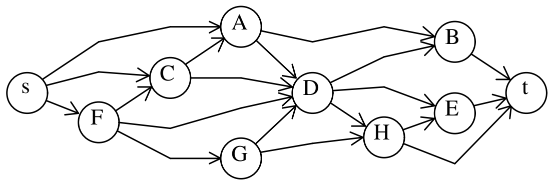


Figure 5.3: Directed acyclic graph.

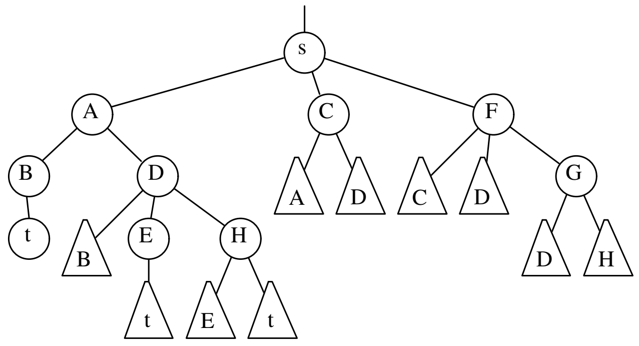


Figure 5.4: DFST-tree of the directed acyclic graph in Figure 5.3.

to this interface. The implementation of the visual tree layout in Figure 5.4 is defined in such a manner that it draws the subtrees of a given node only once. Thus, if the same node is drawn twice, its subtrees (adjacent nodes) are not drawn anymore, and thus the possible recursion ends. We call such a tree *Depth First Search Traversal Tree*. In addition, we refer to the duplicate nodes as *terminal nodes* (illustrated by triangles) and the actual nodes are referred to as *original nodes* (illustrated by circles). This has, however, additional implications. For example, the cross arcs of a graph are not drawn pointing to the actual node, but to a terminal node. This situation could be illustrated in the representation by highlighting the original node while highlighting a terminal. Obviously, terminals do not have any empty subtrees as leaf nodes may possibly[2] have.

Example 5.2.1 *Figure 5.3 shows a labeled digraph with 10 vertices and 19 arcs. Figure 5.4 shows the corresponding Depth First Search Traversal Tree (DFST-tree) using the tree representation. The graph is acyclic, and thus there are no vertices*

[2]This depends on the implementation of the actual tree.

appearing twice along any path from the root to a leaf. There are, however, cross arcs, which implies, for example, that the label D exists in several different paths. Only one of these is the internal node that refers to the original D. Others are duplicates, and are thus represented as leaves.

Usually, the visual representation of a graph makes no distinction between the label of a node and the actual data stored in it. The node is labeled by some key (usually an alphabet key). Thus, a graph may not have two distinct nodes with the same key. However, this is not always true with trees. There might be situations in which two distinct nodes of a tree may contain two different pieces of data with the same visible name. Thus, for two distinct nodes, the keys may be equal. Let us say we have the same key B in two distinct nodes b_1 and b_2 in the tree. Changing the value of b_1 does not affect the value of b_2 or vice versa.

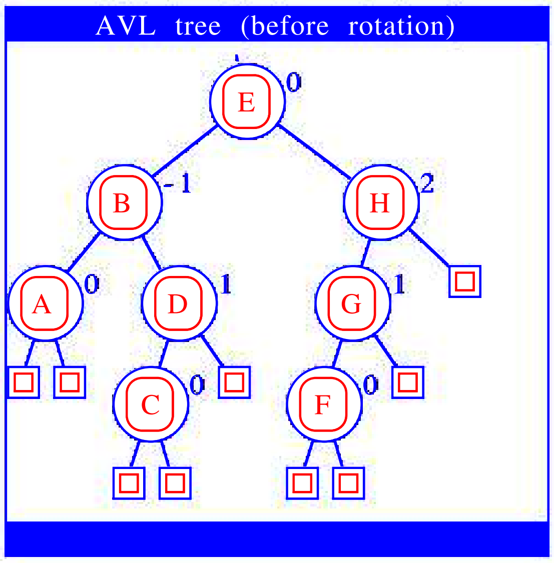
Such a situation could be avoided by labeling the keys as B_1 and B_2. This is not, however, possible in every case. Thus, clear and explicit labeling of nodes is required. This is done by decorating the representation by labels as, for example, in Figure 5.5. The AVL tree balance factor is represented as an integer (label) beside the node. The label could be any other string, too. This allows, for example, the drawing of a distinction between two separate nodes with equals keys by labeling nodes with distinct names.

5.2.4 Decoration

Before proceeding with even more complex visualizations, we briefly describe how to decorate the representation, if required. By default, this kind of implicit information is omitted. There are, however, many situations in which a special kind of decoration information is required.

Example 5.2.2 *Figure 5.5 shows an AVL tree single rotation. Node H violates the AVL balance property because its left subtree is two levels deeper than its right subtree. This is illustrated by decorating all nodes with the difference of the height of the subtrees. We can see now that in node H the balance condition* $|b(H)| = 2 > 1$ *is not valid by the Definition 3.3.2, thus violating the AVL balance property.*

Other visualizations can be decorated in a similar manner. The interface LabelDecorator is a hint for the visualizer to recognize that this particular node (or the whole tree if all nodes are type of LabelDecorator) should be decorated by labeling its nodes. If the node is not of this type, it could still be decorated by manually setting the components labeling on (popUpMenu), and renaming the node properly.



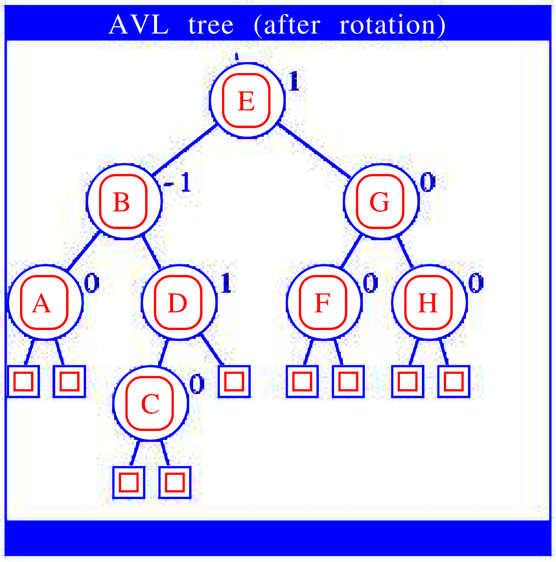


Figure 5.5: AVL tree single rotation.

Other interfaces could similarly be introduced and defined. Of course, the corresponding implementation should be added to the visualizations. The current prototype supports interfaces for labeling and coloring.

5.2.5 Simulation Interfaces

Concept interfaces for simple linear data structures (arrays, lists) include all the methods required to simulate the corresponding structure. With the hierarchical abstractions (trees, graphs) the visualization and the simulation methods are separated. Additional interfaces exist for methods that create new nodes and set adjacent nodes for a particular position in the structure. This allows one to apply the concept interfaces without more complex simulation features for visualization purposes only. However, if simulation is required, the additional methods can be implemented by conforming the target class into the corresponding *simulation interface*.

5.3 Building Higher-Level Representations

Interestingly, from now on, we could use, for example, the array representation and the list representation to produce an adjacency list representation. Similarly, we can build even more complex representations.

In the following, we consider any underlying data structure to be visualized to be a graph of V nodes and E edges. Moreover, let us denote the corresponding visual representations for arrays, lists, trees, and graphs as A, L, T, and G, respectively.

5.3.1 Adjacency List Representation

Let us assume we have any set V of nodes and a set $E \subseteq V \times V$ of edges. The adjacency list representation consists of an array $A[0 \ldots k-1]$ of $k = |V|$ lists L, one for each node in set V, and a function $f : V \to \{0,1,2,\ldots,k-1\}$. For each $u \in V$, the adjacency list $A[f(u)]$ contains all nodes v such that there is an edge $(u,v) \in E$. That is, $A[f(u)]$ consists of all nodes adjacent to u.

Example 5.3.1 *Let us suppose* $V = \{A,C,D,F,G,J,M,P,R\}$ *and* $E = \{(A,C), (A,F), (C,M), (C,D), (F,G), (F,J), (M,P), (M,R)\}$. *Then we could define* $f = \{(A,0), (C,1), (D,2), (F,3), (G,4), (J,5), (M,6), (P,7), (R,8)\}$. *In Figure 5.6 we see this construction as an adjacency list.*

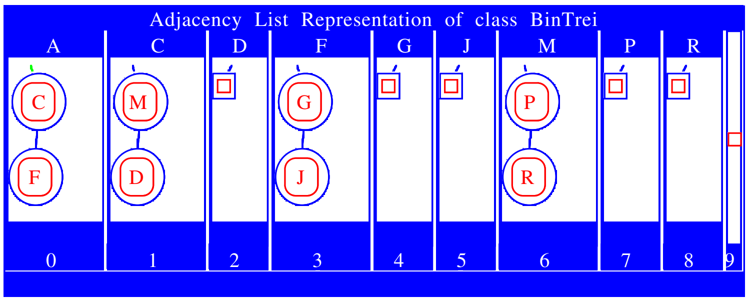


Figure 5.6: Adjacency list representation of a tree.

Thus, we see the adjacency list representation as a composition of an array representation (A) and a linked list representation (L), as illustrated in Figure 5.6. We denote this construction by $A \times L$.

5.3.2 Adjacency Matrix

Obviously, a two-dimensional array M could be represented as a composition of two one-dimensional arrays A. We call such a representation a *matrix* and denote such a construction by $M = A \times A$.

5.4 Case Study: How to Represent B-trees

A *B-tree* is a commonly used balanced search tree structure in which every node (except the root) has between $\lceil M/2 \rceil$ and M children, where $M > 1$ is a fixed integer. The root is either a leaf or has between 2 and M children.

In a *2-3-4 tree* every internal node has either 2, 3 or 4 children (maximum of 3 keys), and thus we have a B-tree of *order* four. In practice, however, much larger branching factors are typically used.

The representation of such a structure is more complex than the representation of a simple binary search tree because there must be an inner structure in the nodes to store all the keys in nodes. However, for representing the overall branching structure, a simple tree representation should be adequate. Thus, a 2-3-4 tree could be represented by the composition of a tree representation T and the array representation A for the internal keys. We denote this representation by $T^{2,3,4} = T \times A$.

The array representation, however, includes some additional information that is not needed in the case of 2-3-4 tree. For example, the title and indices of the array should be hidden while used in B-tree representation. Fortunately, this decorative

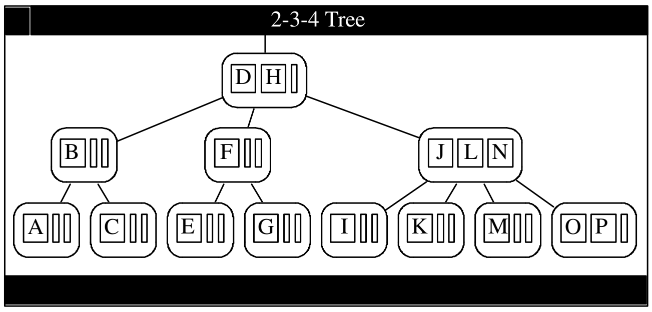


Figure 5.7: 2-3-4 tree.

information can be compressed to elide these uninteresting labels, leaving just the plain sequence of consecutive keys.

Example 5.4.1 *Figure 5.7 illustrates the 2-3-4 tree. The B-tree nodes are indicated as rounded boxes. The consecutive rectangles inside them with three value fields illustrate the data stored in the structure.*

5.5 Summary

As we have shown above, we can represent many different kinds of data structures using two carefully designed and implemented representations, namely the array representation (denoted by A_n, where n is the number of positions in the array) and by the tree representation (denoted by T_n^k, where k is the branching factor and n is the number of nodes in the tree), as summarized in the following list:

1. array A_n
2. tree T_n^k
3. linked list $L_n = T_n = T_n^1$
4. adjacency list $Adj_{n \times n'} = A_n \times L_{n' \leq n}$
5. adjacency matrix $M_{n \times n} = A_n \times A_n$
6. $n \times m$ matrix $M_{n \times m} = A_n \times A_m$
7. 2-3-4-tree $T^{2,3,4} = T^4 \times A_3$

Of course, this is not an exhaustive list of all possible representations.

Chapter 6

Programmer's Point of View

As the reader might have noticed, if we take the definitions of roles literally, we should have no point of view at all here. This is due to the fact that, in general, the programmer should have no prior knowledge of the simulation tool, making this chapter unnecessary. The visualizer can take care of all the details needed to bring the piece of code visual and to allow the user to simulate the operations defined. However, in this chapter we describe how to apply the framework for creating algorithms and data structures in terms of reuse. In addition, the theoretical point of view in Chapter 3 is revisited here, and a connection is made between the theory and the implementation. This point of view is also shared with advanced visualizers who need to gain full functionality among operations in Matrix. For example, the reverse execution of an animation and the execution history views, put into practice by many systems [15, 55, 100, 106], requires the use of special purpose components in order to work in Matrix. We call such components *animatable structures*. Moreover, because the source code always exists in any programming task, we can also look at the system from the visual testing and debugging [85, 117] point of view.

Usually, the functionality of any conceptual data type is derived from some more generic underlying concept. For example, the binary search tree could be seen as a special type of the binary tree. Having an array implementation (implicit binary tree) and the dynamic counterpart implementation (explicit binary tree) for the binary tree, a programmer could reuse these building blocks in order to implement more specific data types such as binary search trees and binary heaps. We refer to such building blocks as fundamental data types, which were discussed in the previous chapters. More generally, fundamental data types are defined for CDT implementations instead of implementing all of the CDTs from the scratch.

The challenge, however, is to determine the complete set of FDTs to satisfy all possible needs that may arise in constructing new CDTs. Although it seems to be impossible to define such a set, we argue that, in most cases, it is possible to define a suitable subset which is a collection of those FDTs needed in most cases. However, if no convenient FDT has been implemented, it should be clear that implementing one together with the actual CDT is at most as difficult a task to achieve as implementing the CDT directly. After all, while implementing a CDT, it is good programming practice to use existing reusable components, or to try to use abstractions in which reusability is taken account. We call such a discipline *reusability*.

As we will see in the following sections of this thesis, reusability has the key role of constructing new visualizable data structures. First, we use reusability for hiding the presence of the animating objects from the data structure to be animated. This is done by hiding the functionality in components called *memory structures* as discussed in Section 6.1[1]. Second, by using only memory structures, we construct more sophisticated building blocks and refer to them as *storage structures*. This is discussed further in Section 6.2.

6.1 Memory Structures

The simplest and most generic memory structure is a linear, one-dimensional array. By *virtual array* we mean a list of objects referenced by a set of consecutive integers. Every primitive data type could be seen as a special type of virtual array, namely an array s of size $|s| = 1$. However, for efficiency reasons, we have separate implementations for the primitives. On the other hand, all the other data structures can be implemented with arrays. This is easy to prove because the memory of traditional computer forms a linear one-dimensional array and all the computing is intended to be performed in this platform.

Virtual array is an implementation for a *memory structure* in Matrix. The most crucial characteristic of the memory structure is that it can store and retrieve elements of any type by simply addressing them with an integer. Note, however, that *any* integer is valid. In other words, we have neither upper nor lower bounds for the virtual array since the structure is implemented so that it allocates more memory only if there is no room for additional elements. The implementation of the virtual array is based on the idea of *dynamic tables* [29].

The following describes the *memory structure* interface. Virtual array conforms to this interface in order to be animatable.

[1]In fact, memory structures are needed for storing and playing the animation sequence back and forth as discussed in Section 7.3.

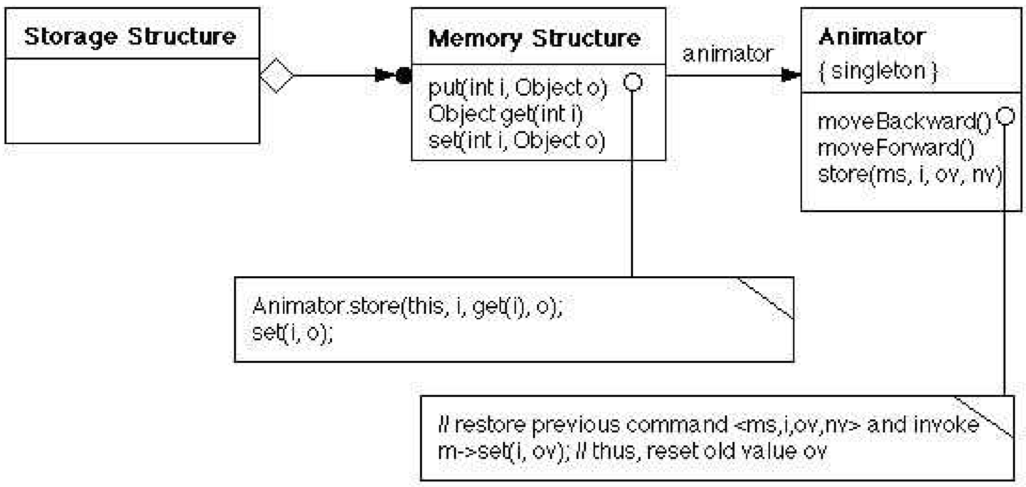


Figure 6.1: Relationships between inner object structures and the animator.

1. void put(int i, Object o); store object o into index i
2. Object get(int i); retrieve object at index i

Every put operation is registered into a special *Animator*, which could be invoked to playback (backward and forward) the instructions an algorithm performs. From now on, all (storage) structures implemented by using only this virtual array gain the animation properties for free. Figure 6.1 shows the meta-level relationships among memory structures, storage structures, and the animator. The animator object is described more thoroughly in Section 7.4 (Algorithm Animation on page 91).

Similarly, all data structures can easily implement the two methods in the memory structure interface. The only concern is to make sure the *put()* operation invokes the required *Animator.store()* method in order to be animatable. This simple procedure allows the animator full control over the structure, *i.e.*, to invoke as well as revoke all the operations as depicted in Figure 6.1.

6.2 Storage Structures

Unfortunately, implementing all the necessary CDTs using only virtual arrays as instance variables is a very time consuming process. Implementing a visualization for all these concepts is even more complex. Fortunately, most of the CDTs are based on some conceptual generalization. For example, a binary search tree could be seen as a special type of a binary tree. On the other hand, the binary tree is a special type of positional trees. Therefore, it could be convenient to implement only the most common FDTs and to use these implementations as building blocks for all the other data structures. We refer to the implementations of FDTs derived[2] from virtual arrays as *storage structures*.

The framework includes also implementations for primitive data types. For example, *virtual integer* provides animatable data type that can store an integer value. Other primitive data types are *virtual boolean* and *virtual object*. These components are typically applied to implement more complex animatable structures such as virtual array. The visualization, however, usually deals with the more complex structures and not with the primitive types directly. In the following, we focus on implementational issues concerning the most common FDTs. Primitive data types, however, are excluded.

6.2.1 Linear Storage Structures

Most programming languages include an array structure that can be indexed. We have reserved the word *array* to denote the reference implementation of such a structure as described in the previous section. The most common linear storage structure based on an array is *Table*. Table, however, has additional restrictions compared to arrays. We have adopted the C programming language style to allocate arrays indexing from zero. Similarly, implementing, for example, single *Linked List* by applying the array implementation is straightforward. Furthermore, implementation of any kind of linear storage structures is made trivial by using the virtual arrays, and thus these are not discussed here any further.

6.2.2 Trees

One of the elementary data structures is *tree*. We use the term tree for all tree-like structures and consider, for example, the binary tree and the 2-3-4-tree to be special types of trees. From the programmer's point of view, however, we would like to implement a tree structure that is reusable and, moreover, easily visualized.

We recall from Chapter 3 that a tree is a structure that allows hierarchical relationship among various objects. It was declared to be a finite set of objects, called nodes, such that tree T is either empty or it contains a distinguished node R_T, called the root node of T, and the remaining nodes of T form a tuple of disjoint (sub)trees

[2]By *derived* we mean any mechanism that uses the virtual array as the only storage for instance variables.

$< T_1, T_2, \ldots, T_N >$. In addition, for each root R_{T_i} of the subtree T_i there exists a relation between the root R_T and the R_{T_i}.

As the case study in Section 3.4 illustrated, there exist many levels of abstractions (definitions 3.4.1-3.4.5) even in such a simple CDT as a binary search tree. Four of these abstractions were presented to introduce the fundamental data type binary tree. These levels, however, could be used as the targets on our journey through the jungle of trees.

First, the last Definition 3.4.5 of the binary search tree model is far too declarative for our purposes. It requires that the keys are drawn from some totally ordered set of elements. On the other hand, the Definition 3.4.1 of the directed rooted tree does not explicitly make any difference to the order of the subtrees. Thus, it does not preserve the order of the children as would be required in most cases.

Of course, we could implement the directed rooted tree and then subclass it in order to implement an ordered tree, and then subclass it in order to implement a positional tree, and so forth. Usually, however, this is not necessary because there is nothing to implement in the middle level abstractions. However, we should keep in mind that it is not clear whether these abstractions are reserved for some special purposes in the future. Therefore proper naming of the objects and interfaces is essential.

In other words, what kind of functionality would be the most generic depends on the case. In this case, we argue that two generic objects could be considered fundamental building blocks for other data types, the positional tree, as described in Definition 3.4.3 and the binary tree, as described in Definition 3.4.4. The positional tree serves as the most generic type of trees[3] because any ordered tree could be represented in terms of positional trees. On the other hand, the binary tree is such a commonly used abstraction in computer science that there exist several different kinds of implementations for it. In particular, the array representation of the binary tree is so essential in implementing heaps that we want it to be included in our construction as a separate implementation. Terms *dynamic tree* and *static tree* are used for these two kinds of implementations for the positional tree and the array representation of the binary tree, respectively.

A dynamic tree could be implemented recursively by implementing a node to instantiate a virtual array A. The node can contain a key that is stored at $A[0]$. The subtrees are then stored at the remaining positions of A. Thus, here the binary tree is a special type of the dynamic tree in which there is at most two subtrees in each node.

[3]Note, however, that even though a binary search tree is a positional tree, which is an ordered tree, it is the case that $\mathcal{B} \subset \mathcal{P}$, and $O \subset \mathcal{P}$, where the sets of all possible binary search trees, positional trees, and ordered trees are denoted by $\mathcal{B}, \mathcal{P}$, and O, respectively.

Binary Tree implementation

One particularly interesting implementation issue with data structures to be visualized is the manner in which the different kinds of information are realized. If there is one-to-one mapping between the components of the implemented data structure and the corresponding visualization, some constructions may lead to problems.

Let us look at the definitions 3.4.2 and 3.4.4 in case we would like to implement a binary tree. The definitions suggest that the implementation should include an entity called node in which we have two references in the left and in the right subtree. We must allocate space for $2N = 2|S|$ references, where S is the set of nodes in the tree. The node without ancestors is called the root node. This kind of implementation, however, must allocate twice as much space for the references than it is necessary. This is because we must allocate space for references pointing to absent subtrees. Because every binary tree with $N \geq 1$ nodes has only $N-1$ relations, we conclude that we have allocated $2N-(N-1) = N+1$ extra space for references.

This is not, however, the only possible way to implement the binary tree. We can think of the relation between a node and its child to be a property of the child instead of the node itself. Thus, we can implement the relation R^{-1} instead of the relation R. We refer to the relations as *dad-links*. Thus, instead of allocating two references for every node we allocate only one reference for the dad-link, because we possibly cannot have more than one father for a single node.

Finally, it is possible to implement a binary tree by allocating no space at all for the references. This is because we can express the relation R implicitly as a mathematical function. Let $T = (S,R)$ and $T' = (S',R')$ be two binary trees and let additionally S' be the level order enumeration of the corresponding complete binary tree on S. This implies that there is also the corresponding mapping for relations R and R'. We can now implement the binary tree by an array A, where the root of the tree is stored in $A[1]$, and given the index i of a node, its father, the left child, and the right child can be computed as follows:

1. father(i) = $A[\lfloor i/2 \rfloor]$,
2. left(i) = $A[2i]$, and
3. right(i) = $A[2i+1]$.

Thus, the relation

$$R = \{(A[i],A[j]) | j = 2i \vee j = 2i+1, i \geq 1, j \leq |S|, \{A[i],A[j]\} \subseteq S\}.$$

Obviously, the relations with $j = 2i$ and $j = 2i+1$ are considered to be references to the left and to the right child, respectively.

Even though we did not allocate any space for relations, we noticed that they are definitely present in the structure. We refer to these kind of structures as *implicit data structures*. Structures implemented as explained in the first two cases are called *explicit data structures* because we explicitly allocate space for every relation. Thus, from the programmer's point of view we should leave all the options open for implementing a reusable data structure, and the visualization system should be prepared to take care of all the alternatives.

6.2.3 Graphs

The most generic hierarchical storage structure is a graph. Graphs are formally defined in 3.1.4 as a pair $G = (V, E)$.

A simple way to implement a dense graph is to use a two-dimensional array A. This is also known as an *adjacency matrix* in which we set

$$A[u][v] = \begin{cases} 1 & \text{if there exists } (u,v) \in E \text{ and} \\ 0 & \text{otherwise} \end{cases}$$

If the graph is sparse, a better solution would be an *adjacency list* in which for each node we keep a list of all adjacent nodes. Thus, we implement a structure in which we have an array of lists, as discussed in the next section.

6.3 Extended Structures

As the reader has noticed, the set of storage structures described here is very much the same as that introduced as the "archetypes" in Section 3.3.4 (p. 37). Only some naming conventions are changed. Thus, the previous storage structures are the basic set of building blocks needed to construct most of the elementary data structures. We summarize these building blocks and give a systematic naming conventions for them as follows:

1. Table (virtual array with upper and lower bounds i and j): $A[i \dots j]$, $|A| = (j - i + 1)$ is the size of the table

2. Linked List; explicit array implementation: $L_{EA} = A[0 \dots 2n]$, where $A[k] =$ element and $A[k+1] =$ pointer to the next index k', $k \in \{1, 3, 5, 7, \dots\}$. $A[0] =$ index of the first item in the list.

3. Linked List; explicit dynamic implementation: $L_D = A[0 \dots 1]$, where $A[0] =$ element and $A[1] =$ pointer to the next node L'.

4. Linked List (static list); implicit array implementation[4]: $L_{IA} = A$, where $A[i], i = 0 \dots n$, contains the elements.

5. Tree (k-ary tree); dynamic implementation: $T_D^k = A[0 \dots k]$, where $A[0]$ contains the element and $A[i]$ = pointer to i'th child node T', $1 \leq i \leq k$.

6. Binary Tree (static binary tree); implicit array implementation: $T_{IA}^2 = A$, where $A[1]$ = root element, $A[i]$ = element, $A[2i]$ = the left child of the element and $A[2i+1]$ = the right child of the element, $i \geq 1$.

We are now ready to implement an adjacency list and an adjacency matrix as described in the previous section. These are examples of *extended structures* that are constructed by reusing the basic building blocks described above. We start from the definition of an adjacency list.

Definition 6.3.1 *Let $A[u \dots v]$ be an array and let additionally $\mathcal{L}$ be a set of linked lists. Then, $A_L = A \circ \mathcal{L}$ is called an* adjacency list *if for each position $p \in \{u, \dots, v\}$ there exists a unique list $A[p] = L_p \in \mathcal{L}$, and for each element $e \in L_p$, $e_{key} \in \{u, \dots, v\}$.*

The definition above does not explicitly state which linked list to use (the subindex p denotes the array index and not the linked list implementation). The choice may dramatically affect the efficiency of an implemented data structure. However, algorithm analysis is out of the scope of this thesis. Therefore, we do not discuss this question any further.

As the reader may already have guessed, the definition of an adjacency matrix is very similar to the definition of the adjacency list above with the exception that here we use two arrays and no list:

Definition 6.3.2 *Let $A[u \dots v]$ be an array and let additionally $\mathcal{A}$ be a set of arrays. Then, $A^2 = A \circ \mathcal{A}$ is called an* adjacency matrix, *if for every position $u', v' \in \{u, \dots, v\}$ there exists a unique array $A[u'] = A_{u'} \in \mathcal{A}$, and an element $e = A[u'][v'] = A_{u'}[v']$, $e_{key} \in \mathbb{I}$.*

Other extended structures could be defined in a similar manner.

[4]Although some authors [104] describe this implementation it is not recommended because, in average case, insertion and removal of an element takes $\Theta(n)$ time.

6.4 Case Study: Binary Search Tree revisited

In object-oriented programming languages, the functionality of binary search trees could be modeled by defining the class binary search tree by subclassing the binary tree, as illustrated in Figure 6.2. We assume here that the keys are drawn from some totally ordered set of elements as required in Definition 3.4.5.

Besides the model, we also need the set of binary search tree operations that could be introduced as an interface type *dictionary*. From now on, it is possible to define the operations of the CDT binary search tree in terms of the FDT binary tree. For example, we could implement the operation search() by using the elements of the binary tree as follows.

Example 6.4.1

```
Define Binary Search Tree extends Binary Tree as {

  // most of the details omitted

  private
  Function find(BinaryTree bt, Element key) of type Element {
    if (bt == null) return null;
    Element e = bt.getElement();
    if (key < e)
      return find(bt.getLeftSubTree(), key);
    else if (key > e)
      return find(bt.getRightSubTree(), key);
    else
      return key;
  }

  public
  function search(Element key) of type Element {
    return find(this, key);
  }
}
```

This is, however, only one possible way for encapsulation[5]. Object-oriented pro-

[5]Some authors [105] claim that inheritance breaks encapsulation, but we argue that well-defined abstract superclasses together with proper interfaces gives a meaningful way to introduce encapsulated objects.

gramming languages, such as Java, provides additional features to support encapsulation.

Similarly, the conceptual data type heap could be defined in terms of the heap model and the set of heap operations. Here, the model refers to the binary heap subclassed from the binary tree and the set of operations are defined in the interface *priority queue*.

As we noticed, these abstractions form a hierarchy. The CDT AVL tree is a special type of the CDT binary search tree, which in turn is a special type of the ADT dictionary. There are also other CDTs that are dictionaries, e.g., balanced search trees, hash tables, splay trees, *etc.* to mention a few of them. However, these do not have to be subclassed from the binary tree.

Note, however, that the diagram in Figure 6.2 or the Example 6.4.1 shows only one possible way to construct relationships between objects. For example, delegation (see Figure 6.3 on page 76) is also a possible way to construct a binary search tree.

In the following sections, we are not restricted to any specific design choice, even though we still use some examples where these kinds of assumptions must be explicitly made visible. Thus, it should be kept in mind that the framework represented here has no restrictions concerning object-oriented design methods.

Reusability Revisited

One of the key ideas of introducing the FDTs is behind the fact that well-defined FDTs may serve as reusable building blocks for more advanced data types. We suggested earlier that an FDT could be inherited by a CDT. Thus the implementation of the CDT is defined in terms of the FDT. We call such a reuse *white-box* reuse. This is not, however, the only possible mechanism for reusing objects.

The other common technique for reusing functionality in object-oriented systems is object *composition*, as discussed earlier, in the context of composite FDTs. By object composition we mean a technique in which functionality is obtained by assembling objects to produce more complex functionality. This style of reuse is called *black-box* reuse. The term black-box refers to visibility: no internal details of objects are visible, but they appear only as “black boxes”. The same mechanism could also be used for obtaining the functionality of CDTs by assembling FDTs.

Example 6.4.2 *Instead of subclassing a binary search tree (BST) from a binary tree (BT) (as in Figure 6.2) a binary search tree might black-box reuse the functionality of a binary tree. This is illustrated in Figure 6.3 in which the diagram shows the binary search tree delegating its getElement() operation to the binary tree instance.*

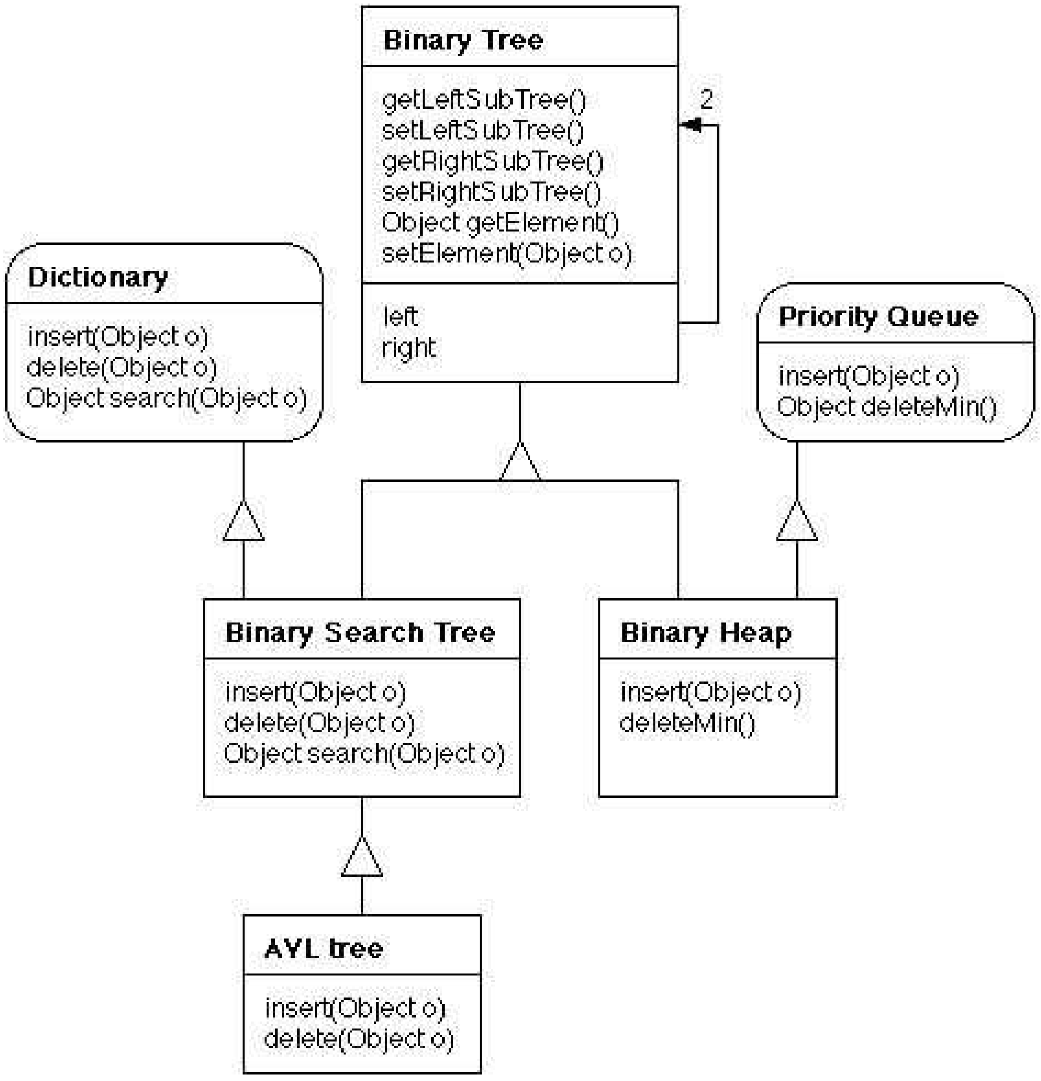


Figure 6.2: Hierarchy of abstractions.

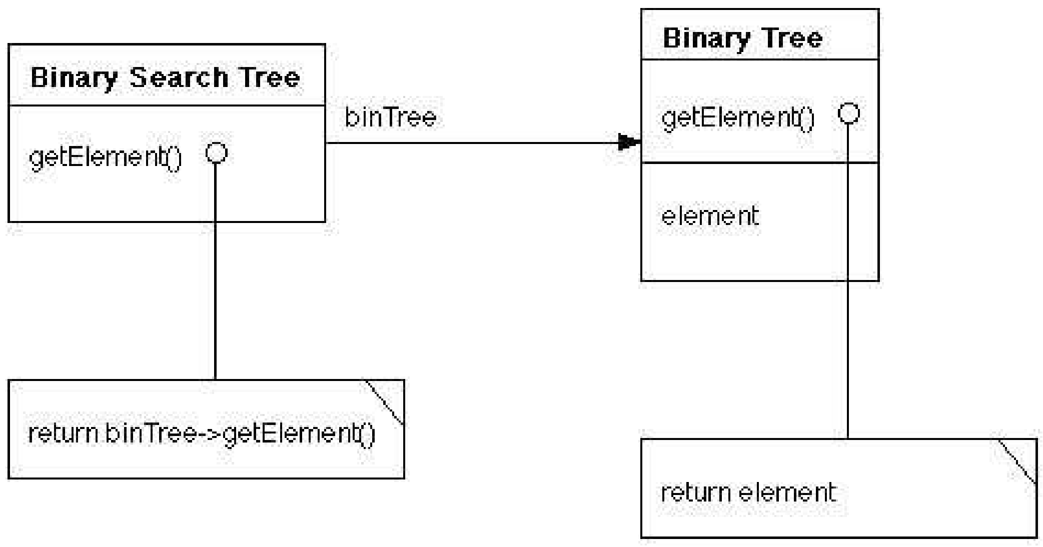


Figure 6.3: Delegation of binary search tree getElement() method to a binary tree instance.

This technique is generally called *delegation*, in which two objects are involved in handling a request in such a way that the receiving object (BST) delegates operations to its delegate (BT). In other words, instead of a binary search tree *being* a binary tree, it would *have* a binary tree. Note, however, that binary search trees must in this case forward all requests to their binary tree, whereas before it would have inherited those operations. However, as we saw in Example 6.4.1 the inheritance has very little to offer, making delegation a very attractive choice.

Finally, CDTs can also be defined in terms of other CDTs, which is also an example of black-box reuse. Thus, we may end up with arbitrary complex data structures. Of course, the visualization system must have a way to visualize these arbitrary complex structures. This is basically the motivation for the design of Matrix, in which we allow arbitrary complex visualization in terms of nested visualization. As an example, let us consider stack and queue, *i.e.*, Definitions 3.3.3 and 3.3.4 again. We can implement all operations of queue with two stacks as illustrated in Figure 6.4. This is an example how the level of abstraction can be extended further. Thus, in other words, there are no limitations of how complex structures might be.

Visual Testing and Debugging

One particularly interesting way to apply the framework is to use it for visual testing and debugging purposes. Let us consider the case of a programmer implementing an

```
Define Queue as {
  Stack s1, s2;

  Procedure Put(Element e) {
    s1.push(e);
  }

  Function Get(Element e) of type Element {
    if (s2.isEmpty())
      while (not s1.isEmpty())
        s2.push(s1.pop());
    return s2.pop();
  }
}
```

Figure 6.4: Implementation of a queue by using two stacks.

AVL tree that extends the binary search tree described before. Here, the system itself can be applied to test the new data structure in terms of algorithm simulation. New test cases can be determined simply by drag & dropping keys into the balanced search tree.

Let us assume the newly implemented structure has an error that prevents the structure to balance itself. The visualization immediately implies such an error because it is easy to see that the structure is unbalanced. Some other testing methods may not notice such an error because the structure perfectly behaves as a search tree, however, making it slow down, if the data inserted is somewhat degenerated.

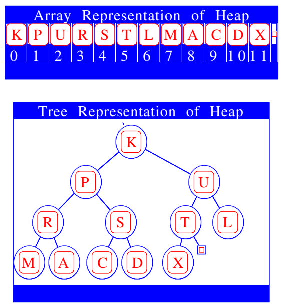
The system could also be used for debugging purposes. The user could spot the error from the source code after it is detected in terms of visual testing. However, it is one of our future challenges to incorporate a visual debugger into the framework.

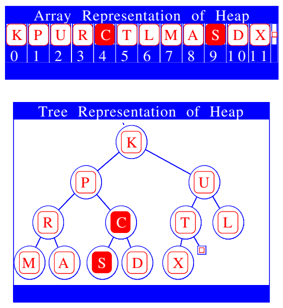
6.5 Implementation of Algorithm Simulation Exercises

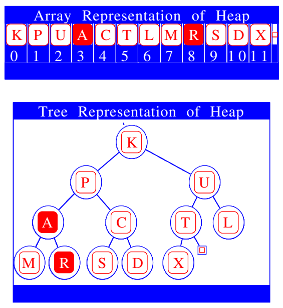
In the following, we will show how to apply the framework for one particular application: an algorithm simulation exercises.

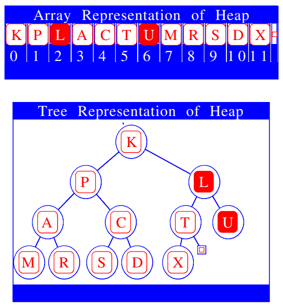
The idea is to provide a set up in which the student has the assignment and tools needed to solve the exercise. The assignment could be, for example, “show the result of using a linear-time algorithm to build a binary heap using the input K, P, U, R, S,

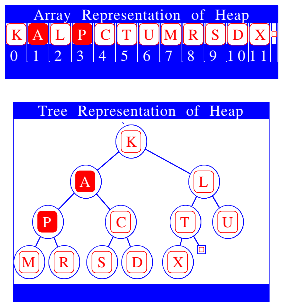
Table 6.1: *Snapshots of binary heap animation.*

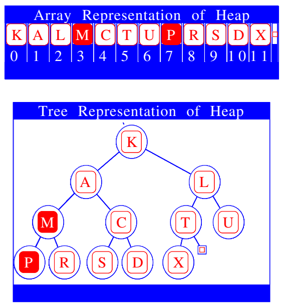


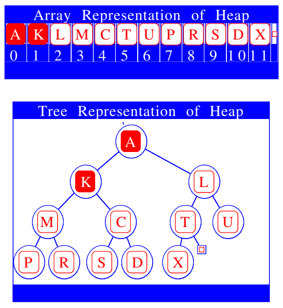


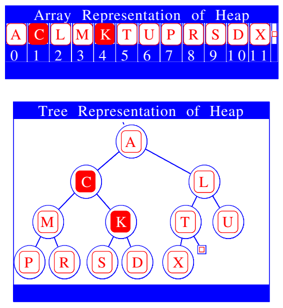


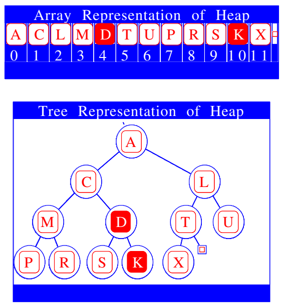












T, L, M, A, C, D, and X". The tools needed for this exercise include two FDTs. The first one is the *array* for the input keys and the second structure is the *binary tree* that is used for simulation of the *buildheap* algorithm.

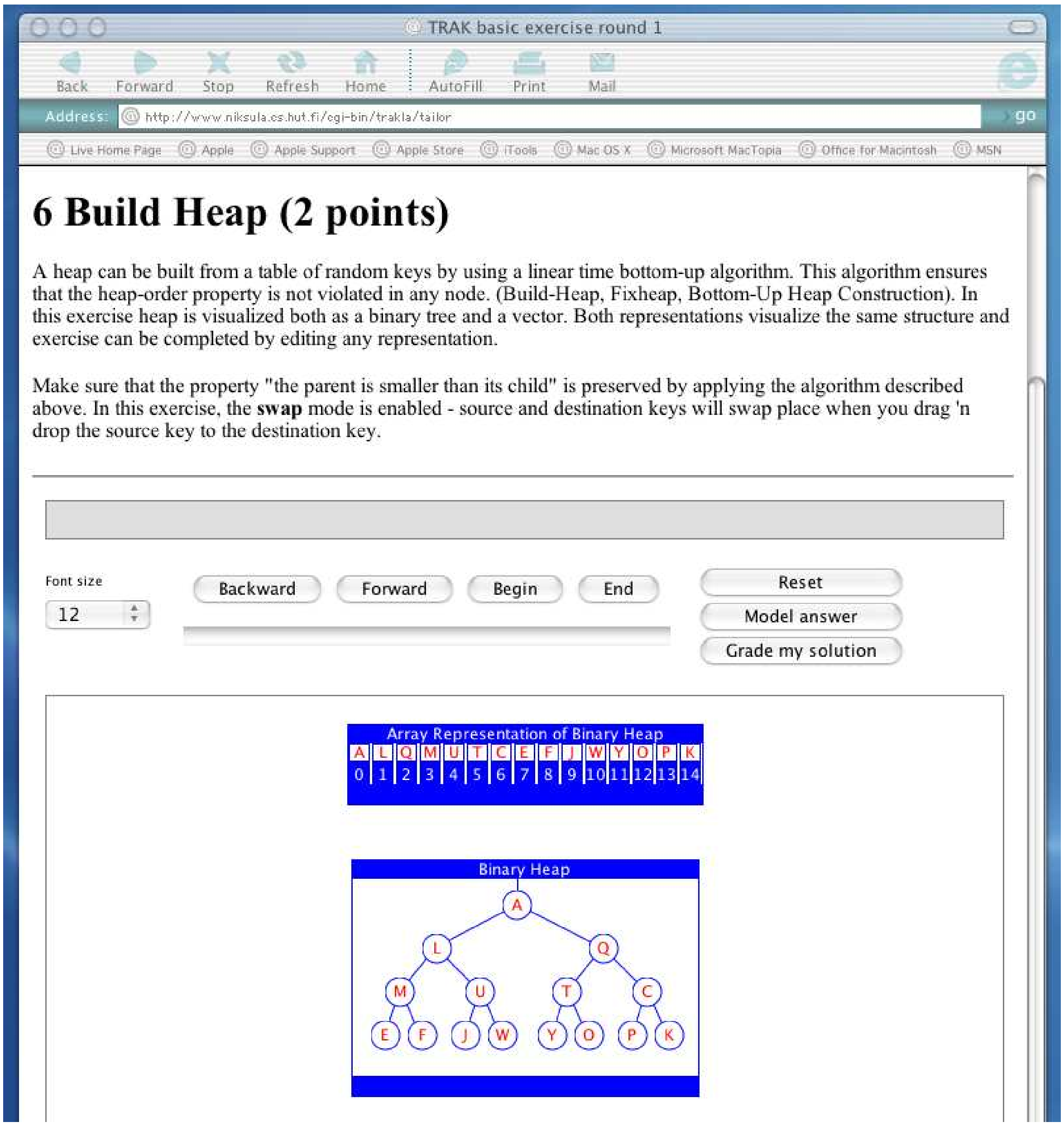
In order to solve the exercise, the student drag and drops the keys into the heap and perform the swap operations illustrated in Table 6.1. Thus, the student simulates the given algorithm in order to come up with a solution. After completing the exercise, the student submits his or her answer for automatic assessment.

As we already have an algorithm to solve the exercise (the one used to create the original animation sequence in Table 6.1), it is also possible to create the model solution for the exercise. In fact, it is possible to compare the model solution to the sequence submitted by the student. Every single state in the submitted sequence can be compared to the corresponding state in the model solution. If the two states differ from each other, a failure is reported to the student.

From the technical point of view, an exercise is a class that conforms to the particular *Exercise* interface. Basically, in order to implement the interface, one needs to define a couple of important methods. First, one needs a method to determine all the data structures and their names for the visualization during the simulation of the exercise algorithm. These data structures are expected to be fundamental data types and ready-made visualizations for them should exist. For the buildheap exercise, only a simple array is required. Second, a method to solve the exercise is needed when automatic assessment takes place or the model solution is otherwise requested. In the example above, the solve method is equal to *Buildheap* algorithm. Of course, the assignment text and a couple of initiating methods (such that fills in the array with new random data) are needed as well.

From now on, the framework is capable of visualizing the exercise by showing the assignment in some text window and by illustrating the FDTs needed to simulate the algorithm. By invoking the solve method, it is also capable of determining the model solution for the exercise. This model solution is then automatically compared to the submitted answer. The assessing procedure then gives some feedback to the student on his or her performance. An implementation of such an exercise in actual learning environment[6] is shown in Figure 6.5.

[6]at http://www.cs.hut.fi/Research/TRAKLA2/



6 Build Heap (2 points)

A heap can be built from a table of random keys by using a linear time bottom-up algorithm. This algorithm ensures that the heap-order property is not violated in any node. (Build-Heap, Fixheap, Bottom-Up Heap Construction). In this exercise heap is visualized both as a binary tree and a vector. Both representations visualize the same structure and exercise can be completed by editing any representation.

Make sure that the property "the parent is smaller than its child" is preserved by applying the algorithm described above. In this exercise, the **swap** mode is enabled - source and destination keys will swap place when you drag 'n drop the source key to the destination key.

A	L	Q	M	U	T	C	E	F	J	W	Y	O	P	K
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figure 6.5: Browser snapshot of a TRAKLA2 exercise.

Chapter 7

Developer's Point of View

7.1 Overview

In this chapter, we describe the technical framework of the Matrix system. The overall goal is to design and implement a system capable of

1. visualization,
2. animation,
3. simulation, and
4. automatic assessment of exercises

in the field of data structures and algorithms.

In order to achieve all these four goals, we must develop techniques for handling the complexity of this kind of system. The first step is to introduce the formal framework for data structures and algorithms as discussed in Chapter 3. We depend heavily on those different kinds of levels of abstractions for developing data structures and appropriate visual objects for representing them. These abstractions are important because this is how we are going to make the distinction between the actual data structure to be represented (discussed in the previous chapters) and the set of visualizing components illustrating the state of the structure.

We needed a principled technique for managing the animation process. The challenge was to find a meaningful way to hide the presence of the animating objects from the actual data structure. This was achieved by introducing two new abstractions called memory structure and storage structure. In addition, a special purpose component called *Animator* was introduced in order to register all operations performed in the environment.

From now on, we need a visualization for every storage structure to create software visualization and to allow algorithm simulation. The underlying concepts are based on the *simulation model*. Finally, in the next chapter, we give an overview how to apply this framework for educational purposes. Among other examples, we discuss how to use the framework for automatic assessment of exercises.

This chapter is organized as follows. In Section 7.2 we will revisit the theoretical framework and present the simulation model needed in the rest of this chapter. In Section 7.3 we introduce the technique, suggested in this thesis, for representing data structures by implementing visualization for each predefined fundamental data type, and using these FDT visualizations as basic building blocks for more complex representations. Section 7.4 summarizes the framework developed so far by briefly discussing the *algorithm animation and simulation* in terms of the designed framework. Finally, in Section 7.5 we summarize the discussion as a whole.

7.2 Simulation Model

Here, we will present a new formal construction called the *simulation model*. The framework is used to implement the visual components necessary for visualizing FDTs, to introduce an algorithm animation engine, and to define the simulation behavior.

7.2.1 Theoretical Framework Revisited

The numerical values of a given data type could be thought of as the numerical data to be visualized. Visualization of any given primitive data type is trivial[1], because these already have a range of numerical values. The challenge, however, is to find a practical model to visualize more complex data types.

If we consider only primitive data types, such as *boolean*, *integer* or *string*, the creation of their visualization is straightforward. We refer to their visual counterparts as *visual booleans*, *visual integers* and *visual strings*, respectively. These visual primitive types serve as the basis of our approach.

[1]Of course, many authors visualize even an integer type in several possible ways. We argue, however, that question of *how* a given primitive type should be illustrated is a customization issue, not a visualization issue.

Definition 7.2.1 *Primitive visual data type T_P is a visualization for an instance of primitive data type $P \in \bigcup \mathcal{N}$, where $\mathcal{N}$ is the set of primitive data types built into a programming language.*

Let $\mathcal{P}$ denote the set of all primitive visual data types. In addition, let $\mathcal{T}$ denote the set of all visual data types. Thus, $\mathcal{P} \subseteq \mathcal{T}$.

From the simulation point of view, visual primitive data types behave like tokens that could be delivered from one point to another. On the other hand, this kind of single delivery could be considered an *instruction*, as described in Definition 3.2.1 (the definition of an *Algorithm* on page 33).

In order to visualize more complex data types such as *arrays*, *lists*, *trees*, *etc.* we need a well defined model that must be rich enough in structure to mirror actual relationships of physical data structures in the real world. On the other hand, it should be simple enough so that we can effectively define new representations when necessary.

We propose the following definition, in which we refer to the visualizations of data structures as *visual containers*. Visual containers are labels for complex structures consisting of variables connected in a specific way. For each variable in structure, there exists a *visual component* that is capable of visualizing the variable. The visual container is responsible for determining in what position or order to put all of its components. In addition, the container may also include a set of *visual references*. A visual reference is a binary relation between two visual components explicitly shown on the visualization.

However, the variable above does not have to be a primitive data type, it may as well be some other data structure. Thus, visual components may be representations for some more complex data structures forming a nested hierarchy of visualizations.

Definition 7.2.2 *Let $\mathcal{T}$ denote the set of visual data types. Then $T \in \mathcal{T}$ is a visual data type, if $T = (C, N, R)$, where*

1. visual container $C = N \cup R$,
2. *Set of* visual components $N \subseteq \mathcal{T}$, *and*
3. *Set of* visual references $R \subseteq N \times N$.

In addition, any part of a visual data type X may have *attributes*, say Y_i. We denote such attributes by X_{Y_i}, where $X \in \{C, N, R\}$. In particular, each visual component $a \in N$ has its *key attribute*, denoted by N_a. The key attribute is the reference into the visual data type $T \in \mathcal{T}$ which the visual component is representing.

First order data structures have a container $C = N \cup R$ in which all visual components N are visual primitives types.

Example 7.2.1 *Let $M = (T, <)$ be the conceptual data type AVL tree (binary search tree) in which we have a binary tree T, and a total order $<$ for the keys stored into the structure. In addition, let $T = (S, P, f)$ be a binary tree rooted at p_{root} in which S is the set of nodes, P is the set of pointers, and f is the positioning function.*

In order to visualize the AVL tree, we need a container $C = N \cup R$ in which the set N of visual components are of type visual node. *Any visual node $a \in N$ is pointing to some AVL tree node $s \in S$ and for each relation $p = (v, u) \in P$ there exists a* visual reference *$r = (a, b) \in R$ connecting visual nodes $a, b \in N$. In addition, for every node $s \in S$ the key s_{key} should be a primitive type, e.g., a string that could be illustrated as a primitive visual data type.*

Figure 5.5 on page 61 illustrates two snapshots of such a visualization. Containers titled "AVL tree..." include all the nodes and references that forms the tree layout for the underlying binary tree T. In addition, each node contains a key value that is the data stored in the AVL tree M.

The key, however, could be some more complex type, too. For example, we may have a visual array of type visual list in which the list itself is a container of, let us say, nodes containing visual primitive keys. Thus, we are dealing with a *second order* data structure. Similarly, we could cope with more complex *k'th order* data structures.

The model also includes non-key *attributes* in order to customize the visualization. For example, a node may have a coloring to illustrate some data structure specific property, or a reference may contain a label to indicate the weight of an edge in a graph.

7.2.2 The Model

So far, we have defined the term visual data type and described its outline. Four kinds of entities were required in order to illustrate any data structure concept: containers, components, references, and primitives. We argue that this sketch is rich enough to be able to simulate any data structure known so far. This is because of we can visualize any simple data structure defined in compile time (*i.e.*, primitives and static arrays) and also those of which are created run-time (*i.e.* in terms of dynamic memory allocation). Moreover, the visualization is able to visualize any combination of these in terms of nested visualizations.

Usually, a simulation model is a *metaphor* for a well-known abstraction. Thus, the models are labeled as *Tree* or *Heap* because it is suggested that they have the qualities of trees or heaps, respectively. Or the model is an *analogy* and resembles some other abstraction (e.g. Array or Graph).

We are now ready to give the definition for the simulation model.

> A **simulation model** is the implementation of a visual data type which simulates the behavior of the underlying data structure.

Generally, it is impossible to say which of these visual abstractions gives the true idea of the object structure. Thus, it should be admitted that the structure may have many possible simulation models to illustrate its behavior. It is the visualizer's decision as to which metaphor(s) to apply. Moreover, it is the user's task to choose from the set of implemented conceptual representations. It may be even possible to have several visualizations active at the same time.

7.3 Data Structure Visualization

The simulation model, discussed in the previous sections, defined the visual components needed to visualize a conceptual data type. It also laid the foundation for algorithm animation and simulation. In this section, we will discuss the visualization of FDTs. We start by briefly describing some technical details how the framework could be applied for visualizing a single state of a data structure. After that, we describe the overall design and architecture of the system. We also define the most common FDTs in terms of the simulation model.

7.3.1 Visual Components

As we can see from the previous section, the simulation model consists of four different kinds of entities, namely the visual containers, visual components, visual references, and visual data. To prevent misunderstanding, we use the names container, component, reference, and data, respectively. The last one refers to the set of primitive visual data types.

The container represents the overall structure and consists of a set of components and references, as already illustrated in Example 7.2.1. A component may have a key, which in turn could be either another container or a piece of data (primitive data type). Thus, there is no restrictions for the complexity of a visual representation.

The Definition 7.2.2 additionally included a special set of *attributes* that could be attached to an object. From the visual point of view, these attributes behave like additional responsibilities an object may have. Thus, visualization should also include various mechanisms to decorate objects with additional information. The title of a container is an example of an additional label. Other examples could be the coloring of a graph or a tree, the balancing information of a balanced tree, the weights of a graph *etc*. This kind of *decoration* could be included into an object in several ways. First, we could set a *label* for any object. This label is then shown as the name of the object. For example, the balancing information for each node in Figure 5.5 illustrates such a label. Second, coloring is another possible method to decorate an object. Thus, also the *color* could be set depending on the state of the object.

Implementation

As discussed above, the system and all of its visual objects should support several methods to function properly. The top level design is illustrated in Figure 7.1 together with an example of object structure relationships of the VisualTree.

All components described in definitions 7.2.1 and 7.2.2 are inherited from the abstract superclass VisualType. The common functionality shared among these objects is implemented there. For example, the decoration functions *set label* and *set color* are defined as general methods (setName() and setColor(), respectively) for any kind of visual object because these are methods of the VisualType; all other visual objects are the subclasses of it. The VisualType also implements many other features often needed to implement a concrete visual type. These include methods for decorating, debugging, component management, pop-up menu handling, basic simulation functionality such as drag & dropping of components, customization of behavior, drawing tools, printing, and data structure repository. The repository is a storage that contains all data structures represented at some particular moment. This storage is needed for determining whether a structure is already shown in some place on the display or not. This is essential, for example, in order to prevent infinite loops in tree representations.

Typically, visual objects do not inherit the VisualType directly, but by inheriting either the class VisualContainer or the VisualComponent. Obviously, all containers and components are subclassed from these two classes, as well. Only references and primitives are directly inherited from the VisualType. The idea of subclassing containers and components from special superclasses is rooted in the fact that these objects usually share a lot of functionality. Thus, default functionality can be implemented in these superclasses for every container and component. This kind of default behavior can be, for example, the default coloring of containers and objects or component management inside a container.

The diagram in Figure 7.1 shows a model that contains the objects and relations of the *VisualTree* object structure. An example of this kind of structure is illustrated in Figure 5.5.

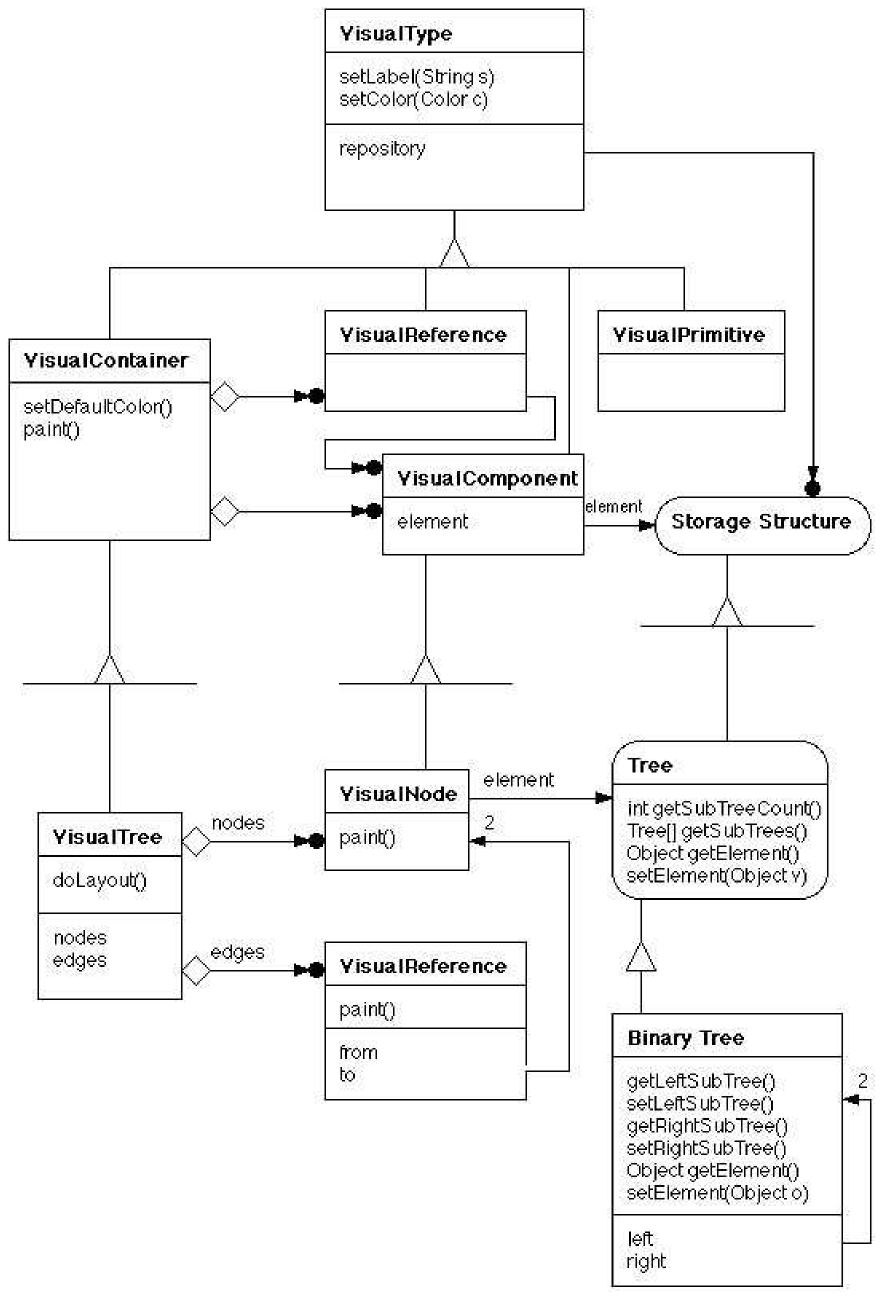


Figure 7.1: Meta-level simulation model and the design of VisualTree simulation model.

Example 7.3.1 *The* VisualTree *in Figure 7.1 and its* VisualNodes *are subclassed from the* VisualContainer *and* VisualComponent*, as described above. The* VisualTree *also includes a set of* VisualReferences*. The* VisualReference *is an object that takes two nodes and creates an edge between them.* VisualTrees *are graphical representations for some real* Trees *such as* Binary Tree*. VisualNodes also contain an* element *that is a* storage structure*. Thus, the elements may be complex data structures or primitive data types.*

7.3.2 Ready-Made Visualization Concept

The ready-made visualization concept refers to an idea in which the system will offer a set of ready-made visual containers for a set of FDTs. These ready-made visualizations could be nested to produce more complex visualizations. Thus, the ready-made visual container also behaves like the corresponding FDT or its implementation (storage structure) in which composition was introduced for the first time.

Of course, we must also have a set of ready-made primitive types. In this thesis, we will use only strings as key attributes. Therefore, we introduce only one visual primitive type called the *VisualKey*.

Currently, we have visualizations for arrays, lists, trees, and graphs. We will refer to these visualizations as *VisualArray*, *VisualList*, *VisualTree*, and *VisualGraph*, respectively.

If we would like to implement a new CDT, it should be clear that reusing one of the storage structures implies that we could animate the structure if we only had a visualization. Thus, the concept of ready-made visualization should also include the idea of having a ready-made visualization for every storage structure introduced in the previous chapter. In fact, this is exactly the case. The reuse of the storage structures gives us the benefits not only of reusability, but also of visualizability. In other words, the visualization does not know the actual subtype of the structure that it represents. It only knows the supertype (storage structure), which is actually enough[2]. Thus, a single visualization may represent many possible storage structures.

Example 7.3.2 *Let us consider the following problem. We would like to implement two new search trees, let us say a splay-tree and an AVL tree, both of which could be implemented by reusing an explicit binary tree. We could, for example, subclass the explicit binary tree, as done in Figure 7.2. The AVL tree may additionally reuse the binary search tree functionality by not subclassing the binary tree directly, but*

[2]Enough with respect to FDTs. Of course, we cannot possibly cover all details a data structure may have. This is, however, possibly fixed by decoration, as discussed in Section 7.3.1.

by subclassing the binary search tree. However, while doing so, we also subtype the interfaces Array and Tree. From now on, the system is capable of illustrating both of our new search trees as VisualTrees, since they both implement the interface Tree required for this visualization. The same holds for the Array representation. Thus, we gain the possibility to illustrate instances of our new search trees graphically either as an array or as a tree, without concern about the implementational details, because the storage structure binary tree already gives an implementation for these interfaces.

Moreover, because we now have separate implementations for storage structures and their visualizations, it should be possible to allow a storage structure to have many visualizations to choose from. The following example illustrates our approach of having many possible visualizations for a single storage structure.

Example 7.3.3 *The static binary tree implementation can be used to construct a binary heap. There are, however, two possible representations for this kind of structure, namely the explicit binary tree representation and the (implicit) array representation of the heap. Both of these interfaces are implemented by the static binary tree, and both are represented in Figure 7.3. It should be noted, however, that both of these representations in the figure illustrate the same storage structure. Thus, making an update for one representation also brings about the corresponding change to the other.*

As we can see, visualizability could be achieved by implementing an interface. Thus, every visual representation should have a similar interface to that of the visual tree representation. This also implies that we could implement this interface without reusing any existing containers. It should be noted, however, that doing so leaves us the responsibility to use the memory structure in order to be able to animate the visualized structure. On the other hand, if animation is not required, it should be adequate to implement only the visual interface, and thus have a proper visualization without any possibility to animate nor simulate[3] the structure. This kind of animation is quite a close to *visual debugging*, as some authors [85] prefer to call it.

Implementation of visualization

As we can imagine, implementation of visual representations should be done with great care and discipline. For example, the visual tree representation described above is designed so that it does not enter an infinite loop even if one tries to use it with graphs. In fact, we argue that this could be considered a valid case where the proper representation is the DFST-tree of the graph, as illustrated in Figures 5.3 and 5.4. Thus,

[3]Simulation constraints are discussed later.

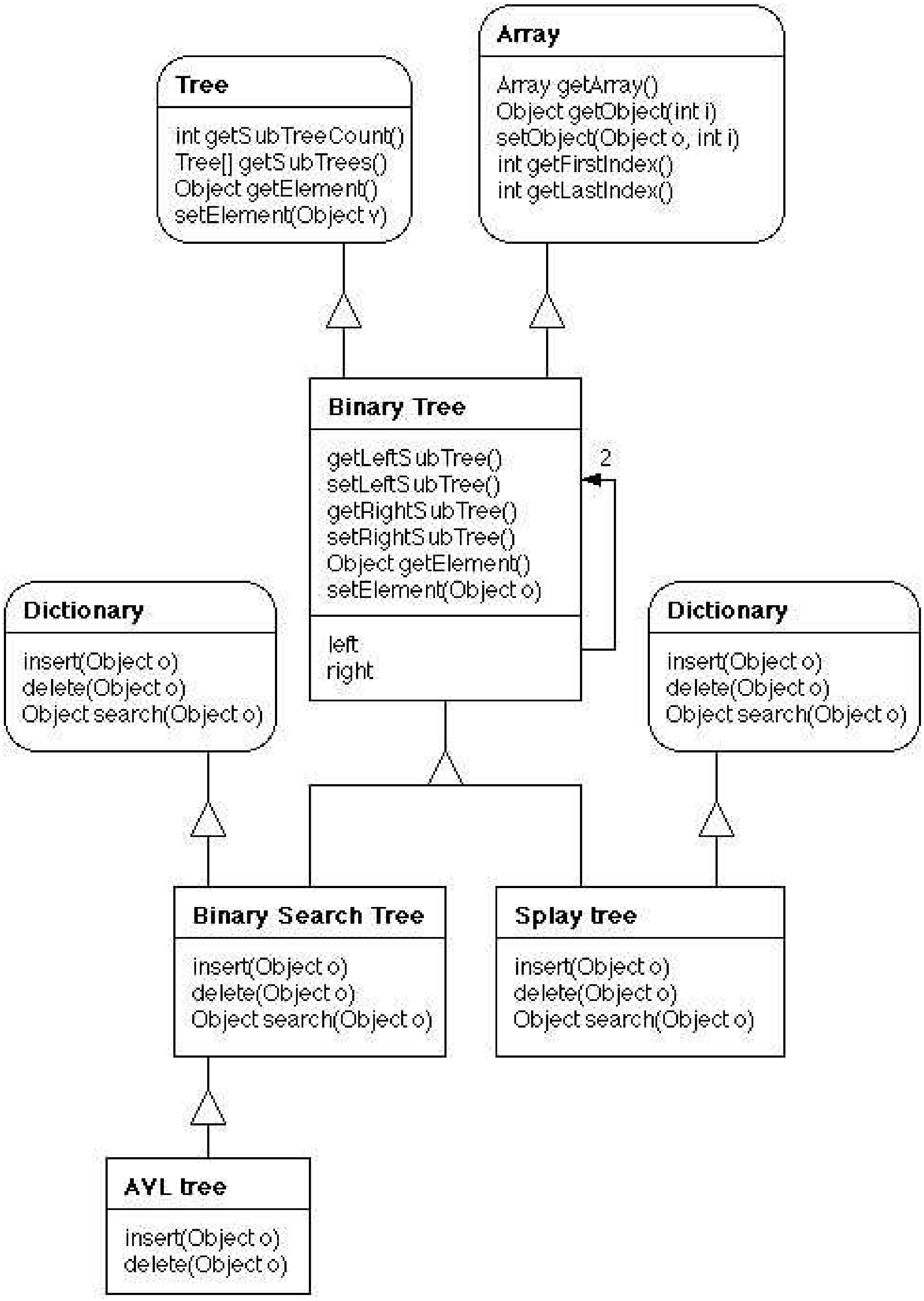


Figure 7.2: Extended hierarchy of conceptual data types.

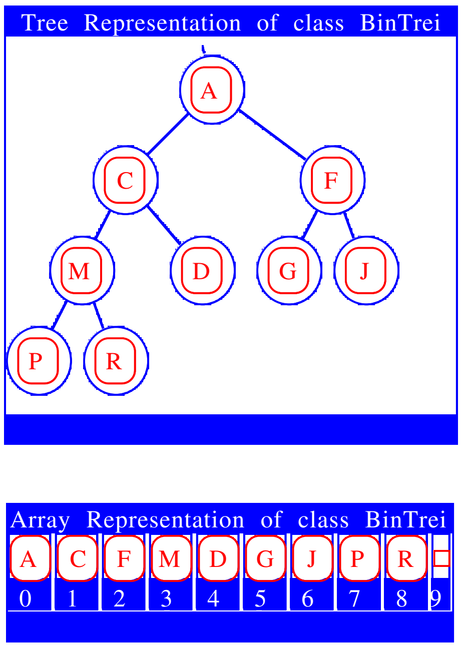

Tree Representation of class BinTrei
A
C
F
M
D
G
J
P
R
Array Representation of class BinTrei
A
C
F
M
D
G
J
P
R
0
1
2
3
4
5
6
7
8
9

Figure 7.3: Explicit and implicit binary heap representations.

we say that VisualTree is one possible representation for a graph. In other words, any structure implemented as a graph may implement the interface Tree and thus benefit from the VisualTree representation.

However, good design and implementation of these kinds of visual components is essential. The superclass VisualType offers several techniques to handle a situation like that above, where we avoid entering into an infinite loop. However, functionality like this, among other features of the class VisualType, is not discussed here any further.

7.4 Algorithm Animation and Simulation

So far, we have discussed the visualizations of data structures. From now on, it is time to put those static representations in action. By representing those visualizations as a continuous stream of frames, it is possible to create an animation. However, since the transition between two contiguous frames is the result of change in the underlying structure, we need a set of operations in order to modify the structure. Obviously, these operations can be defined as an algorithm. Thus, we are dealing with *algorithm animation*.

With the animated representation of an algorithm, the viewer can grasp the logic how the algorithm works. It is also possible to observe strengths and weaknesses of the algorithms by carefully studying their performance. The user also takes control of the *animator* that is used to store the animation sequence, which can therefore be played back and forth as many times as the user wants.

From the technical point of view, the animation process requires that the animated structure works together with all the other structures around in order to function properly. Thus, a special animator object stores the animation sequence. The animator object must be informed every time an assignment takes place. Because it is implemented as a *singleton pattern*, there is only one instance of it in which all actions are stored by invoking the method *store (object, index, old_value, new_value)* as illustrated in Figure 6.1 (on page 67).

The parameter *object* refers to the invoking object, the *index* is the (virtual) array position that stores the *old value* replaced by the *new value* during the assignment. Thus, it is now possible to reset the old value if the user takes over the animator controls and plays the assignment backwards. This is done simply by invoking *obj→set(index, old_value)*. This requires that the invoking object conforms to the interface *Memory Structure* in which the method *set(int i, Object o)* is declared.

The most convenient way to enable the animation for the structure is to reuse the *virtual array* that was introduced in the previous chapters. Thus, declaring all instance variables as virtual arrays encapsulates the animation process from the application. There are, however, a few exceptions. Because we cannot possibly store primitive types (such as int, float, double, ...) as objects, we must have a special set of memory structures for primitives. This is done by introducing a set of *Primitive Memory Structures* called *Int*, *Float*, *Double*, *etc*.

The animation functionality, however, is not effective without a representation. In particular, we often want to represent our data structures graphically in terms of visualization as described in the previous chapter. Using storage structures is the standard procedure to represent data structures in this environment, as described in Section 6.2. However, redesign of the actual data structure may possibly be needed. This is done in terms of fundamental data types, as described in Section 3.3.4.

However, the result should be a highly visual dynamic algorithm animation. In Table 6.1 (on page 78), we saw a set of snapshots of such an animation. The *buildheap* routine is invoked to an array of eleven elements. The resulting structure is represented after each swap operation in both array and tree representations. The swap operations are performed by the user in terms of algorithm simulation.

7.5 Summary

Recently, the research community has paid much attention to the concepts of program and algorithm animation, which refer to the visual, interactive presentations of an algorithm on a computer display for study and experimentation. Basic utilities, model instrumentation, and representational facilities are relatively easy to implement, for example, in the Java programming language. However, the basic building blocks necessary for representing data structures and the handling of dynamic relationships are not adequately supported by any of the more widely used SV systems. Thus, we have provided a modeling framework for this specific application area. The framework is based on several implementations of extensible general-purpose storage structures together with the ready-made visualizations for these.

We have introduced the notion of fundamental data type (FDT) in order to outline the field of data structures, and provided a set of reusable components to implement these FDTs. The design of a new conceptual data type (CDT) is defined in terms of fundamental data types. After that, the implementation is straightforward by reusing the existing implementations of the FDTs. The FDT instances include the possibilities to visualize, animate, and simulate the new CDT. These concepts provide an extensible set of building blocks and a common application framework for software visualization.

The framework is also capable of accepting new visualizations. Thus, development of new visual environments should be more rapid than creating them from scratch. The animation functionality is hidden in the basic building blocks called memory structures. By using only these basic building blocks as instance variables, the programmer does not need to know anything about the animation process. The programmer should, however, create carefully designed general purpose elements to be reused in construction of the actual abstract elements. The framework consisting of these general purpose elements provides a platform in which visualizations could be characterized in terms of VisualContainers, VisualComponents, VisualReferences and VisualData. In addition, the framework supports several kinds of tools for maintaining these kinds of visual object structures.

The programmers must reuse the memory structures in order to obtain all the functionality needed to animate objects properly. This might become an obstacle to producing animation. This is especially true when we are implementing a data structure without any prior knowledge of the framework. On the other hand, the use of memory structure is a somewhat more complex task to learn than subclassing an existing storage structure. Therefore, the technique should be very easily adopted by content experts. We believe that the content expert will also be needed in the future, for example, while designing new exercises for the TRAKLA2 environment, but this person's duties are very different compared with those of the current version of

TRAKLA. There is enough flexibility to construct new data types based on reuse of existing ones. Thus, visualization, animation, and even simulation do not require extensive programming by the content expert. This is because they can reuse the existing code that the developers provide for them. The reusable code hides all the complexity that drives the visualization processes. However, in any case, the programmer does not need to know the details of the animation process deep inside the framework (which is also true in the case of a content expert). Therefore, the whole functionality of data structure visualization, and algorithm animation and simulation is hidden in the reusable components.

Kreutzer [74] has stated that these kind of model generators can be used for rapid prototyping of stereotyped applications, but they lack the flexibility to cope with more complicated problems. However, while this is at least partially true, it should be noted that the framework is intended, in the first place, to be used with textbook examples, where the topic indeed is the illustration of very clear basic structures, thus stereotypes! In the next section, we will demonstrate how to apply the framework for such stereotyped applications.

Chapter 8

Educational Point of View

8.1 Overview

As stated in the previous chapters, simulation could be used for exploring and understanding symbolic models of complex systems. On the other hand, the purpose of any kind of education is to gain deeper understanding of some predefined topic. In this section, we will demonstrate and outline how the framework developed could be used for educational purposes in this sense. However, the question of how well the system communicates information to the student is left to the next chapter.

The educational point of view includes two separate viewpoints. In Section 8.2 we illustrate how the teacher or instructor may apply the prototype application as a demonstration tool to represent algorithms and data structures. On the other hand, in Section 8.3 we describe another important way of looking at the teaching process in which the student plays the key role. As a result, the framework is applied to provide the student with a task for practice. Moreover, an example shows how to provide feedback to the student on his performance in terms of automatic assessment.

8.2 Demonstration Tool

The idea of visual representations in understanding algorithms and data structures is by no means a new one. As far back as 1947, Goldstein and von Neumann [46] demonstrated the usefulness of flowcharts. After that, several systems have been developed to generate flowcharts automatically [63, 103]. This kind of "static" technique (the opposite to dynamic) has a role to play even today. On the other hand, some of the early approaches to "dynamic" techniques were those of Knowlton [61, 62] and

Baecker's [5] films. These were also the first to address the visualization of running programs.

As the reader has noticed, this thesis includes many snapshots of data structures illustrating several examples of the overall framework. This kind of "static" snapshot could be used in written text to illustrate data structures or to describe a specific state of an algorithm. We have used the tool to illustrate data structures and algorithms in Figures 4.3 (Dominator Tree, p. 52), 5.4 (DFST-Tree, p. 59), 5.6 (Adjacency List, p. 63), 5.7 (2-3-4 Tree, p. 64), 5.5 (AVL Tree Rotation, p. 61), and 7.3 (Binary Heap, p. 91).

However, the major problem in teaching data structures and algorithms is the difficulty of capturing their dynamic nature in static materials such as books and lecture notes. A proper tool for classroom demonstration would provide an ideal way to teach this kind of material. Such tools can support custom input data sets [18], provide multiple views possibly with different levels of abstraction [49, 112], include execution history [55], and support flexible execution control [15, 101, 106]. One possible use for Matrix (or TRAKLA(2)) is that of a demonstration tool in a computer science class to help illustrate and represent algorithms. It has been used in teaching to aid students' understanding of how algorithms work or to help illustrate how the data structure is constructed.

One obvious way is to use a sequence of visual snapshots, as described above (see, for example, Table 6.1 Buildheap, p. 78). These pictures could be shown using an overhead projector as a sequence of slides. On the other hand, Matrix can also be used for showing the whole animation live, still having the ability to define new examples by running an actual algorithms with several input data, or create an on-the-fly algorithm animation by manually simulating algorithms; play the animation sequence back and forth; have several windows (views) representing the same data structure in different representations, and so on.

Many papers have been published on how to *teach* data structures and algorithms. One explanation for this is the rapid evolution of the computer science discipline that has had a profound effect on the field, affecting both content and pedagogy [6, 30, 43, 78]. However, it is surprising that there has been so little discussion about how to *learn* data structures and algorithms, *e.g.* Faltin's "Exploratory Algorithm Learning" [41] or the "Active Learning Applets for Hypertextbooks" introduced by Ross and Grinder [99]. Thus, we would like to promote the discussion above not only to cover the need for new teaching tools but also for innovative learning resources. Surprisingly the requirements of such learning environments are very similar to those discussed above [89]. Hence, we will show how to apply the capabilities of the Matrix framework also for this greater task.

8.3 Electronic exercise book

From the student's point of view, the learning material within a learning environment can be divided into two categories. First, passive *instructive material* includes text, pictures and ready-made algorithm animations. Second, *constructive material* allows the learner to modify interactively ready-made examples, and also design and explore examples of his own. We consider this latter form of material to be more valuable because it increases learner engagement [70, 89]. There is also empirical evidence suggesting that learners do better through active rather than passive activities [54, 78].

However, interactive constructive material seems to be very time consuming to produce and is one of the most important factors that discourages the use of algorithm visualization in computer science education [89]. In the following, we will introduce two settings that can be considered effortless creations of such constructive study material. Both are applied in practice in our Data Structures and Algorithms course.

8.3.1 Laboratory Experiments

One way to foster the learner engagement is to set up feasible laboratory infrastructure for the discipline [14]. The course material for laboratory experiments in computer science, however, could be quite theoretical and conceptual, focusing on analysis techniques such as theoretical algorithm analysis (Big-Oh notation), experimental tests, and simulation. Thus, the new framework could be applied, for example, for a case study on a laboratory course. The framework provides an instrument to illustrate the observed results. It also provides tools for simulating algorithms. Thus, it makes it easier to create the set up for experiments. The idea is that when studying new algorithms with the aid of the framework, the user can observe strengths and weaknesses of the algorithms.

Moreover, the laboratory setting allows the instructor to pay attention to those students who need guidance while the others can experiment by themselves and explore the behavior of the algorithm freely. The assignments can be compared to ordinary exercises where a specific question demands a specific solution. Alternatively, the exercise can be more open in its nature and several different solutions can be accepted and discussed further. We will introduce a more detailed study of this question in the next chapter as we look at the question more closely in terms of simulation exercise taxonomy.

8.3.2 Automatic Assessment

One possible use for the whole system is to produce individually tailored *algorithm simulation exercises* to be solved by the students in Algorithms and Data Structures courses. This idea is adopted from the TRAKLA system and implemented as one of the main applications for the framework.

The framework is capable of visualizing the exercises by showing the assignment in a text window and by graphically illustrating the fundamental data types needed to simulate the algorithm as illustrated in Figure 6.5 on page 80. It is also capable of determining the model solution for each exercise. This model solution is then automatically compared to the submitted answer. The assessing procedure then gives immediate feedback to the student on his or her performance. Moreover, the model solution can be visualized and displayed for the learner as an algorithm animation sequence. The different types of exercises are discussed and described more thoroughly in Section 9.1.1.

Moreover, the overall TRAKLA system has been a learning experiment since the very beginning, because it was implemented as a student project at the Helsinki University of Technology. Moreover, the whole TRAKLA concept is based on the idea of having a Web-based *learning environment* in which students have all necessary facilities available to take over the control of their own learning process.

8.4 Summary

We summarize the different perspectives, applications, and use cases introduced in this chapter here. The system can be applied as follows.

1. *Instructive* learning material for a student provides text, pictures and ready-made algorithm animations to be studied at learners own pace and time.

2. *Demonstration tool* for a teacher allows illustrating various data structures and algorithms as static pictures or dynamic algorithm animations to such an extent that the visualizations can be constructed on-the-fly.

3. *Closed labs* allow learners to solve exercises and to experiment with the environment in class room under the guidance of their instructor.

4. *Open labs* allow students to solve exercises via the internet & WWW using the learning environment at their on pace and time.

5. Any combination of above, for example open labs followed on from closed labs where different aspects of the questions are concerned more closely or where

the learner receives guidance only for those questions that have proven to be difficult.

As we have seen, the framework has a number of options for applications. In the following final chapters, we will focus on a couple of them more thoroughly and come up with the evidence gathered during the many experiments that supports our view that the tools have a valuable positive influence on teaching and learning.

We are going to evaluate the systems especially in their effect on learning. Hence, the learner's point of view is concerned to be the most important perspective and the applications from teacher's point of view is not discussed any further.

Chapter 9

Evaluation

In this chapter, we are going to represent and analyze the empirical data gathered in many experiments conducted with the framework and learning environments under study. We start by comparing the two environments, TRAKLA and TRAKLA2, to each other based on the taxonomical approach in the context of simulation exercises.

After that, we represent the main results of the large scale intervention study carried out with TRAKLA environment [69]. The study shows that there is no significant difference between groups of learners doing exercises on the Web compared to those exercising in classroom. However, the evidence also supports the view that more challenging exercises could devise the new environment to achieve even better learning outcomes. Thus, this fact justifies our view of developing the environment further by building up more versatile exercises.

Another important point of view is to gather evidence on successful changes in course setup over a longer period [80]. We argue that great care must be taken of not just developing the exercises but also the process the learner goes through while solving the exercises. For example, algorithm visualization seems to be a promising motivation factor but there is very little evidence that it actually fosters learning intrinsically. Therefore, we suggest that we should identify other motivating factors that tends the learner to spend the time needed to learn a topic within the environment. These facts are discussed at the end of this chapter.

9.1 Automatic Assessment and Feedback

Computers are now being used to examine the behavior of complex systems in a variety of disciplines, including mathematics, computer science, and engineering. There are several methods to apply. One such method is simulation and another is animation of the behavior of complex systems.

Today, the primary use of algorithm animation has been for teaching and instruction. However, from the pedagogical point of view, *feedback on students' performance* is needed in order to see the system as a learning environment. Fortunately, the formal nature of algorithms and data structures allows us to compare the student's solution to the correct model solution easily. This gives an opportunity to produce a system which not only portrays a variety of algorithms and data structures, but also distributes tracing exercises to the student and then evaluates the student's answer to the exercises. This is called the *automatic assessment and feedback* process of simulation exercises.

In this thesis, we have described how to apply algorithm animation and simulation together to construct a *learning environment* for data structures and algorithms. By a learning environment we mean a system that is capable of meaningful interaction with the user with respect to the topic of the class to which this environment is attached. We refer to the users of such an environment as students.

9.1.1 Taxonomy of Algorithm Simulation Exercises

As the dynamics of a simulation is targeted to algorithms we can derive the taxonomy of algorithm simulation exercises by examining the function $P : I \rightarrow O$ that the algorithm A is supposed to compute. The exercise can employ any of the three components shown here, *i.e.*, the algorithm (instructions to compute P), the input I or the output O, or any combination of them, while the other components are fixed (predefined in the assignment). Thus, an *algorithm simulation exercise* is a tuple $E_A = (P, I, O)$, where P is the problem to be solved by the applied algorithm A, I is the set of allowed inputs (problem instances), and the solution space $O = P(I)$.

Moreover, an *algorithm simulation exercise type* is a tuple $E_T = (X_1X_2X_3)$, where X_i is substituted by F if the corresponding E-component, *i.e.*, algorithm, input or output is fixed, respectively, and Q if it is in question. Most of the TRAKLA exercises ask the student to simulate a particular algorithm A with individually tailored input sequence $f_1, f_2, \dots, f_k \in I$, and show how the corresponding structures change by determining the output sequence $q_1, q_2, \dots, q_k \in O$. Thus, the exercise is of type $E_T = (FFQ)$ and we denote the input and output sequences as F_k and Q_k, respectively. The question is, in general, what is the output for the given algorithm with the given input. For example, F_k can be an ordered set of keys to be inserted into an initially empty binary search tree or it can be the parameter(s) an algorithm receives in each recursive call. Here, for each f_j the corresponding $q_j = A(f_j)$, *i.e.*, the binary search tree after each insertion or the computed result after each recursive call to A.

On the other hand, two different subtypes for E-components can be identified: implicit and explicit. Explicit question (denoted by capital case Q) requires the learner to produce an answer for the question, for example, the output in the previous example. Implicit questions (denoted by lower case q) only require that the learner is familiar with the topic in question but no explicit answer is required and the learner may choose from among a set of alternatives. For example, the learner may be asked to apply any algorithm that produces topological sort and not any particular one. Usually, this happens when there is more than one Q-component in an exercise. On the other hand, also fixed E-components can be implicit. For example, it is obvious what is the output structure while sorting an array of keys. Thus, we denote the implicit output as lower case f.

Eight different basic types of exercises can be employed. In the following, we summarize them briefly and give an example of each.

1. $E_1 = (FF_if)$ - *Determining characteristics*: Which items in the array F_i are compared with the given search key $k \notin F_i$ in binary search?

2. $E_2 = (F_aF_iQ)$ - *Tracing exercise*: i) Insert the set of input keys F_i into an initially empty binary search tree. ii) Insert the set of keys F_i into an initially empty hash table using linear probing with the given hash function F_a.

3. $E_3 = (FQF_o)$ - *Reverse engineering exercise*: Determine a valid insertion order for the keys resulting the AVL tree F_o in question.

4. $E_4 = (FQq)$ - *Exploration*: Determine such an input string for Boyer-Moore-Horspool algorithm that satisfies the statement coverage (every statement is executed at least once with the test set). Describe the output of such an execution.

5. $E_5 = (QF_iF_o)$ - *Determining algorithm*: The following binary tree F_i was traversed in different orders. The resulting traversing order of nodes are F_o. Name the algorithms.

6. $E_6 = (qFQ)$ - *Open tracing exercise*: i) Trace topological sort on a given graph. ii) Compare several recursive in place sorting algorithms with each other. Show the state of the input array F after each recursive call.

7. $E_7 = (qQf)$ - *Open reverse engineering exercise*: Compare several sorting algorithms to each other. Determine an example input for each of them that leads to the worst case behavior.

8. $E_8 = (QQQ)$ - *Completely open question.* Let us consider the following broken binary search tree. All the duplicates are inserted at the left branch of the tree, but the deletion of a node that has two children replaces the key with the next largest item (from right branch). Give a sequence of insert and delete operations with keys of your choice that results in a tree that is no longer a valid binary search tree (Hint: the search routine can only find duplicates from the left branch).

In general, algorithm simulation exercises require systems that have a deeper knowledge of the underlying data structures to be visualized. A simple drawing tool is not enough because it lacks the power of invoking arbitrary complex procedures by single manipulation operations. This is essential if the system is targeted to cover more complex operations than those provided by the developer of the system. It is, for example, easy to construct a linked list representation in terms of simple pointer manipulation, but more complex operation, such as AVL-tree rotations, typically require interference of the developer of the drawing tool. However, we have separated the roles of programmer and developer in order to better answer the demands of different kinds of applications. Thus, it is the programmer who should focus on the task of programming the AVL-tree rotations without the need to worry about the visualization, animation and simulation details.

The taxonomy of algorithm simulation exercises was invented by the author, but an interesting analogy exists independently with completely different area of teaching. In their paper [113], Sutinen and Tarhio explains similar example of problem classes that are applied for characterizing problem management in thinking tools. They also have three components (Start (input), Technique (algorithm), and Goal (output)) that they call dimensions. Again, these expand to eight classes, if restricted to binary values "open" and "closed" (in question and fixed above). This construction spans the creative problem management space.

Automatically Produced and Assessed Algorithm Simulation Exercises

Two main characteristics should be taken into account while designing new exercises. First, we should make clear how these exercises are delivered to the students. We identify two main classes here, namely *closed labs* (CL) and automatically assessed exercises (AAE). The main difference between these two is the available support from instructors. In closed lab sessions, we assume that an instructor is present and takes actions to guide the learner by asking specifying questions, giving additional feedback, and so on. The exercises can be solved by "exploring" the state space and the correctness of such an exploration can be assessed by the instructor. Thus, the exercises are more open in their nature. On the other hand, automatically assessed exercises should be self-explanatory so that the learner can cope with the assignments by himself. In addition, the feedback should be explicitly targeted to the assignment in question and aid the learner to find the correct solution. Roughly speaking, open questions are more suitable for closed labs while tracing exercises suit well to automatic assessment. There are, however, counter examples that contradict this discrete view.

The other main characteristics is that, especially with automatically assessed exercises, the fixed E-components can be parametrized (denoted by subindex) so that each learner has his own individualized exercises. For example, in the tracing exercise $E_2 = (FF_iQ)$, where keys are inserted into an initially empty binary search tree, the keys F_i to be inserted can be randomly drawn from the set of all possible keys. Thus, each learner has an individually tailored set of keys to be inserted. In the case of input and output components, this is trivially achieved by randomly picking a suitable number of keys into the corresponding structure. Of course, some constraints must be taken into account in order to prevent the exercise turning out to be trivial (for example, the AVL tree insertions should include both single and double rotations). However, parametrized algorithm exercises $E_{1\ldots4} = (F_aXX)$ are slightly trickier. Usually, the implementation of such an parametrization depends on the algorithm. For example, in linear probing, the hash function $h(k) = (k + p) \bmod q$ can be parametrized by individually tailoring p and q.

Examples of Automatically Assessed Algorithm Simulation Exercises

In the following, we will summarize the types of exercises the current TRAKLA system supports, and we will extend the selection to include several new types incorporated into the new TRAKLA2 system. Of course, the taxonomy presented here is not exhaustive in the sense that there exist exercises that are not algorithm simulation exercises at all. However, the taxonomy gives us a systematic way to cover one particularly interesting area of exercises throughout.

Other systems promote algorithm simulation exercises as well, but only within limited scope. PILOT [16] is targeted to tracing exercises, but covers only graph algorithms. There is also an option to allow parametrized input graphs. Thus, the type of the exercises is $E_2 = (FF_iQ)$. On the other hand, "stop-and-think" questions requiring an immediate response from the learner introduced in JHAVÉ [88] can be interpreted to be $E_1 = (FF_if)$ questions where the learner determines characteristics of the given algorithm. Moreover, the same system illustrates exercises where the learner is asked to manually trace an algorithm on a small data set. However, in this case there is no automatic assessment involved. Thus, in the following Table 9.1, our aim is to address the many possibilities of automatically assessed algorithm simulation exercises in the broad context of data structures and algorithms.

Table 9.1: *Automatically assessed exercises supported by TRAKLA and TRAKLA2.*

Exercise type	TRAKLA2	TRAKLA
$E_1 = (FFF)$	FF_if	FF_if
$E_2 = (FFQ)$	FF_iQ	F_aF_iQ, FF_iQ, fF_iQ
$E_3 = (FQF)$		
$E_4 = (FQQ)$	FQq, Q implies q	
$E_5 = (QFF)$		
$E_6 = (QFQ)$		qF_iQ, q implies Q
$E_7 = (QQF)$	qQ_if	
$E_8 = (QQQ)$	$QQ'q, Q'$ implies q	

All the exercise types marked for TRAKLA have been in production use. Also exercises FF_if, FF_iQ, and FQq in TRAKLA2 have been actually tested with students. The rest of the exercises are only designed to be incorporated into the system in the future. In addition, exercises not marked for TRAKLA2 could be incorporated into the system, but we have decided to implement, for example, the tree traversing exercises ($E_6 = (QF_iF_o)$) mentioned on page 102 and the Huffman code exercise ($E_6 = (qF_iQ)$) introduced with TRAKLA a different way, *i.e.*, as a tracing exercise ($E_2 = (FF_iQ)$), and as an open reverse engineering exercise ($E_7 = (qQ_if)$), respectively. In the following, we map each exercise type described above to an actual exercise implemented or designed.

1. FF_if; binary and interpolation search; the algorithm is explicitly determined as well as the parametrized input. The result is obvious (the key is not found from the structure). The user must determine which items in the array F_i are compared with the key to be searched during the search.

2. F_aF_iQ, FF_iQ, fF_iQ; linear probing, deque, radix sort; the algorithm can be parametrized as it is in case of linear probing, it can also be expressed implicitly as it is done on radix sort (the learner must determine which radix sort is simulated by examining the intermediate states of the input structure).

3. FQq (TRAKLA2 only); Determine such an input Q for Boyer-Moore-Horspool algorithm that satisfies the statement coverage (every statement is executed at least once with the test set). Describe the output of such an execution. The selected input Q implies the output q.

4. qF_iQ, q implies Q; topological sort, Huffman code; in both of these exercises several correct solutions are possible for a given input. The algorithm applied by the learner implies the resulting output.

5. qQ_if; Huffman code; the learner is asked to simulate Huffman's algorithm q to form the Huffman code f in order to decode the original text string Q_i. The actual visible input for the algorithm is the frequencies of the characters in Q_i.

6. $QQ'q, Q'$ implies q; broken binary search tree; many procedures can lead into a correct solution. Several correct answers are possible due to the nature of the exercise. The input keys chosen by the learner imply the resulting output.

We have currently evaluated only the effect of the exercise types supported by TRAKLA, as we have discussed in Chapter 8. It should be noted, however, that as we can see, those exercises are only a subset of exercises supported by TRAKLA2. Thus, we conclude that the new TRAKLA2 environment is at least as effective as the old one.

9.2 Intervention Study

This section is based on the article [69] describing a large scale intervention study carried out to prove that the learning environments incorporating automatically assessed exercises actually promote learning. Several attempts have been made to produce both effective and high quality environments that are capable of automatic assessment of exercises [13, 40, 56, 57, 66, 102]. The need for automation is due to the trend towards teaching large numbers of students at low cost. Moreover, virtual courses benefit from automatic feedback since it is difficult and expensive (or practically impossible) to provide constant (or round the clock) service with human resources. However, the current trend in automatic assessment research seem to emphasize only the assessment of programming exercises. Of course, many programming course instructors are glad to see this development. On the other hand, our research aims at a more abstract level in this respect. We are not that interested in focusing on how to program an algorithm but to introduce the logic behind the code lines. Moreover, we believe that introducing an algorithm, for example, in a lecture as an animation, is not enough.

In order to understand the behavior of a complex procedure, one should try it out by himself, *i.e.* practise on the topic. This process, however, requires constant feedback that aids the learner to resolve misinterpretations and shows the way to the correct solution. We believe that in this way, the learner can engage in the exploration process promoted by the exercise.

Recent automatic assessment systems are not capable of assessing students' exercises in as much detail and giving as sophisticated feedback as teachers do. On the other hand, some of these systems have advantages that could compensate for these limitations. For instance, the TRAKLA environment under study has features that traditional instruction cannot provide [66]. The students solve *individually tailored* exercises by means of algorithm simulation, and the system is capable of *immediate* automatic assessment of these exercises *around the clock*. The assessment produces feedback that enables *revision of answers*. Moreover, after the deadline has passed, the system automatically produces *individual model solutions* for the exercises.

The study described here originated from the need to research the quality, advantages, and limitations of these approaches. The question was whether the advantages of fully automatic assessment can compensate for its limitations? In this experimental intervention study with three randomized groups A, B, and C ($|A| = 372, |B| = 77, |C| = 101$), student performance was monitored over the second year course Data Structures and Algorithms course during a twelve week period. The examination results of students in the virtual learning environment were compared with those in the traditional classroom sessions.

9.2.1 Setup

For the research, the students were divided into three separate groups, A, B, and C, each of which had different homework exercises. The experimental group A used the TRAKLA system. The control groups B and C had traditional classroom sessions in which the tutors gave feedback on exercises for the students. The control group B did the same simulation exercises as the experimental group A. However, the control group C did exercises which were beyond the scope of automatic assessment (*i.e.* designing new algorithms, writing essays, *etc.*). This latter set of exercises was designed to be more challenging for the students than the TRAKLA simulation exercises. We not only covered the algorithms described in the text book, but also designed new algorithms solving similar problems to those in the simulation exercises. For example, if the text book described the standard Tree Traversing algorithms for such binary trees in which each node has links to the left and the right subtrees, the assignment was to design the same set of algorithms for the common tree implementation, in which each node has links to its first child and its next sibling.

Table 9.2: *Summary of Groups. N denotes the total number of students in a group. Each group was split into s subgroups of average size ave. Each group had a different teaching method; this was the variable of the survey.*

Gr.	*N*	*s*	ave	Teaching method
A	372	1	372	virtual simulation exercises
B	77	3	25	classroom simulation exercises
C	101	2	50	classroom design assignments
$\sum$	550	6	-	-

The groups were formed randomly. The sizes of the control groups were limited by the classroom capacities. Group *B* was divided into 3 subgroups with approximately 25 students in each, and group *C* into 2 subgroups with approximately 50 students. Each subgroup had two tutors who gave feedback to the students during the classroom sessions. Group *A* was not split since they worked with the web-based exercises and space was therefore not an issue. All students had the option of getting guidance from course assistants during office hours, or by asking questions in the course newsgroup. The summary of the groups is represented in Table 9.2.

All students in this study were native Finnish speaking freshmen on the course, *i.e.*, adult students, students who had failed the course previously, as well as foreign students were not included in the study. Such students were, however, allowed to take the course.

Examination

The final examination was designed to cover all subject matter. There were five kinds of assignments for each main topic (sorting, searching, graphs) of the course. First, certain key concepts had to be defined. Second, an important data structure or algorithm within the topic had to be defined or explained. The students had a degree of freedom in deciding which structure/algorithm they explained, *e.g.*, they could define any balanced search tree instead of a certain balanced search tree. In the third assignment, they had to apply their algorithm to a simulation case, where the initial data structure was given. Fourth, they had to explain how algorithms within the field can be compared with each other while choosing an algorithm for an application. Finally, they had to construct some simple new algorithms and analyze their performance briefly.

Table 9.3: *Summary of Results. E denotes the average examination points (max = 70). D_1 and D_2 denote the percentage of students who dropped the course in the beginning without any submissions, and those who dropped the course during the exercises, respectively. D_3 denotes the percentage of students who passed the exercises but either did not attend any examination or failed the examinations. P_{exam} denotes the percentage of students who passed both the exercises and the examination.*

Gr.	*E*	D_1	D_2	D_3	P_{exam}
A	33.04	4.6%	8.9%	4.8%	82%
B	32.74	0.0%	12%	5.2%	82%
C	39.56	8.8%	26%	2.0%	64%

9.2.2 Results

The correlation between the exercise points and the examination points is high in each group (Pearson's $r_A = 0.402$, $r_B = 0.420$, and $r_C = 0.577$). Thus, there is significant positive linear relationship between the two variables. Closer examination reveals that there are also significant differences among the groups.

The results of the study are summarized in Table 9.3. The three groups were compared with each other with respect to two important factors. The performance of a student was determined according to the *examination points*. Not all of the students who initially enrolled in the course were able to finish the course successfully in the final examination. Consequently, the *degree of drop out* during the course was additionally measured. The drop out in this case, however, was determined by counting those students who were not able to finish their exercises before the end of the course. On the other hand, not all of the students that were able to finish their exercises showed up in the final examination.

Note. The students had in total 5 possibilities to take the course examination after the course. The examination points are reported only from the first examination, which most of the students attended. The points from the latter examinations are not reported since those are not comparable with each other. On the other hand, the drop out rates are reported after all the examinations because the comparability of the examination results is not important from this point of view.

Performance in Final Examination

Those students of group C who attended the final examination did well compared with the other groups. The t-test showed that their performance was significantly better than that of groups A and B (p-values 0.000 and 0.005 respectively). On the other hand, there is no significant difference between groups A and B ($p = 0.871$). However, the comparison becomes complicated if the huge drop out in group C is taken into account. Examination of the data shows that there is no single attribute that could account fully for the phenomenon.

Drop out

There are several reasons for the higher percentage of students who failed the exercises in group C. First, all the groups knew the presence of each other; thus, it was natural that they compared their exercises. The design assignments were more difficult than the simulation exercises, and some students in group C felt that we pushed them too hard, which caused lower motivation. Second, the design assignments may have been too difficult, causing the students to feel anxious and worried. Third, the size of the group was possibly too large for only two tutors to provide enough feedback for such a difficult set of problems to as many as 50 students within the two hours time limit. Moreover, those who took part in the examination appeared to perform better in all of their studies than those who did not.

Within groups A and B the number ($D_1 + D_2$) of students who dropped the course during the exercise phase is about the same. On the other hand, the drop out in the very beginning of the course is different. Students attending the classroom exercises in group B were motivated to start the course seriously although a number of them gave up later. Students working in the web-based environment dropped the course easier already in the beginning, whereas those who started working seriously were less likely to give up later. This phenomenon was much stronger within the group of students (not presented in the table) who were not taken into the research setting. 45% of these students, among whom were those who had failed the course earlier, foreign students and a number of working adult students, dropped the course either before or during the exercises phase.

9.2.3 Conclusion

First of all, there is no significant difference in the learning results of students doing their homework exercises in a virtual learning environment and students attending a traditional classroom. In addition, the commitment to the course (low drop out), is almost equal in both versions of the course. However, students working in the virtual environment seems to be more likely to drop the course earlier than students working in the classroom.

The examination results were much better when the exercises were more difficult. The problem is, however, how to maintain a reasonable drop out rate. The results imply that, if we intend to foster learning, we may choose either traditional or web based learning assignments, as long as they are suited to the subject at hand. We should, however, pay more attention to the quality of the exercises. Furthermore, the quality of the feedback and the time used for dealing with a topic should be appropriate for the challenges to the students. Thus, the more challenging the exercises are, the more support and time are required to cope with the task.

On the other hand, it seems to be that the challenges in our virtual learning environment are not difficult enough. At least the improvements we have made over the past few years, (*e.g.* the possibility to revise answers, graphical front end *etc.*) have made it possible for students to cope with the exercises much better than, lets say, five years ago. Thus, the plan to produce a more powerful system that is capable of delivering more challenging exercises is justified.

As we have demonstrated, several important factors can be identified that promote the learning, for example, the possibility to resubmit answers after receiving feedback and the challenge of exercises. However, our experience shows that these are not the only key factors. In the following section, we will show how the grading scale affects the learning curve. Thus, a successful combination of all of these finally produces good learning results.

9.3 TRAKLA 10 Years Study

We have been using the TRAKLA system since 1991 on the Data Structures and Algorithms course. Since then, the system has evolved and certain changes in the course requirements have taken place. Throughout this period, the algorithm simulation exercises have been a compulsory part of the course. In addition, most of the assignments have remained unchanged since 1993. The lecturer and teacher-in-charge changed only once during this time, in autumn 1996.

In the following, we present some experiences and observations on the TRAKLA system that we have made in a ten year period [80]. It is inevitable, for example, that students get better results when they are allowed to resubmit their work. However, this is not enough; it is also important to organize the course so that the skills and the challenges of the students coincide. Also several other attributes come up regularly that have their role to play in the learning process as a whole.

9.3.1 Results

In Figure 9.1, we can see the relative distribution of grades during the time period 1993-2001. There are four categories of students. First, those who achieved less than 50% of maximum points and failed the exercise part. Second, the students who received 50-90% of maximum points and passed. Third, those who received more than 90% of the maximum points and received the maximum grade, and finally those students who solved all the exercises correctly. The number of students varied between the minimum of 334 students in 1993 to the maximum of 718 students in 2000.

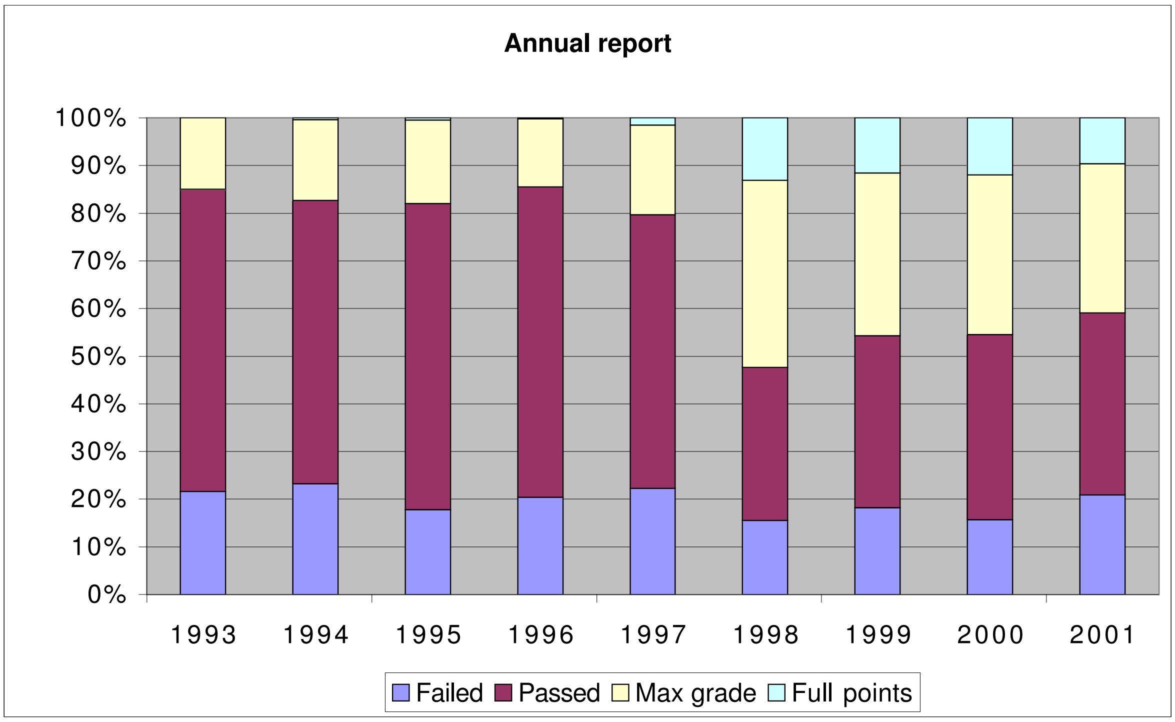


Figure 9.1: Relative distribution of grades into four categories.

It should be noted, however, that the presented results are not the final results. We required that all student receive a certain minimum number of points for each of the 5 to 6 exercise rounds. Each round typically included exercises related to a specific important course topic, such as sorting, search trees, or graphs. If a student failed to get the minimum from one or more rounds, he or she had the option to pass them by completing extra assignments later. They could also raise their grades. Normally, 50-100 students used this option. However, organization of the extra assignments has changed quite considerably during the period. Thus, we present the results without these additional points.

New Tools and Features

As we can see, there are significant changes in the distribution. Up to 1997, the overall distribution remained roughly consistent but thereafter the students achieved far higher grades, particularly in 1998. Subsequently, the results weakened slightly.

Between 1993 and 1996, both the TRAKLA system and the course requirements remained unchanged. In 1997, however, we introduced two major enhancements to the system. First, the resubmission of answers was allowed. Up to 1996, students had only one chance to submit their solution, and they received the feedback after the deadline for the submission round was passed. Second, we introduced a graphical web-based front-end, similar to that of TRAKLA2, for solving assignments. Previously, the only method for submission was to send an email message in a predefined format to the TRAKLA server. All data structures had to be presented as ASCII text. Thus, the method was somewhat clumsy, for example, in the case of trees. From now on, the front-end, based on direct manipulation, took care of the transformation of the required states into the format required by the TRAKLA server. Moreover, simple formatting errors were avoided in the solutions, and visual representations could be provided for data structures. As can be anticipated, the results for 1997 were somewhat better than for 1996. It is interesting to speculate, however, what happened in the next year? We have excluded several factors.

On average, students submitted an answer to an assignment 1.45 times in 1997, and 1.47 times in 1998. These figures are between the normal variation compared with other years. The proportion of students who actually used the graphical front-end was 23.7% and 42.6%, respectively. However, when we compared two separate groups of students, those who used only email for submissions, and those who used only the graphical front-end, we observed no difference in performance. Thus, the submission media does not appear to account completely for the change.

The relative distribution of overall grades in Figure 9.1, however, do not explain the whole phenomenon. The results of the *first submissions* in 1997 and 1998 compared to 1996 (when only one submission was allowed), were clearly worse. These figures are shown in Figure 9.2. After the resubmission was allowed in 1997, the students submitted their first answer more carelessly than before. Nonetheless, the overall performance was still slightly better in 1997 and considerably better in 1998.

Changes in Course Requirements

Because we cannot explain the difference between the results in 1997 and 1998 as described above, the changes in the course contents and organization together may have had the major effect on the results in 1998.

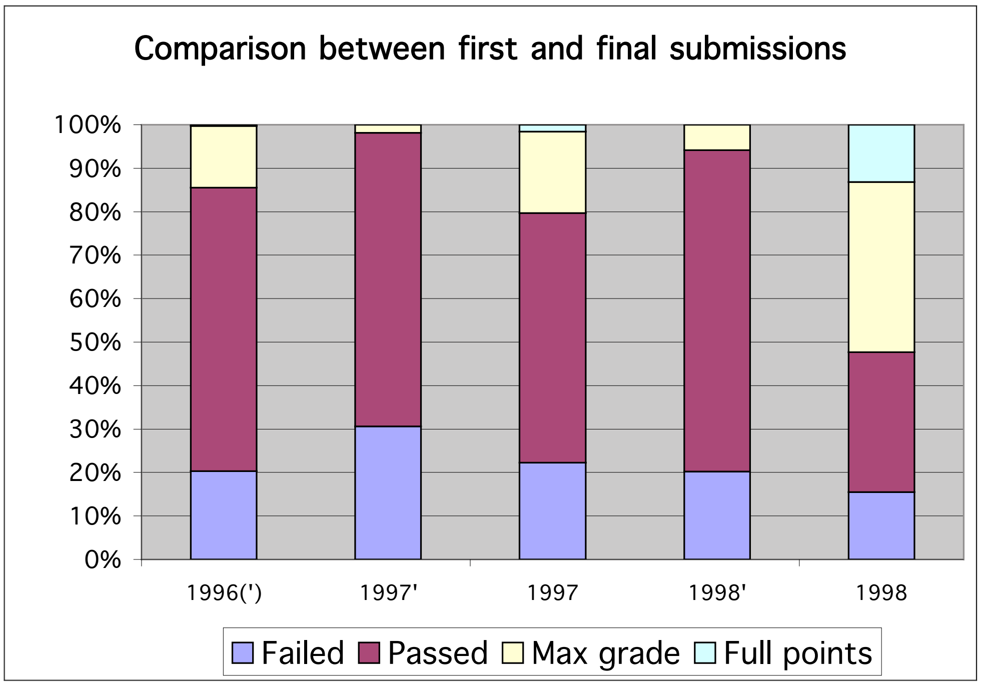


Figure 9.2: Relative distribution of grades between the first submission denoted by prime and the final submission in years 1997 and 1998. Year 1996 represents a comparative figure in case no resubmission was allowed.

First, we dismissed such simple exercises as manipulating stacks and queues, because they were too trivial in graphical form. Instead, we added a number of more challenging exercises, such as inserting items into an AVL tree, a red-black tree, a B^+-tree, using extendible hashing, together with generating minimum spanning trees and shortest path trees.

Second, we introduced another type of exercise, which was submitted on paper and checked manually. These considerably more challenging *analytical and constructive exercises* included algorithm analysis and small algorithm design problems.

Third, the grading policy of the simulation exercises changed. If a student received between 50% and 80% of the maximum points, the final grade on the course was the same as the examination grade (0=failed, 5=maximum) in 1997. Moreover, if the result was at least 80%, the final grade was the examination grade plus one unit, unless the student failed or achieved 5/5 of the exam. However, one year later, the home exercise grading was changed, as illustrated in Table 9.4. The simulation exercises and the analytical exercises became a part of the examination and both were worth 30% of the total grade. The remaining 40% was covered by the final examination.

Table 9.4: *The effect of home exercises on final grade vs. total grade (weight 30%) before 1997 and after 1998, respectively (% denotes the percentages of points out of the maximum).*

%	grade -'97	grade '98-
[00 - 50)	failed	failed
[50 - 60)	passed	1
[60 - 70)	passed	2
[70 - 80)	passed	3
[80 - 90)	+1 unit	4
[90 - 100]	+1 unit	5

9.3.2 Discussion

As an overall effect, the course requirements and the course work load were increased quite considerably in 1998. It is interesting that the results were far better under these circumstances. Why did this happen? We assume that two factors had a role to play here. First, the change in the grading policy, combined with the option to resubmit the exercises, motivated the students to aim at better grades because the exercises had more effect on the final grade. Second, the more advanced exercises raised their motivation, and had a positive effect on the learning curve as a whole, *i.e.*, previously, the course had been too easy to pass and did not motivate students enough to learn more.

Moreover, because of the increase in the work load for both the teachers and the students was considerably high, we replaced the analytical exercises with a half-semester team project in years 1999 - 2001. During this period, the results weakened slightly compared with 1998. Thus, it seems that the difficulty of the exercises motivates the students to perform better. This is, however, a two-fold matter, as we have demonstrated in the previous section. If the students are pushed too hard, the drop-out rate grows prolifically.

In 1998, however, the drop-out was relatively small. One explanation for this could be the fact that all the students solved the same set of exercises. Hence, they did not have a comparison group with "easier tasks". Thus, their motivation remained the same throughout the course.

Finally, an interesting matter is that approximately 10% of the students gained full points from the exercises, although they knew that they would not profit directly from the extra work. Thus, they were apparently motivated by the assignments.

9.4 Other Research

The educational effectiveness of algorithm visualization have been studied in several integrative reviews. We cover two of them briefly by summarizing those aspects of these studies that support our view of the effectiveness of the use of visualization in computer science education as well as some questions that raises the antithesis against systems that rely completely on the use of visualization.

Chen *et al.* [28] have done statistical meta-analysis of a sample of 35 experimental studies of information visualization. Their analysis determines that individual differences have a larger and more consistent effect on human performance than does information visualization technology itself. On the other hand, Hundhausen *et al.* [54] have introduced a systematic meta-study of 24 experimental studies. They performed two separate analysis: learning theory analysis and measurement technique analysis. They argue in their conclusion that the students use of algorithm visualization (AV) technology has a greater impact on effectiveness than what AV technology shows them.

As the number of studies in both of the meta-studies above imply, the number of original papers describing the studies is too numerous to be cited here. However, we introduce a couple of those that support our view of what kind of algorithm animation is effective in computer science education. Three issues are often repeated directly or indirectly in these papers.

The effect of time used. For example, Hansen's *et al.* experiments with HalVis [49] shows that students might spend more time with an AV system than with conventional material. Although the authors conclude that it was the HalVis system that caused the effect, we make the hypothesis that it is the time used with the system that counts. This fact can also be found from the evidence in the original paper. The students using the HalVis system used considerably more time with the topic than others. Other authors [58, 106] have made similar conclusions in their previous work.

Engagement to the course. Another issue closely related to the above is the student's engagement to the course and the visualization tools [89]. This is a two-folded issue. It is evident that the more the students are involved with the course, the better results they will achieve. This can be fostered by opening up new possibilities for students to get better grades as we have illustrated in the previous section. However, one particular problem with this issue is the question of how to archive reliable results that last for a long time. As the students become increasingly familiar with graphical symbol systems and are more aware of their capabilities, the effect of such a learning environment seems to diminish over a long period. This can be seen in the Figure 9.1. Seeing that the latest numbers of best students have not achieved the level of the year 1997, the trend is clear. Of course, competition with other courses also having AV technology available may have an impact on this. However, the impact of the “hype effect” of visualization systems seems to disappear in long run. Thus, we must be very careful before jumping to the conclusions of the results gathered during experiments. This is even more important while monitoring small student groups. Is the effect still there next year?

Looking is not always seeing. In his article on Multimedia Learning, Mayers [82] proposes that students learn better if visual explanation is added to a verbal one. Or inversely, Naps, Eagan, and Norton [88], for example, argue that to be effective, AV must be accompanied by comprehensive explanations. On the other hand, Petre [92] points out that text and graphics are not necessarily an equivalent exchange and we still do not fully understand what is good about graphics. This raises the question whether the visualization has anything to do with the learning results? This is noted by other authors as well [33, 78, 106]. To put it another way,

“With few exceptions, we found that studies in which students merely viewed visualizations did not demonstrate significant learning advantage over students who used conventional learning materials. Perhaps contrary to conventional wisdom, this observation suggests that the mere presence of AV technology – however well designed and informative the visual representations it presents may appear to be – does not guarantee that students will learn an algorithm.” [54]

Our hypothesis is that the learning results are consequences of the process the student is performing. Of course, visualization or good verbal description can help the student to cope with the topic at hand, but no visualization or verbal depiction can influence learning without the process. Looking at the illustration is fruitless without seeing what is happening. This is why we have emphasized algorithm simulation - a technique that encourages the student to interact with the visualization. It is a process that cannot be done without understanding the underlying concept and thus, forces the student to see the dynamics of an algorithm. In order to prove this hypothesis, however, many new studies are still needed. This is, however, an ongoing process and, for example, the working group in the 7th annual conference on *Innovation and Technology in Computer Science Education* launched a series of studies aiming to prove that learner engagement is the most crucial factor in learning [89].

9.5 Other Research Areas and Techniques

Although at least Matrix has its applications outside of educational scenarios – for instance in software engineering – we are unaware of research that has systematically traced such systems or has given a framework for such an evaluation. Thus, the scope of our evaluation above is intentionally limited to experimental studies of the effectiveness of the target systems in educational use. Moreover, controlled experimentations and quantitative observational studies are just one of the several empirical evaluation techniques that the target systems are subjected to during the past ten years. Thus, we specify three other empirical techniques that have been employed by past research even though the results and conclusions of these studies are out of the scope of this thesis. Still, these are tasks that must be done "behind the scenes" in order to prepare the visualizations for students use and integrating the systems into a course curriculum. Interestingly, these are the same areas that Hundhausen *et al.* [54] suggest for alternative empirical methods for future directions in AV technology meta-studies.

1. Questionnaires and surveys — regularly applied technique to gather responses to a set of questions in which we have been interested.

2. Usability test — especially the tests to identify and fix problems in the user interface of the new Matrix framework and its applications. Usually, a small number of participants interacting with the system was monitored while they completed a predefined task.

3. Ethnographic field techniques — both of the systems are subjected to several qualitative field studies in a naturalistic settings. The behavior of the participants were observed or they were interviewed during or after the study.

Each of these alternative methods has played a valuable role in helping us to gain insight into the target systems effectiveness. For example, the questionnaires on Data Structures and Algorithms course showed that each year the students have found the TRAKLA system to be the most effective single instrument that aided their learning. However, the new TRAKLA2 environment got even better results while it was applied in test use this year. 15 new exercises for TRAKLA2 were developed and tested with some 500 students. Likewise, usability tests helped us to identify and fix problems that may have otherwise possibly prevented the students from interacting properly with the user interface in the first place. Occasional ethnographic field surveys have been a valuable aid to identifying how and why the systems might be effective in an educational context. However, in this thesis we are focused on the research we have actually reported in the form of published paper [69, 80]. Therefore, these items are not discussed any further.

Chapter 10

Discussion

This thesis has tried to lay a foundation for visual algorithm simulation frameworks in the context of data structures and algorithms. Two separate systems have been introduced employing algorithm simulation functionality. The older, TRAKLA, has been in production use over ten years. During the period, it has been subjected to several experimental studies revealing its educational effectiveness. Subsection 10.1 concludes the discussion briefly. On the other hand, the old system suffers from several drawbacks. For example, the creation of new exercises for the system is a very time consuming process. This, among other things, led us to develop a new system which design allows more flexible development of not only new exercises but also other algorithm simulation applications. The new application framework includes all the capabilities of the older one. The design, however, allows many other features. We summarize the educational benefits of automatic assessment exercises in Subsection 10.1. The benefits of the new design are discussed in Subsection 10.2 and Subsection 10.3. Finally, in Subsection 10.4 we address some future research topics that remains to be studied.

10.1 Automatic Assessment

The idea of using virtual learning environments to enhance the learning outcome has become widely accepted. We have demonstrated a setup in which there is no significant difference in the learning results between students doing their homework exercises in a virtual learning environment and students attending a traditional classroom session. In addition, the commitment to the course (low final drop out), is almost equal in both cases even though students working in the virtual environment seems to drop the course earlier than students working in the classroom. However, the virtual learning environments can provide advantages in terms of Automatic Assessment (AA) which the traditional teaching methods cannot offer.

The principal benefit of AA, based on our experience, is that it enables one to *set up enough compulsory exercises* for the students and to *give them feedback* on their work on large scale. Our experience shows that voluntary exercises do not motivate students enough to reach the required level of skills. Compulsory assignments do not provide much more unless we can monitor students' progress regularly and, more important, give them feedback on their performance. This would be possible in weekly classroom exercises, if we only had enough classrooms, time and instructors, which unfortunately is not the case. By means of AA, we can partly solve the feedback problem even though we do not argue that computerized feedback would generally match expert human feedback. In some cases, however, it does, as we have demonstrated in this thesis. In general, on the other hand, computerized feedback is far better than merely dealing out with model solutions without any feedback on students' performance.

The second benefit of AA is that the computer can *give feedback at any time and place*. Thus, it applies particularly well to virtual courses, and in fact, it has turned most of our courses partially virtual. However, we do have lectures and classroom sessions as well, but the students do most of their self-study on the Web.

The third benefit is that we can allow the students to *revise their submissions* freely based on the feedback. By setting up the assignments in such a way that pure trial-and-error method is no option, the student is pushed to rethink his or her solution anew, and to make hypotheses about what is wrong. This is one of the conventions that constructivism [93] suggests we should put into effect. Moreover, it should be noted that in classroom sessions, such a situation is rarely achieved for every student, since the teacher cannot follow and guide the work of each individual at the same time.

The fourth benefit is that we can *personalize* exercises, *i.e.*, all students have different assignments, which discourages plagiarism and encourages natural co-operation while solving problems. With human teaching resources, it is practically impossible to achieve this in large courses.

Finally, AA tools save time and money. We feel, however, that they represent an extra resource that helps us to teach better. Moreover, they allow us to direct human resources to work which cannot be automated, for instance, personal guidance and feedback for more challenging exercises such as design assignments.

10.2 Matrix

In this thesis, we have introduced a novel application framework for algorithm simulation tools. The framework has been applied to renewing the learning environment on a Data Structures and Algorithms course and enhance the possibilities to release simulation exercises beyond the scope of our prior system while still maintaining the good qualities of the old environment. Moreover, we have also paid attention to the process of developing new assignments in order to allow creation of new exercises by only focusing on the algorithms needed to evaluate the student's answers. The design of the framework is based on a formal mathematical model of data structures and algorithms. The main purpose was to enhance the capabilities of algorithm simulation — a novel concept that fosters the interaction between the user and the visualization environments.

Three main concepts can be identified within the system. First, the system is based on algorithm visualization in which common reusable visualizations of all the well known fundamental data types can be attached to any underlying objects within an executable run-time environment. Second, the execution of an algorithm can be memorized in order to animate the running algorithm that manipulates these objects. Third, algorithm simulation allows the user to interact with the system and to change the content of the data structures in terms of direct manipulation.

Summary

Finally, we conclude the main dimensions of the new Matrix framework. Three main characteristics $C = \{V,A,S\}$ can be identified from visual algorithm simulation systems: visualization (V), animation (A), and simulation (S). On the other hand, we can look at these characteristics from five separate point of views $P = \{U,V,P,D,A\}$: user (U), visualizer (V), programmer (P), developer (D), and application (A). Thus, the new framework has contributions for all of the 15 subfields that are the outcome of the cartesian product $C \times P = \{V,A,S\} \times \{U,V,P,D,A\}$ of these two dimension. We highlight some of these aspects in Table 10.1.

Moreover,

1. the system makes distinction between the implementation of an actual data structure and its visualizations;
2. one data structure can have many conceptual visualizations;
3. one conceptual visualization can be used for displaying many different data structures;

Table 10.1: Summary of Matrix framework. The three main themes, visualization (V), animation (A), and simulation (S) are examined from five separate points of views: User (U), Visualizer (V), Programmer (P), Developer (D), and Application (A).

PoV	**V**	**A**	**S**
U	Static conceptual displays (Fig 4.1 on page 46).	Dynamic conceptual displays (Fig 6.1 on page 78).	**Visual algorithm simulation** (VAS).
V	Creation of static and dynamic illustrations in terms of VAS.		
P	Reusability & visualization of user made algorithms.		Visual testing and debugging.
D	Fast prototyping of algorithm animation and simulation applications.		
A	Scripting facilities (Example 5.1.1 on page 55).	Export facilities for static and dynamic animations (TeXdraw and SVG).	Application framework for algorithm simulation applications (*e.g.*, **TRAKLA2**).

4. the framework enables visualization, algorithm animation, and algorithm simulation of user-made code without restrictions concerning object-oriented design methods; and

5. the prototype demonstrates the exercise interface and the possibility to automatically assess the exercises in the new TRAKLA2 learning environment.

10.3 The Goal Revisited

We are now ready to revisit the goals described in Section 1.2.2. First, the new TRAKLA2 system should not suffer from those problems identified with the old TRAKLA system. As the experimental study shows, this goal has been met; actually, we managed to produce new kinds of visual exercises for the new system more rapidly than we could produce verbal exercises for the old system. Second, many new features should be included, as we already have summarized. As we have pointed out, the most crucial part of the design of the new SV system is the visualization methods used. For example, *specification style* and the chosen *connection technique* both have a very important role to play. As we have demonstrated in this thesis, all the methods have their advantages and disadvantages depending on the teaching and learning style. Thus, we have decided to provide several different methods to cooperate, including

1. algorithm simulation,
2. user-coded algorithms connected by interfaces, and
3. user-coded programs connected by probes.

In the case of algorithm simulation, the user can *simulate* an algorithm, thus the user specifies the animation in terms of *algorithm simulation*. User-coded algorithms refer to *algorithm visualization* and *concept animation*. Moreover, it should also be possible to produce user-coded programs by reusing existing components called *probes*, thus providing also a set of *self-animating* components. This idea can possibly be developed further by including *program animation* capabilities in the future. Thus, we can provide a framework also for approaches other than algorithm simulation exercises to promote AV in education.

10.4 The Future

At the moment, we feel that all the capabilities the framework supports are only a very small subset of all those features possible to include. Many new ideas remain and are discussed very briefly here. Thus, in the near future at least the following tasks should be completed:

1. implementation of all the exercises the old TRAKLA system includes, and complete the set of example exercises derived from the algorithm simulation exercise taxonomy,
2. incorporate the system in the existing WWW-TRAKLA learning environment,
3. apply the framework to other applications than TRAKLA2, for example, to a demonstration tool for teacher,
4. incorporated visual debugger into the framework,

5. enhanced recording facilities for algorithm simulation,
6. establishment of global library of exercises and demonstrations (requires possibly a web file system),
7. taking full advantage of the several notations introduced in Chapter 5 and 6 to create code and visualizations of new nested FDTs directly from the definitions,
8. implementation of internal pseudo-programming language interpreter (in order to provide on-line programming without compilation),
9. combining the abilities of several animation and visualization tools in terms of shared common import and export format (possibly by sharing an XML format definition for common data structures),
10. further customization of representations, and
11. more research on program animation capabilities.

The first three tasks are the primary research topics that complete the work needed to create the electronic exercise book and the learning environment. On the other hand, the scope and content of the new system should provide a generalized platform-independent framework that is capable of visualizing even large examples. The system should be rich enough to provide flexibility to develop the framework and its applications further. For example, elision control is essential since its value is emphasized especially in large examples. The application framework should also provide tools for developing proper user interfaces for new applications. Nevertheless, the effectiveness of such systems cannot be evaluated until an application is actually implemented. Thus, the effectiveness of the new framework has been beyond the scope of this thesis due to its subjective nature, but should be promoted in future studies. However, since we have demonstrated that the new TRAKLA2 environment includes all the features of the old TRAKLA, we conclude that these results similarly apply to the new framework in the future. Moreover, the framework itself could be developed further to provide an even more flexible environment for software visualization. Finally, the framework provides a platform to start new research projects, for example, to produce visualizations for arbitrary graphs.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] D. Albert. *Quantum Mechanics and Experience*. Cambridge, MA, Harvard University Press, 1992.

[3] O. Astrachan and S. H. Rodger. Animation, visualization, and interaction in CS1 assignments. In *The proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 317–321, Atlanta, GA, USA, 1998. ACM.

[4] O. Astrachan, T. Selby, and J. Unger. An object-oriented, apprenticeship approach to data structures using simulation. In *Proceedings of Frontiers in Education*, pages 130–134, 1996.

[5] R. M. Baecker. Sorting out of sorting. Narrated colour videotape, 30 minutes, 1981.

[6] R. M. Baecker. *Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science*, chapter 24, pages 369–381. The MIT Press, Cambridge, MA, 1998.

[7] R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, pages 261–265, New Orleans, LA, USA, 1999. ACM.

[8] G. D. Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.

[9] M. Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[10] M. Ben-Ari, editor. *Second Program Visualization Workshop*. University of Aarhus, Department of Computer Science, HorstrupCentret, Denmark, 2002.

[11] M. Ben-Ari, N. Myller, E. Sutinen, and J. Tarhio. Perspectives on program animation with jeliot. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 31–45, Dagstuhl, Germany, 2001. Springer.

[12] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

[13] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. M. Zin. Ceilidh: A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*, Vienna, Austria, 1993.

[14] M. Birch, C. Boroni, F. Goosey, S. Patton, D. Poole, C. Pratt, and R. Ross. DYNALAB: A synamic computer science laboratory infrastructure featuring program animation. In *SIGCSE Bulletin*, volume 27, pages 29–33. ACM, 1995.

[15] C. M. Boroni, T. J. Eneboe, F. W. Goosey, J. A. Ross, and R. J. Ross. Dancing with dynalab. In *In 27th SIGCSE Technical Symposium on Computer Science Education*, pages 135–139. ACM, 1996.

[16] S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *The proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 139–143. ACM, 2000.

[17] M. Brown, J. Domingue, B. Price, and J. Stasko. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[18] M. H. Brown. *Algorithm Animation*. MIT Press, Cambridge, Massachussets, 1988.

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

[20] M. H. Brown. Zeus: a system for algorithm animation and multi-view editing. In *Proceedings of IEEE Workshop on Visual Languages*, pages 4–9, Kobe, Japan, 1991.

[21] M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, 1992.

[22] M. H. Brown and R. Raisamo. JCAT: Collaborative active textbooks using Java. *Computer Networks and ISDN Systems*, 29(14):1577–1586, 1997.

[23] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *Proceedings of SIGGRAPH'84*, pages 177–186. ACM, 1984.

[24] M. Bäsken and S. Näher. GeoWin a generic tool for interactive visualization of geometric algorithms. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 88–100, Dagstuhl, Germany, 2001. Springer.

[25] P. Carlson, M. Burnett, and J. Cadiz. A seamless integration of algorithm animation into a visual programming language. In *Proceedings of Working Conference on Advanced Visual Interfaces (AVI'96)*, pages 290–299, 1996.

[26] G. Cattaneo, G. F. Italiano, and U. Ferraro-Petrillo. CATAI: Concurrent algorithms and data types animation over the internet. *Journal of Visual Languages and Computing*, 13(4):391–419, August 2002.

[27] S. Chang. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, 1987.

[28] C. Chen and Y. Yu. Empirical studies of information visualization: A meta-analysis. *International Journal of Human-Computer Studies*, 53:851–866, 2000.

[29] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[30] A. J. Cowling. Teaching data structures and algorithms in a software engineering degree: Some experience with java. In *Proceedings of the 14th Conference on Software Engineering Education and Training*, pages 247–257, Charlotte, North Carolina, USA, 2001. IEEE.

[31] K. C. Cox and G.-C. Roman. A characterization of the computational power of rule-based visualization. *Journal of Visual Languages and Computing*, 5(1):5–27, March 1994.

[32] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, April 2000.

[33] M. E. Crosby and J. Stelovsky. From multimedia instruction to multimedia evaluation. *Journal of Educational Multimedia and Hypermedia*, 4:147–162, 1995.

[34] A. Cypher. *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, Mass., 1993.

[35] C. Demetrescu and I. Finocchi. A technique for generating graphical abstractions of program data structures. In *Proceedings of the 3rd International Conference on Visual Information Systems*, pages 785–792, Amsterdam, 1999. Springer.

[36] C. Demetrescu and I. Finocchi. Smooth animation of algorithms in a declarative framework. *Journal of Visual Languages and Computing*, 12(3):253–281, 2001.

[37] C. Demetrescu, I. Finocchi, and J. Stasko. Specifying algorithm visualizations: Interesting events or state mapping? In S. Diehl, editor, *Software Visualization: International Seminar*, pages 16–30, Dagstuhl, Germany, 2001. Springer.

[38] S. Diehl, editor. *Software Visualization*, volume 2269. Springer Verlag, Dagstuhl, Germany, 2002.

[39] S. Diehl, C. Görg, and A. Kerren. Animating algorithms live and post mortem. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 46–57, Dagstuhl, Germany, 2001. Springer.

[40] J. English and P. Siviter. Experience with an automatically assessed course. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00*, pages 168–171, Helsinki, Finland, 2000. ACM.

[41] N. Faltin. Structure and constraints in interactive exploratory algorithm learning. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 213–226, Dagstuhl, Germany, 2002. Springer.

[42] R. M. Felder and L. K. Silverman. Learning styles and teaching styles in engineering education. *Engineering Education*, 78(7):674–681, 1988.

[43] R. Fleischer and L. Kucera. Algorithm animation for teaching. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 113–128, Dagstuhl, Germany, 2001. Springer.

[44] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*. Elsevier Science Publishers, 1984.

[45] P. A. Gloor. *User interface issues for algorithm animation*, chapter 11, pages 145–152. The MIT Press, Cambridge, MA, 1998.

[46] H. H. Goldstein and J. von Neumann. Planning and coding problems of an electronic computing instrument. *Collected Works*, pages 80–115, 1947.

[47] O. Grillmeyer. An interactive multimedia textbook for introductory computer science. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 286–290. ACM Press, 1999.

[48] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user algorithms on the web. In *Proceedings of Symposium on Visual Languages*, pages 360–367, Isle of Capri, Italy, 1997. IEEE.

[49] S. R. Hansen, N. H. Narayanan, and D. Schrimpsher. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2(1), May 2000.

[50] R. R. Henry, K. M. Whaley, and B. Forstall. The university of washington illustrating compiler. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233, 1990.

[51] C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 46–50. ACM Press, 2002.

[52] T. Hill, J. Noble, and J. Potter. Scalable visualization of object-oriented systems with ownership trees. *Journal of Visual Languages and Computing*, 13(3):319–339, 2002.

[53] C. Hundhausen and S. A. Douglas. SALSA and ALVIS: A language and system for constructing and presenting low fidelity algorithm visualizations. In *Visual Languages*, pages 67–68, 2000.

[54] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.

[55] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. In *The proceedings of the 31st ACM SIGCSE Technical Symposium on Computer Science Education*, pages 6–10, Austin, Texas, USA, 2000. ACM.

[56] J. Hyvönen and L. Malmi. TRAKLA – a system for teaching algorithms using email and a graphical editor. In *Proceedings of HYPERMEDIA in Vaasa*, pages 141–147, 1993.

[57] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proceedings of 28th ACM SIGCSE Tech. Symposium on Computer Science Education*, pages 335–339, San Jose, California, USA, 1997. ACM.

[58] D. J. Jarc, M. B. Feldman, and R. S. Heller. Assessing the benefits of interactive prediction using web-based algorithm animation courseware. In *The proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages pp. 377–381, Austin, Texas, 2000. ACM Press, New York.

[59] P. Kabal. TeXdraw – PostScript drawings from TeX. Web page, 1993.

[60] A. Kerren and J. T. Stasko. Algorithm animation. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 1–15, Dagstuhl, Germany, 2001. Springer.

[61] K. C. Knowlton. L^6: Bell telephone laboratories low-level linked list language. 16 mm black and white sound film, 16 minutes, 1966.

[62] K. C. Knowlton. L^6: Part II. an example of L^6 programming. 16 mm black and white sound film, 30 minutes, 1966.

[63] D. E. Knuth. Computer-drawn flowcharts. *Communications of the ACM*, 6:555–563, 1963.

[64] A. Korhonen. World wide web (www) tietorakenteiden ja algoritmien tietokoneavusteisessa opetuksessa. Master's thesis, Helsinki University of Technology, 1997.

[65] A. Korhonen. *Algorithm Animation and Simulation.* Licenciate's thesis, Helsinki University of Technology, 2000.

[66] A. Korhonen and L. Malmi. Algorithm simulation with automatic assessment. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 160–163, Helsinki, Finland, 2000. ACM.

[67] A. Korhonen and L. Malmi. Proposed design pattern for object structure visualization. In *The proceedings of the First Program Visualization Workshop – PVW 2000*, pages 89–100, Porvoo, Finland, 2001. University of Joensuu.

[68] A. Korhonen and L. Malmi. Matrix – concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, May 2002. ACM.

[69] A. Korhonen, L. Malmi, P. Myllyselkä, and P. Scheinin. Does it make a difference if students exercise on the web or in the classroom? In *Proceedings of The 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*, Aarhus, Denmark, 2002. ACM.

[70] A. Korhonen, L. Malmi, P. Mård, H. Salonen, and P. Silvasti. Electronic course material on data structures and algorithms. In *Proceedings of the Second Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 16–20, October 2002.

[71] A. Korhonen, L. Malmi, J. Nikander, and P. Silvasti. Algorithm simulation – a novel way to specify algorithm animations. In M. Ben-Ari, editor, *Proceedings of the Second Program Visualization Workshop*, pages 28–36, HorstrupCentret, Denmark, June 2002.

[72] A. Korhonen, J. Nikander, R. Saikkonen, and P. Tenhunen. Matrix – algorithm simulation and animation tool. http://www.cs.hut.fi/Research/Matrix/, November 2001.

[73] J. F. Korsh and R. Sangwan. Animating programs and students in the laboratory. In *Proceedings of Frontiers in Education*, pages 1139–1144, 1998.

[74] W. Kreutzer. *System Simulation Programming Styles And Languages*. Addison-Wesley, 1986.

[75] P. LaFollette, J. Korsh, and R. Sangwan. A visual interface for effortless animation of C/C++ programs. *Journal of Visual Languages and Computing*, 11(1):27–48, 2000.

[76] S.-P. Lahtinen, T. Lamminjoki, E. Sutinen, J. Tarhio, and A.-P. Tuovinen. Towards automated animation of algorithms. In N. Thalmann and V. Skala, editors, *Proceedings of Fourth International Conference in Central Europe on Computing Graphics and Visualization*, pages 150–161. University of West Bohemia, Department of Computer Science, 1996.

[77] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, USA, 1982.

[78] A. Lawrence, A. Badre, and J. T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages, St. Louis, MO*, pages 48–54, 1994.

[79] C. L. Liu. *Elements of discrete mathematics*. McGraw-Hill, 1985.

[80] L. Malmi, A. Korhonen, and R. Saikkonen. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *Proceedings of The 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*, pages 55–59, Aarhus, Denmark, 2002. ACM.

[81] D. V. Mason and D. M. Woit. Providing mark-up and feedback to students with online marking. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 3–6, New Orleans, LA, USA, 1999. ACM.

[82] R. Mayer. Multimedia learning: Are we asking the right questions? *Educational psychologist*, 32(1):1–19, 1997.

[83] B. P. Miller. What to draw? When to draw? An essay on parallel program visualization. *Journal of Parallel and Distributed Computing*, 18(2):265–269, 1993.

[84] F. Modugno, A. T. Corbett, and B. A. Myers. Graphical representation of programs in a demonstrational visual shell – an empirical evaluation. *Transactions on Computer-Human Interaction*, 4(3):276–308, 1997.

[85] S. Mukherjea and J. T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *Transactions on Computer-Human Interaction*, 1(3):215–244, 1994.

[86] B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[87] B. A. Myers, R. Chandhok, and A. Sareen. Automatic data visualization for novice pascal programmers. In *IEEE Workshop on Visual Languages*, pages 192–198. IEEE, 1988.

[88] T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the SIGCSE Session*, pages 109–113, Austin, Texas, March 2000. ACM.

[89] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodgers, and J. Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

[90] J. Noble. Visualising objects: Abstraction, encapsulation, aliasing, and ownership. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 58–72, Dagstuhl, Germany, 2002. Springer.

[91] R. J. Noble and L. J. Groves. Tarraingím – a program animation environment. *New Zeland Journal of Computing*, 4(1), December 1992.

[92] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications ACM*, 38(6):33–44, 1995.

[93] D. C. Phillips. *Constructivism in Education. Opinions and Second Opinions on Controversial Issues. Ninety-ninth Yearbook of the National Society for the Study of Education. Part 1.* The University of Chicago Press, Chicago, Illinois, USA, 2000.

[94] W. Pierson and S. Rodger. Web-based animation of data structures using JAWAA. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 267–271, Atlanta, GA, USA, 1998. ACM.

[95] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[96] R. Rasala. Automatic array algorithm animation in C++. In *The Proceedings of the 30th SIGCSE Technical Symposium on Computer science education*, pages 257–260, New Orleans, LA, USA, 1999. ACM.

[97] K. A. Reek. The TRY system or how to avoid testing student programs. In *Proceedings of SIGCSE'1989*, pages 112–116. ACM, 1989.

[98] G.-C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

[99] R. J. Ross and M. T. Grinder. Hypertextbooks: Animated, active learning, comprehensive teaching and learning resource for the web. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 269–283, Dagstuhl, Germany, 2002. Springer.

[100] G. Rößling. The ANIMAL algorithm animation tool. In *Proceedings of the 5th Annual SIGCSE/SGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00*, pages 37–40, Helsinki, Finland, 2000. ACM.

[101] G. Rößling and B. Freisleben. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

[102] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of The 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*, pages 133–136, Canterbury, United Kingdom, 2001. ACM.

[103] D. A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6(5):28–36, 1989.

[104] C. A. Shaffer. *A practical introduction to data structures and algorithms analysis*. Prentice-Hall, 1998.

[105] A. Snyder. Encapsulation and inheritance in object-oriented languages. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 38–45, Portland, OR, 1986. ACM.

[106] J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning? an empirical study and analysis. In *Proceedings of the INTERCHI'93 Conference on Human Factors on Computing Systems*, pages 61–66, Amsterdam, Netherlands, 1993. ACM.

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

[108] J. T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of Conference on Human Factors and Computing Systems*, pages 307–314, New Orleans, Louisiana, USA, 1991. ACM, New York.

[109] J. T. Stasko. Using student-built algorithm animations as learning aids. In *The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 25–29, San Jose, CA, USA, 1997. ACM.

[110] J. T. Stasko. *Building Software Visualizations through Direct Manipulation and Demonstration*, chapter 14, pages 103–118. MIT Press, Cambridge, MA, 1998.

[111] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[112] L. Stern, H. Søndergaard, and L. Naish. A strategy for managing content complexity in algorithm animation. In *Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and technology in computer science education, ITiCSE'99*, pages 127–130, Kracow, Poland, 1999. ACM Press.

[113] E. Sutinen and J. Tarhio. Teaching to identify problems in a creative way. In *Proceedings of the 31st ASEE/IEEE Frontiers in Education Conference*, page p. T1D813. IEEE, 2001.

[114] S. Takahashi, S. Matsuoka, A. Yonezawa, and T. Kamada. A general framework for bi-directional translation between abstract and pictorial data. In *Proceedings of the 4th annual ACM symposium on User interface software and technology*, pages 165–174. ACM Press, 1991.

[115] S. Turkle and S. Papert. Epistemological pluralism and the revaluation of the concrete. *Constructionism*, pages 161–192, 1991.

[116] D. Ungar and R. B. Smith. SELF: The power of simplicity. *Lisp And Symbolic Computation*, 4:187–205, June 1991.

[117] A. Zeller. Animating data structures in DDD. In *The proceedings of the First Program Visualization Workshop – PVW 2000*, pages 69–78, Porvoo, Finland, 2001. University of Joensuu.

Appendix A

Binary Search

The set of basic blocks in linear code, derived from the following binary search function, are illustrated graphically in Figure 4.3 on page 52.

```
#include <stdio.h>

int search(int *table, int low, int high, int key) {
  int try;

  while (low < high) {
    try = (low + high)/2;
    if (key < table[try])
      high = try - 1;
    else if (key > table[try])
      low = try + 1;
    else
      return try;
  }
  return low;
}
```