Ari Korhonen

Algorithm Animation and Simulation

Licentiate’s thesis: May 31, 2000

Supervisor
Professor Eljas Soisalon-Soininen

Instructor
Acting Professor Lauri Malmi
Understanding data structures and algorithms, both of which are abstract concepts, is an integral part of elementary computer science education. On the other hand, people usually have difficulties in understanding abstract concepts and processes such as procedural encoding of algorithms and data structures. One way to improve their understanding is to provide visual examples to make the abstract concepts more concrete.

This thesis presents the design and implementation for an 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, which we call algorithm simulation, the framework for exploring and understanding algorithms and data structures is exhibited.

We argue that the combination of algorithm animation and algorithm simulation could be a valuable teaching aid for instructors, if only the examples could be created easily and rapidly. Moreover, we want to explore the capabilities such a system may offer. For example, this kind of framework can provide a tool for delivering, representing, solving, and submitting tracing exercises. In addition, it may include automatic assessment of such exercises to give meaningful feedback for students on large-scale courses.

**Keywords:** Algorithm Animation, Algorithm Simulation, User Controlled Simulation, Software Visualization, Automatic Assessment
### Tekijä ja työn nimi:
Ari Korhonen
Algoritmianimaatio ja -simulaatio

### Päivämäärä:
31. Toukokuuta 2000

### Sivumäärä:
viii + 94

### Osasto:
Tietotekniikan osasto

### Professuuri:
Tik-106

### Työn valvoja:
Professori Eljas Soisalon-Soininen

### Työn ohjaaja:
Ma. professori Lauri Malmi

Tietorakenteet ja algoritmit kuuluvat tietojenkäsittelytieteen keskeiseen oppisisältöön. Tämä oppisisältö on huomattavan käsitteellistä ja ihmisillä on yleensä vaikeuksia hahmottaa käsitteellisiä asioita ja prosesseja. Eräs tapa auttaa hahmottamaan tällaisia monimuutkaisia asioita on tarjota havainnollistavia esimerkkejä.

Tässä työssä suunnitellaan ja toteutetaan sovelluskehys, joka yhdistää kahden eri tutkimussuunnan tuloksia. Toisaalta työssä hyödynnetään algoritmianimaatiota, jolla havainnollistetaan tietokoneohjelmien perustoinnallisuutta ja käyttäytymistä. Tähän yhdistetynä algoritmisimulaatio mahdollistaa algoritmin ja tietorakenteiden toiminnan kokeellisen tutkimisen ja sitä kautta niiden toiminnan ymmärtämisen.


### Avainsanat:
Algoritmianimaatio, algoritmisimulaatio, käyttäjäperustainen simulointi, ohjelmien havainnollistaminen, automaattinen tehtävien tarkastus
Preface

This study was carried out at the Laboratory of Information Processing Science, Helsinki University of Technology. First I owe a great debt of gratitude to Professor Lauri Malmi for first arousing my interest in this field and for his thorough and enthusiastic supervision during this work.

I am sincerely grateful to Professor Eljas Soisalon-Soininen, Head of the Laboratory of Information Processing Science, for his positive attitude and for creating the excellent atmosphere for scientific work.

It is my pleasure to thank Professor Jorma Tarhio for the constructive criticism during the preparation of the manuscript. My special appreciation goes to him.

I am also indebted to all my colleagues and associates. Especially I am thankful to Vesa Hirvisalo and Esko Nuutila for their help with the creation of more realistic examples and to Mrs. Elizabeth Heap-Talvela for help with the English language. To these and to everyone else who contributed I wish to express my thanks.

Finally I wish to thank my family for their love and support. I dedicate this work to them.

Otaniemi, May 31, 2000

Ari Korhonen
# Contents

1 Introduction .................................................. 1
   1.1 Background ............................................. 2
   1.2 Electronic exercise book ................................. 3
      1.2.1 Goal ............................................. 4
   1.3 Contributions ........................................... 5
   1.4 Organization of this Thesis ............................. 5

2 Previous Work .................................................. 6
   2.1 Terminology ............................................. 6
      2.1.1 Learning Environment ............................. 6
      2.1.2 Simulation ....................................... 7
      2.1.3 Software Visualization ........................... 8
      2.1.4 Automatic Assessment and Feedback ............... 9
   2.2 Taxonomy ................................................ 9
      2.2.1 Categories ....................................... 9
      2.2.2 Roles ........................................... 10
      2.2.3 Symbols ......................................... 11
   2.3 Example systems ......................................... 11
   2.4 Evaluation .............................................. 13
      2.4.1 Scope ........................................... 13
      2.4.2 Content ......................................... 14
CONTENTS

2.4.3 Form ................................................................. 16
2.4.4 Method ............................................................ 17
2.4.5 Interaction .......................................................... 19
2.4.6 Effectiveness ....................................................... 20
2.5 Conclusion ............................................................ 23
2.5.1 The Goal revisited ................................................ 24

3 Theoretical Point Of View ........................................ 26

3.1 Prerequisites .......................................................... 26
3.1.1 Basics of Set Theory .............................................. 27
3.1.2 Basics of Graph Theory ......................................... 28
3.2 Algorithms ............................................................ 28
3.3 Data Types ............................................................ 29
3.3.1 Abstract Data Types .............................................. 30
3.3.2 Fundamental Data Types ........................................ 31
3.3.3 Data Structures .................................................... 32
3.4 Case Study: Binary Search Tree .................................. 33

4 Programmer Point Of View ........................................ 37

4.1 Methods and Tools ................................................... 38
4.2 Memory Structures ................................................... 38
4.3 Storage Structures .................................................... 39
4.3.1 Table ................................................................. 40
4.3.2 Linked List ........................................................ 40
4.3.3 Trees ................................................................. 41
4.3.4 Graphs .............................................................. 43
4.4 Extended Structures .................................................. 43
4.5 Case Study: Binary Search Tree revisited ....................... 44
4.5.1 Reusability revisited ............................................. 46

5 Visualizer Point Of View ........................................... 50

5.1 Overview ................................................................. 50
  5.1.1 Two approaches ................................................... 51
  5.1.2 Organization of this chapter ..................................... 51
5.2 Simulation Model ...................................................... 51
  5.2.1 Theoretical framework revisited ................................. 52
  5.2.2 The Model .......................................................... 54
5.3 Data Structure Visualization ....................................... 54
  5.3.1 Visual Components ................................................. 54
  5.3.2 Ready-made visualization concept ............................... 58
  5.3.3 Interfaces ........................................................... 61
  5.3.4 Building higher-level representations ........................... 66
  5.3.5 Implementation .................................................... 68
  5.3.6 Summary ............................................................ 68
5.4 Algorithm Animation ................................................ 69
5.5 User Controlled Simulation ......................................... 70
  5.5.1 Read and Write Operations ...................................... 72
  5.5.2 Insert Operation .................................................... 73
  5.5.3 Miscellaneous Operations ....................................... 75
5.6 Summary ................................................................. 75

6 Educational Point Of View ........................................... 77

6.1 Overview ................................................................. 77
  6.1.1 Teacher and Student Points of View ............................ 77
6.2 Demonstration Tool .................................................... 77
  6.2.1 Visual Examples ..................................................... 78
6.2.2 Classroom demonstration ........................................... 78
6.3 Electronic exercise book ............................................. 78
  6.3.1 Laboratory Experiments ......................................... 79
  6.3.2 Object-Oriented Programming and Design ....................... 79
  6.3.3 Automatic Assessment ........................................... 80

7 Discussion .......................................................... 82
  7.1 Contributions in terms of Taxonomy ................................. 83
    7.1.1 Scope .......................................................... 83
    7.1.2 Content ....................................................... 83
    7.1.3 Form .......................................................... 84
    7.1.4 Method ....................................................... 84
    7.1.5 Interaction and Effectiveness .................................. 85
    7.1.6 Summary ..................................................... 86
  7.2 The Future ...................................................... 86

A Binary Search ...................................................... 94
Chapter 1

Introduction

Let us assume we have a visualization for some data structure and an algorithm that manipulates the data structure. Obviously, the visualization could be used for animation purposes. Animation is a sequence of visual snapshots of the data structure. During the execution of the algorithm the state of the data structure changes. These states can be visualized for the user, one after another to animate the algorithm. In addition, the user may have some kind of control of the process, so he can interact with the system in order to stop and continue the animation. This kind of step-by-step animation could be compared to visual debugging [33] 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 simply algorithm animation.

Furthermore, we can allow the user to perform operations of his or her own. Thus, instead of letting the algorithm to execute instructions and manipulate the data structure, we can allow the user to take control over the manipulation process. If the user processes the data structure as the algorithm did in the previous example, we say that the user simulates the algorithm. We refer to such an algorithm simulation as user controlled simulation of an algorithm. See Figure 1.1.

Figure 1.1: General overview of the algorithm animation and simulation.
Data structures and algorithms are important core issues in computer science education. Because they are often complex concepts, visual debugging and user controlled simulation are both attractive methods to be considered as learning aids. From the pedagogical point of view, however, a plain tool for “playing and having fun”, is not good enough. At least we would like to be sure whether some kind of progress in learning has taken place. We have to have some kind of pedagogical environment where we can give and obtain feedback on the student’s performance. On the other hand, the vast masses of students on the courses on basic data structures and algorithms have led to the situation in which giving individual guidance for a single student is impossible. Thus, some kind of automatic instructor would be useful. This might sound far too ambitious. But, as we will see, it is not so difficult to improve the concept above to support exercises in which automatically generated feedback is possible. We call such a process automatic assessment of exercises (Fig. 1.1).

1.1 Background

In this thesis, we describe the elements of an automatic assessment system, called Object-TRAKLA, whose origin can be traced back to the early 1990’s. At that time, a project was launched to look into the possibility to automatically evaluate students’ answers to exercises for the course on data structures and algorithms [20]. 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 special purpose system called TRAKLA\(^1\). There was, however, a natural way to improve the concept by developing a separate graphical user interface called Tred [24]. This user interface had the capability to hide unnecessary format details and to turn in the exercise after recording the user controlled simulation process performed by the student. This graphical user interface was also capable of representing the simulation sequence as an algorithm animation. Moreover, TRAKLA has evolved to be a crucial part of a more complex open\(^2\) distributed\(^3\) learning environment, called WWW-TRAKLA, that is a software system in which the whole set of common telematic tools (such as email, newsgroups, WWW forms, and some special purpose tools like Object-TRAKLA) are combined. These tools form the learning environment that is very flexible and easy to maintain [25].

As stated before [25, 26], we repeat that TRAKLA together with Tred has been a valuable teaching tool during the past decade. It has been in production use since 1991 in a course in which yearly enrollment has been 400-700 students each solving

---

\(^1\)TRAKLA is an abbreviation of the Finnish words “Tietorakenteet ja Algoritmit; KotiLaskujen Automaattinen tarkastaminen”.

\(^2\)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 which conform to well-defined interfaces.

\(^3\)This is purely a technical term which refers to the concept of autonomous processing elements which cooperate by sending messages to each other.
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. provide exercises with immediate feedback,
4. produce statistics about students’ overall performance,
5. provide the opportunity to revise incorrect answers,
6. research the overall performance of a given class, and
7. encourage natural discussion about topics between students.

In addition, we stress here that a student has to think about the revised solution anew for each new submission, since the solution space for exercises is simply too large for using any trial-and-error method. The immediate feedback is a particularly important feature. The student receives an email for every submission within a few minutes, containing additional information about his or her performance. This information is based on the automatic assessment of the submitted exercises.

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

Many new ideas have been proposed to improve the concept further. Unfortunately, the design and architecture of the current system does not meet the requirements of adding these new features. Since similar systems do not exist, we have concluded that we should develop a very new system having the good qualities of TRAKLA and Tred in one single framework. The first phase of this work is represented in this thesis and it provides the framework and the prototype called Object-TRAKLA.

1.2 Electronic exercise book

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 content expert\textsuperscript{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. Second, the overall client-server architecture between TRAKLA and its graphical user interface Tred is based on the assumption that a student has active on-line access to the system during the exercise session. This assumption might lead to inequality because not all the students have free connection to the internet. However, some of them do have, thus putting these students in a privileged situation.

In the electronic exercise book metaphor the overall goal is to provide a system in which the set of exercises could be easily extended by the teacher and all the exercises could be solved off-line. Only the submission of the exercises should require on-line access. This would minimize the on-line costs, thus putting all students in an equivalent situation. Moreover, the automatic assessment raises the question in which circumstances it is safe to perform the assessment procedure automatically and how to avoid the possible pitfalls. Finally, the rapid development of the Java programming environment has driven us into the situation in which Tred, the graphical user interface of TRAKLA, has become obsolete. These, among other things, are the main reasons why we have decided to start developing a brand new system.

1.2.1 Goal

Our goal is to develop a framework for an electronic exercise book in which the key topics of data structures and algorithms could be learned in an environment that is capable of supporting user controlled simulation of algorithms as exercises. The electronic exercise book should be capable of automatic assessing of exercises and giving immediate feedback on student’s performance. The further development of the system should require only programming skills, but not any technical skills of a content expert, to produce a new exercise. The framework should also provide a meaningful starting point for developing similar environments for other related topics.

The challenge is to determine a visual representation for all possible abstract data structures, in order to manage the algorithm animation. It should be noted, however, that we are only developing a prototype and therefore we possibly omit, 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 have to 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

\textsuperscript{4}By a content expert we mean a person who is familiar with all technical details of the overall system and the framework.
whole design and architecture of the system. Thus, components created during the
development of the environment should be reusable.

Since we are dealing with a highly complex system, it is adequate to use formal
methods [15, 31] 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 [1, 11, 30, 37, 43]. However, most of the terms
and notions are more-or-less adopted from this 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 [14, 16]. In particular, the Java programming
language [12] has been selected as the primary implementation language because of
its many good properties.

1.3 Contributions

The main contributions are the following

1. The theoretical simulation model for data structures and algorithms consisting
   of well-defined mathematical models for some elementary data types together
   with a new concept of abstraction called fundamental data type.

2. Tools to support algorithm visualization, animation and simulation for a set of
   ready-made exercise algorithms and data structures together with functionality
   of automatic assessment and immediate feedback on student’s performance.

3. Framework for supporting visualization, animation and simulation of user-
   made algorithms.

1.4 Organization of this Thesis

This thesis consists of 7 chapters which are organized as follows. In Chapter 1, we
have given the motivation for this thesis and a brief overview of the research topic
that is discussed further in Chapter 2. In Chapter 3, we create the theoretical con-
struction, which forms the basis of the rest of this thesis. In Chapters 4 and 5 we
describe the object-oriented application framework, called Object-TRAKLA, imple-
mented as a part of this thesis. In Chapter 6, we give a brief overview of how to use
the framework for educational purposes. Finally, in Chapter 7, we give the summary
and conclusions.
Chapter 2

Previous Work

In this chapter, we will describe the world of algorithm animation and simulation by first defining these terms in the context of education. We also summarize the achievements of the field by analyzing some example systems in terms of the taxonomy frequently used in journal papers.

2.1 Terminology

We start by discussing some basic terms introduced in the title of this thesis. Although clearly all interested readers have a picture of what those words mean, we would like to offer a little deeper power of looking into the field, and understanding clearly what we mean by these words in this thesis.

2.1.1 Learning Environment

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.

In this thesis we are going to describe how to apply both of these methods 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 as students.
2.1.2 Simulation

There are two different styles of simulation by Kreutzer [28]. On the one hand, there is its classical application as a vehicle for numerical predictions. 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 any 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.

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

“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.” [28]

In this respect we would like to give the formal representation for all data structures as done in the next chapter. The challenge is to construct a simulation tool based on this theoretical framework. A learner could use this simulation tool for “exploration of states of behavior” of algorithms and data structures.

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.” In this way we can see the construction of simulation tools as also justified from the researcher point of view. A different facet of the simulation method is rooted in the fact that it is an experimental technique, comprising aspects like model calibration and data collection as well as experimental design and output analysis.

Managing complexity is at the heart of simulation programming and has guided the development of its tools. Graphical symbol systems also have a role to play. This repertoire of specialized 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.

Advantages of graphical representations are that structural connectivity and symmetry are emphasized. Things that are conceptually related appear to be close. They also provide a rich syntax to visually define such concepts as links, flows, 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.
Representation will always be needed for simulation, since it is by definition an experimental technique where data generated by each experiment has to be collected, summarized and reported in 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.

2.1.3 Software Visualization

Software visualization (SV) 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 and low-level views that are not expressive enough to convey adequately how an algorithm functions. Thus, from the pedagogical point of view, we consider this to be a more trivial task to achieve.

In the second domain, called algorithm visualization (AV), we are interested in visualizing all the necessary states of the data structures during the execution of an algorithm. In other words, we want to explicitly show the object structures (components, subcomponents, and their values) which are required to fully understand 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 have any additional information about the behavior. However, the visualization as the discrete sequence of views of data structures should include enough frames to fully 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.

Algorithm animation is defined as the combination of the trace information together with the visualization of all the necessary states of the abstract data types needed by the algorithm.

User Controlled Simulation

If human interaction is allowed between the visual demonstration of an algorithm and the user in such a way that the user can manipulate the data structure representation, we are talking about user controlled simulation of an algorithm (or simply algorithm simulation). The simulation is conceptually the opposite to the algorithm animation with respect to the information flow. Where algorithm animation delivers the visualizations from the system to the user, algorithm simulation delivers the input from the user to the system, possibly through some graphical user interface (See Figure 1.1 in page 1).

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. Thus, user controlled simulation of an algorithm is defined as a method that requires human interaction for monitoring and executing the simulation model, e.g., a sequence of visualized data structures.

In this thesis, we are only interested in visualizing data structures; and the visualization of the trace information (program visualization) is not discussed any further.

2.1.4 Automatic Assessment and Feedback

Today, the primary use of algorithm animation has been for teaching and instruction. However, from the pedagogical point of view, feedback on student’s performance is needed in order to see the system as a learning environment. Fortunately, the formal nature of algorithms and data structures gives a way 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.

2.2 Taxonomy

A well-founded taxonomy is needed to facilitate communication about ideas and discoveries. Taxonomies provide the common terminology and allow new discoveries to be identified and catalogued.

The study of the example systems in the next section is based on the principled taxonomy for the systems involved in the visualization of computer software gathered by Price, Baeker, and Small [34], the review article written by Boroni et al. [8], and other discoveries identified and reported by several other authors developing SV systems. In addition, we have included a few subcategories to cover also the needs of the electronic exercise books. These modifications and other subcategories are discussed thoroughly in Section 2.4.

2.2.1 Categories

The evaluation of the example systems is divided into the following six top-level categories for SV systems as suggested by Price et al. [34]:

1. **Scope** describes the range of software that can be handled. This category sets
the fundamental limitations and restrictions the system has. We see two major divisions of Scope: generality and scalability. The generality is related to the platform-independency and the scalability describes to what degree does the system scale up to handle large examples.

2. **Content** describes the subset of Scope that the system really uses in constructing the visualization. Two opposite domains can be identified. The differentiation between these domains is subtle and can best be described from a student perspective: in one domain the system is designed to educate the user about a general algorithm, thus it falls into the domain of **concept animation**. In the second domain the system is teaching the user about one particular implementation of an algorithm, thus it is more likely **program animation**. If the decision is not clear, we can always use the common term **algorithm animation**.

3. **Form** characterizes the output of the visualization from the user point of view. This category describes how the fundamental characteristics of the system are directly related to what can be displayed. Many subcategories exist such as **presentation style**, by which we mean the general appearance of the visualization, and the **granularity** or “coarse-granularity details” of the software.

4. **Method** characterizes the elements of the visualization specification. This area describes the fundamental features of the SV system which the visualizer uses to create a visualization. The taxonomy [34] divides this into two areas, one describing the **style** in which the visualizer specifies the visualization and one describing the way in which the visualization and the program source code are connected.

5. **Interaction** characterizes the interactivity of the system. Subcategories include **style**, **navigation** and **scripting facilities**. Style describes the methods the user employs to give instructions to the system. Navigation characterizes to what degree does the system support navigation through a visualization. Finally, scripting facilities allow the system to provide facilities for managing the recording and playing back of interactions with particular visualization.

6. **Effectiveness** describes the empirical evidence of the systems effectiveness. As a highly subjective measure this category may be made of many factors. We have included five subcategories that are **purpose, clarity, evaluation, production use**, and the level of **automatic assessment**.

### 2.2.2 Roles

The user may act in many different roles. Thus, the parties involved require definitions within the scope of this thesis. These definitions are adopted from the used taxonomy [34]. Particularly, in the case of an educational system, the user is usually one of the following: a student, an instructor, an assistant or a content expert.
1. *programmer* – person who wrote the original program that is being visualized (usually a student, in some cases any user)

2. *SV software developer* – person who wrote the system used to create visualizations (content expert)

3. *visualizer* – person who created the visualization from the original program using the SV system (usually a student)

4. *user* – person using the visualization to understand the original program (usually a student).

### 2.2.3 Symbols

Finally, here is the explanation of symbols used in tables:

1. **i** – lowest or no
2. **ii** – below average
3. **iii** – average
4. **iii** – above average
5. **** – highest or yes

### 2.3 Example systems

In this chapter, we describe the main observations we have made in the literature survey done during the past few years. A wide variety of systems has been developed for creating algorithm animations for educational purposes. We will we give a very short description of six different SV systems developed in university environments. More detailed descriptions of some of the key features of these systems are provided in Section 2.4. These systems do not fully span the space of used taxonomy, but they serve as concrete reference points for mapping the taxonomy into familiar examples.

### Rasala

One of the easiest yet rarely used methods of automatic SV is the usage of smart components. When algorithms are animated, they rarely make use of the whole vocabulary of reserved words available in programming language. Thus, based on C++ templates and operator overloading it is possible to develop automatic array algorithm animations [35] that are even entirely free of any code related to animation.
Jeliot

Jeliot [18] is a web based proof-of-concept environment that is capable of animating algorithms (programs) written in the Java programming language by visualizing data structures as smoothly moving graphical objects. Jeliot supports multiple synchronized self-animating views for predefined visualizations and program animation capabilities. The Web system automatically adds necessary animation calls in the source code.

Sangwan

Sangwan’s prototype [27, 29] supports multiple synchronized self-animating views for integers and types derived from them by using pointers, arrays, and structs. The system enables visualization of programs requiring only simple syntactic changes to standard C/C++ code.

Samba

Samba [40] is an application program (X windows and Motif) of the Polka [41] algorithm animation system. An algorithm implemented in any language can be animated because the system annotates a program with interesting events that indicate program states of interest. Animation scenes are then triggered corresponding to these events.

JDSL Visualizer

The JDSL Visualizer [6] is a set of tools for visualizations of user-written data structures in the Java programming language, and testers which check the functionality of these data structures. The animation is achieved by providing application interfaces (API) for smart components and thus does not require any modification to the student’s code. The testers provide accurate reports on the functionality of student’s programs by interacting directly with the user-written data structure through the API and by comparing the behavior of the student’s code to the reference implementation.

TRAKLA

TRAKLA [20, 24, 25, 26] system is a Web-based, computer-aided learning environment for teaching algorithms and data structures as described in the previous

1The service is available at http://www.cs.helsinki.fi/research/aaps/Jeliot/.
chapter. The client-server architecture has two separate systems that work together. The server distributes tracing exercises to the students and then evaluates their answers to the exercises. The client portrays a variety of algorithms. The students turn in their exercises by using the interactive graphical client that is coded in the Java programming language.

Other systems

The complete evaluation of the example systems is limited only to the systems described above. However, in the following sections we have also referred to some other systems in which some special feature or characteristics is well illustrated. These systems include the pioneering visualizers BALSA [9], Tango [39], and Zeus [10]; the Java version of Samba [40], called Lambada [3, 4]; “Algorithms in Action” tool reported by Stern, Søndergaard, and Naish [42]; and the WWW environment for online marking, described by Mason and Woit [32]. Note that the pioneering visualizers above are evaluated by Price et al. [34] and therefore that evaluation is not repeated here.

2.4 Evaluation

The following rankings of the example systems are derived from our understanding of the systems based on the papers written about them and in some cases by our own use of the system or by consultation with its author.

2.4.1 Scope

Only a few systems are capable of visualizing a completely unrestricted set of programs. This first category sets the basic limits of the SV systems. On the other hand, a generalized system can generate visualizations of arbitrary programs within a particular programming language and system. However, it usually has some restrictions with regard to its capabilities on platform (hardware, operating system, windowing system, etc.), programming language, concurrency, applications (what kinds of user programs can be visualized), and speciality (what kinds of programs is it good at visualizing).

The original paper, introducing the category Scope [34], is written in 1993, thus rating the generality of most of the systems as highest possible. The Java programming language, however, introduces a new insight into the field of generality by providing a way to produce platform-independent software. Moreover, TRAKLA gives a meaningful way to provide language independent visualizations because the algorithm animation is produced by the user itself in terms of user controlled simulation. Therefore, a system which is ranked high in generality before should be
reassessed as medium today as technology has improved markedly. On the other hand, if only user controlled simulation is possible to produce animation, this raises the question how to assess the Code Scalability? This is why we have decided not to estimate that subcategory at all in the case of TRAKLA.

*Scalability* is also one of the fundamental categories of Scope. It can be divided into two separate domains: program (Code) and data sets (Data). These characteristics refer only to the fundamental limitations of the system. See category Effectiveness on page 20 to determine how well it presents visualizations of large programs.

In Table 2.1 we show the relative rankings for our example systems against the Scope subcategories.

<table>
<thead>
<tr>
<th>System</th>
<th>Generality</th>
<th>Platform</th>
<th>Language</th>
<th>Applications &amp; Speciality</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basala</td>
<td>??</td>
<td>PC,Mac</td>
<td>C++</td>
<td>sorting, searching</td>
<td></td>
</tr>
<tr>
<td>Jeliot</td>
<td>??</td>
<td>any</td>
<td>Java</td>
<td>basic data types</td>
<td></td>
</tr>
<tr>
<td>Sangwan</td>
<td>??</td>
<td>SunOS</td>
<td>C++</td>
<td>basic data types</td>
<td></td>
</tr>
<tr>
<td>Samba</td>
<td>??</td>
<td>X-windows any</td>
<td>any</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDSL Visualizer</td>
<td>??</td>
<td>any</td>
<td>Java</td>
<td>basic data types</td>
<td></td>
</tr>
<tr>
<td>TRAKLA</td>
<td>??</td>
<td>any</td>
<td>any</td>
<td>basic data types</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.1: Evaluation of the systems against category Scope

Note, however, that in order to gain the highest possible scalability (code and data), we expect the system to be capable of handling very large systems as required in software engineering. Most of the systems intended for educational purposes do not meet these requirements. In addition, highest possible generality should be assessed only for platform-independent systems which are capable of visualizing a completely unrestricted set of programs.

### 2.4.2 Content

Most of the visualizers can be separated into three groups in relation to their use for teaching levels of abstractions of data structures as described by Boroni *et al.* [8]. The first group provides a high-level, conceptual view (*concept animation*) and animations of some computer science topic. The goal is to provide interactive animations for the fundamental concepts, thus allowing users to interact with the system at runtime. Typically, one limitation is that the systems only allow access to a limited subset of the potential functionality of an object structure. Moreover, such
<table>
<thead>
<tr>
<th>System</th>
<th>Animation type</th>
<th>Visualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rasala</td>
<td>Algorithm Animation</td>
<td>I</td>
</tr>
<tr>
<td>Jeliot</td>
<td>Self-Animating Programs</td>
<td>III</td>
</tr>
<tr>
<td>Sangwan</td>
<td>Self-Animating Programs</td>
<td>III</td>
</tr>
<tr>
<td>Samba</td>
<td>Concept Animation</td>
<td>I</td>
</tr>
<tr>
<td>JDSL Visualizer</td>
<td>Concept Animation</td>
<td>II</td>
</tr>
<tr>
<td>TRAKLA</td>
<td>Concept Animation</td>
<td>I</td>
</tr>
</tbody>
</table>

Table 2.2: Evaluation of the systems against category Content

animations tend to be laborious to develop further. The second group, pure algorithm animation, refers to the visual, graphical representation of an algorithm with limited interaction. The user controls the animation process, but it is typically limited to allow control over the operational interface of the animator, instead of allowing the user to take full control over the functional interface of the animated algorithm. The third group (program animation) allows the user considerable interactivity with their own data structures as well as a display of the different data structures in memory, by integrating visualization with a source-level debugger. Nevertheless, they accomplish this task by displaying the physical organization of the data in memory rather than the underlying structure it represents.

Integration of program animation and concept animation allows students to create programs which are self-animating [18, 29]. Unlike visual debuggers (pure program animation) such systems display the dynamics of a program and visualize the context in which changes to data structures occur. The challenge seems to be to develop systems in which the programmer’s manual intervention is limited to minimum. As a matter of fact, there exist even systems in which “artificial intelligence” is able to recognize the high-level data structures and display proper abstractions for them [19].

As an example of concept animation, the JDSL Visualizer displays a set of fundamental data structures at the conceptual level instead of simply visualizing the contents of memory, thus allowing users to interact at runtime with all the methods of a data structure to verify its operation. Moreover, TRAKLA represents fundamental data structures allowing the student to simulate manually these methods in a lower level (for example, by simulating delete operation of binary search tree by manipulating the visualization of fundamental data structure binary tree.). However, there is also an additional advantage. The simulation is independent of the actual programming language, thus focusing on the logic, but not on the implementation of an algorithm.
Table 2.2 summarizes the relative rankings of the example systems. Animation type refers to the main purpose of the overall system as discussed above. Code and Data columns illustrate to what degree does the system visualize the instructions in the program source code and data structures, respectively. The timeline illustrates the data gathering time and visualization generation time (batch job or live).

Systems focusing on concept animation have usually no code animation included at all. The JDSL Visualizer is an exception even though it does not show the actual code. However, it does represent those methods an application interface declares and the corresponding visualized class therefore implements. Average program animation includes source code window with currently executing line highlighted. More advanced systems should be capable of real debugging of source code such as making program stop at certain points (breakpoints), examining stack, and several levels of stepping program (one instruction, function, etc.). Thus, more advanced program animation systems are more likely to be debugging tools and environments for software engineering.

2.4.3 Form

In this category we discuss the fundamental characteristics of the system, which are directly related to what can be displayed. The subcategories include:

1. presentation styles such as colors, dimensions, animation, basic blocks and components, and sound;
2. granularity and elision that defines the degree of coarse-granularity details;
3. multiple views that characterize the possibility to provide multiple synchronized views of different parts of the software being visualized; and
4. program synchronization that characterizes the possibility to generate synchronized visualizations of multiple programs simultaneously.

Even though our example systems do not include any other examples of understanding the “visualization” than those of which deal with visible images, it should be noted that there could also be other considerations. Therefore, the root word “visual” should be understood as a “mental picture” of something that possibly is not even actually present to the sight but is resulted from input from other human senses. Zeus [10] is an example system in which sound is used in algorithm animation to produce multi-modal visualizations in this sense.

Table 2.3 summarizes the relative rankings of the example systems against category Form.

There is no example system whose granularity nor elision is estimated to be the highest possible. This is because they lack of the ability to show the big picture of the
<table>
<thead>
<tr>
<th>System</th>
<th>Presentation Style</th>
<th>Granularity &amp; Elision</th>
<th>Multiple Views</th>
<th>Program Synch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rasala</td>
<td>chart, point, line</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeliot</td>
<td>data type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sangwan</td>
<td>data type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Samba</td>
<td>line, shape, position, color</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDSL Visualizer</td>
<td>data structure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAKLA</td>
<td>basic data type</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: Evaluation of the systems against category Form

underlying concept; or they have no way to hide unnecessary details as required. In addition, only Samba is capable of fully supporting multiple views of one particular data object structure. Moreover, it is the only system which is capable of illustrating the program synchronization of the concurrent runs of different algorithms.

2.4.4 Method

New methods and techniques are increasingly being developed and used to improve the teaching aids of data structures and algorithms. All of these techniques require some degree of manual intervention by the programmer. Two methods are heavily used:

1. insertion of trace calls to signal events of interest to an auxiliary animation system (instrumentation), and

2. explicit use of specialized data types or operations in the code of the algorithm (probes).

The real challenge seems to be to minimize the manual intervention. One possible way to do that is to increase the level of automation. For example Jeliot provides a parser to annotate the original source code. On the other hand, Rasala [35] describes how the C++ programming language specific techniques could be used for hiding intervention details from the programmer by using templates and self-animating arrays. This approach requires some kind of library of smart components that implements the templates and produces the animation.

Another approach for hiding the connection technique is the intervention style described by Astrachan et al. [3, 4] in which the Java version of Samba, called Lambada, is created. The animations do not require extensive programming by students since
they use classes and code that the system provides. Most of the details used for producing the animation process can be hidden together with the complexity of the system. Originally, the system was dependent on tracing calls just like Samba.

Furthermore, we have the TRAKLA exercises in which the animation is constructed by user controlled simulation of the algorithm. Thus we call the specification style *emulation*. In addition, there is no connection technique involved since there is no actual implemented algorithm to produce the animation.

<table>
<thead>
<tr>
<th>System</th>
<th>Connection Technique &amp; Specification Style</th>
<th>Tailorability</th>
<th>Code Ignorance Allowance</th>
<th>System-Code Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rasala</td>
<td>automatic, templates</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeliot</td>
<td>automatic parser annotation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sangwan</td>
<td>library, probes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Samba</td>
<td>manual instrumentation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDSL Visualizer</td>
<td>manual, interfaces</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAKLA</td>
<td>emulation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: Evaluation of the systems against category Method

The *connection techniques and specification styles* described above have their pros and cons depending on the educational purposes from the pedagogical point of view. For example, the group of the pioneering visualizers such as BALSA [9], Tango [39], Polka [41], and Polka’s application (front-end) Samba [40] 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 [40]. On the other hand, a system without any intervention requirements makes it possible to the programmer to create limited algorithm animations without any prior knowledge of the actual animating system. For example, Rasala [35] describes an approach to array algorithm animation that requires no intervention into the algorithm itself. The automatic animation helps the student understanding what the programmed algorithms are doing and in debugging errors. From the research point of view, it is possible to animate, for example, some low level utilities through application program interfaces (API), thus learning how they are function without even looking into the source code.

Many of the systems also provide a way to customize the visualization by interactive manipulation on the part of the user. We refer to this feature as *tailorability* which characterizes to what degree can the user customize the visualization. Examples of customizations are window resizing, scrolling and zooming, layout changing, etc.

Other subcategories related to intervention technique include:
1. High Code Ignorance Allowance (CIA) allows production of visualization without any code knowledge on the part of the visualizer.

2. High System-Code Coupling (SCC) indicates that the SV system is very closely tied to the program it is visualizing.

None of the systems qualifies for the highest level of tailorability. This is because none of them allow representations of data to be changed live by direct manipulation with the corresponding data in the program and other views changed automatically as could be possible for example in Zeus [10].

2.4.5 Interaction

The level of interaction between the user and the system could be divided at least into the following five separate groups:

1. No other control than the animation playback,
2. visual debugging abilities,
3. step by step control over the sequence of the snapshots of the visualized structure backward (backward debugging) and forward,
4. interface methods and operations, and
5. the user controlled simulation of fundamental data structures.

Visual debuggers and visual debugging usually provide the very same set of functionality than traditional debuggers with the exception that some portions of the code is also illustrated graphically. This concept is then possibly developed further by including, for example, the backward code traversal ability.

Algorithm simulation gives another approach in which we are not so interested in the actual algorithm or program code, but its logic and behavior in terms of changes in the data structures. The systems provide sophisticated visualizations for data structures and the user could manipulate the structures by invoking some predefined operations. This kind of user controlled simulation could be driven by some user defined algorithms and methods, or the user may directly interact with the visualization in order to simulate the operations to be performed step by step.

The subcategories of the category Interaction could be characterized as follows:

1. Style – the method(s) the user employs to give instructions to the system.
2. Navigation – the degree of supported navigation through visualization.
3. *Elision Control* – the degree of supported user controlled suppression of details from the display.

4. *Temporal Control* – the temporal aspects of the execution of the program (direction, speed, etc.).

5. *Scripting* – the degree of supported facilities for managing the recording and playing back the interactions of particular visualizations.

<table>
<thead>
<tr>
<th>System</th>
<th>Style</th>
<th>Navigation</th>
<th>Elision Control</th>
<th>Temporal Control</th>
<th>Scripting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rasala</td>
<td>command line</td>
<td></td>
<td></td>
<td>speed</td>
<td></td>
</tr>
<tr>
<td>Jeliot</td>
<td>buttons, menus</td>
<td>III</td>
<td>III</td>
<td>speed</td>
<td>IIIII</td>
</tr>
<tr>
<td>Sangwan</td>
<td>buttons, menus</td>
<td>III</td>
<td>III</td>
<td></td>
<td>IIIIII</td>
</tr>
<tr>
<td>Samba</td>
<td>buttons</td>
<td>II</td>
<td></td>
<td></td>
<td>IIIII</td>
</tr>
<tr>
<td>JDSL Visualizer</td>
<td>buttons, menus, direct manipulation</td>
<td>IIIII</td>
<td>I</td>
<td>speed, direction</td>
<td>IIII</td>
</tr>
<tr>
<td>TRAKLA</td>
<td>buttons, menus, user controlled simulation</td>
<td>IIIII</td>
<td>I</td>
<td>direction</td>
<td>IIII</td>
</tr>
</tbody>
</table>

Table 2.5: Evaluation of the systems against category Interaction

None of the systems qualify for the highest possible ranking in navigation since for example the system described by Stern *et al.* [42] shows the very sophisticated strategy for managing content complexity in algorithm animation by providing a tool to support multiple level of detail and the facility for the user to control the level of detail being viewed. Also the elision control could be much more evolved than it is in systems described here. One possible reason for little or no elision control might be rooted in the fact that all of the systems are intended for educational purposes and the power of elision control cannot be utilized until very large examples.

### 2.4.6 Effectiveness

The effectiveness of educational SV systems could be characterized by the question *how well* does the system communicate information to the student? This is highly subjective measure and can be made of many factors. The taxonomy suggests four subcategories which are

1. *Purpose* – the intended target group and goal of the system, for example,
   
   (a) Novice/Expert (N/E),
(b) Classroom Demonstration (CD),
(c) Algorithm Development (AD),
(d) Debugging (D), or
(e) Educational Exercises (E).

2. Appropriateness and *clarity* – the degree of the visual metaphors inspiring understanding.

3. Empirical *evaluation* – the degree of experimental evaluation the system has been subjected to.

4. *Production* use – the degree of production use.

However, educational systems could also be evaluated against some other categories. We have selected one important additional category: the level of *automatic assessment* and grading. Unlike visualizers, this topic has received little attention in the computer science education literature. However, by automatically testing students’ programs we may provide accurate reports on the functionality of their code, which in turn increase grading efficiency.

One fundamental difference between the JDSL Visualizer and TRAKLA is that the first one is designed to *programming class* to provide interactive debugging tools for educational purposes while TRAKLA is intended for *data structures and algorithms class* to illustrate and grasp the logic and concepts of data structures and algorithms.

Many of these tools mentioned here are intended for some specific computer science class. Naturally, this has also effects on what kind of properties are supported. It seems to be fact that the field of algorithm animation has been divided toward two domains. In one domain we are interested in visualize ones own algorithms in order to learn if they are functioning correctly. These are typically coded in some predefined programming language. These types of animations refer to debugging of an algorithm by tracing the execution of the algorithm instructions. In the second domain understanding some predefined algorithm and its principles and logic is essential. No actual programming language is required, but the examples are rather on some pseudo-language level.

Finally, while discussing teaching and learning aids, it is natural to establish control over the process in terms of feedback. Usually, two kinds of feedback is required. First, automatic systems should provide teacher with feedback on how the students are performing. This is naturally integrated to the automated administration tools for course operation, if provided. Second, the student needs feedback himself in order to keep up the learning process. There are several ways to provide this kind of feedback. Traditionally, assignments are evaluated manually by the teacher. However, learning environments are capable of several meaningful ways to improve the feedback and marking process by allowing, for example, on-line marking functionality to provide in-context annotations of work returned to students [32]. These systems usually have no software visualization involved, thus they are not discussed
here any further. On the other hand, more advanced systems are also capable of
testing of student-written code [6, 7] or even automatically evaluate assignments
returned in some predefined format [20]. We summarize the methods as follows:

1. manual evaluation of students programs/assignments,
2. on-line evaluation of students programs/assignments,
3. program testers designed to automate the grading of students programs (AEP), and
4. automatic assessment of students assignments (AAA).

For example, the JDSL Visualizer includes a set of program testers (AEP) to com-
pare the student’s implementation to the correct model solution and judge the cor-
rectness of the student’s implementation. On the other hand, TRAKLA is designed
to distribute tracing exercises to the students and to evaluate their answers to the
exercises (AAA).

In both of these last two cases, the nature of the grading process is dynamic. The
student turns in an exercise which correctness is judged by comparing it to some pre-
defined model solution. In the first approach several tests (a number of experiments
and corresponding model solutions) are used to gather evidence of the correctness of
the solution in order to be sure it has no errors. In the latter case, every student has
its own individual assignment and a deterministic algorithm is used for computing
the correct model solution for each student (instance of the assignment).

<table>
<thead>
<tr>
<th>System</th>
<th>Purpose</th>
<th>Clarity</th>
<th>Evaluation</th>
<th>Production Use</th>
<th>Automatic assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rasala</td>
<td>N+AD+D</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>manual</td>
</tr>
<tr>
<td>Jeliot</td>
<td>N/E+AD+D</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>manual</td>
</tr>
<tr>
<td>Sangwan</td>
<td>N/E+AD+D</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>manual</td>
</tr>
<tr>
<td>Samba</td>
<td>N+CD+E</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>manual</td>
</tr>
<tr>
<td>JDSL Visualizer</td>
<td>N+CD+D</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>AEP</td>
</tr>
<tr>
<td>TRAKLA</td>
<td>N+E</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>AAA</td>
</tr>
</tbody>
</table>

Table 2.6: Evaluation of the systems against category Effectiveness. N/E = for
Novice/Expert (or N = for Novice only); CD = Classroom demonstration; AD =
Algorithm Development; D = Debugging; E = Educational Exercises.
2.5 Conclusion

As we can see, there are efforts to develop systems in which the programmer’s manual intervention to produce an animation is limited to minimum. For example, the Astrachan group [3, 4] are developing the “interesting event approach” based on pioneering systems BALSA [9], Tango [39], Polka [41], and Samba [40]. Another characterizing property is whether a system is intended to be used for program animation as a visual debugger or to concept animation in order to grasp the logic of visualized concept. The Rasala’s Array Animation prototype [35] completely hides the intervention in the algorithm itself. However, the current prototype neither has visual debugging capabilities nor any concept animation components.

The other issue is the level of interaction. The program animation naturally requires “debugging functionality” [18, 29]. On the other hand, the concept animation seems to provide more sophisticated methods to interact with the system. The student may change the behavior of the system by programming some new methods [6] or by simulating some sequence of primitive operations [20, 24]. In roughly speaking, we could say that in the program animation we are interested in algorithms by focusing on the data structure visualization, and in the 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 to become tools for improving the quality and functionality of computer programs. On this kind of interactive process the feedback is essential and thus the quality of the feedback is becoming more and more important.

However, most of the systems do not give any guidance how to give or to obtain some 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 [6, 7, 20]. Most of these systems are intended for testing whether the students’ program is producing correct outputs for a given input or whether some errors were encountered [6, 7]. In this respect, TRAKLA [20] represents a unique genre by providing the automatic assessment of assignments.

Even though the feedback to the 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 a very important issue in order to improve the teaching process. Another concern is whether the student is actually doing something useful with these tools or is he just playing and enjoying, for example, the animation without any actual learning. As it is reported [6], 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 also a variety of weekly homework assignments are widely used. Thus, automatic assessment of these
kinds of assignments would be of great value to large-scale courses.

### 2.5.1 The Goal revisited

We are now ready to define the goal described in Section 1.2.1 in terms of the taxonomy. First, the new version of TRAKLA, called Object-TRAKLA, should not suffer from those problems described in the previous section. Second, many new features should be included, as we summarize here.

The most crucial part of the design of the new SV system is the category Method in which the specification style and the chosen connection technique both have a very important role to play. As we have demonstrated in the previous section, all the methods have their pros and cons depending on the teaching and learning style. Thus, we have decided to provide several different methods to cooperate, including

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

In the case of user controlled simulation the user can emulate an algorithm, thus the user specifies the animation in terms of algorithm simulation. User-coded algorithms refer to algorithm visualization and concept animation. Moreover, user-coded programs should also be possible to produce 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.

The Scope and Content of the new system should provide a generalized platform-independent framework that is capable of visualizing even large examples. The Form 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 applications. Nevertheless, the category Interaction cannot be evaluated until some applications are really implemented. Finally, the Effectiveness of the new framework is beyond the scope of this thesis due to its subjective nature.

**Note to the Reader**

In this thesis, we illustrate the design and architecture of the implemented framework by providing several different kinds of representations. In Chapter 3 we characterize the framework by using formal methods. At least the software developer should be
aware of the underlying formal definitions and semantics. This kind of content expert
should also be familiar with the practical programming details that are described in
Chapter 4. Although it is not necessary, it is recommended that the programmers
and visualizers read that chapter, as well. This is because the connection technique
Method is described there in detail. Moreover, in Chapter 5 we describe the technical
details needed to visualize, animate, and simulate an algorithm. There we also
describe the specification style Methods the framework allows. This functionality is
based on the results obtained in Chapters 4 and ch:Technical.

A limited application and the user interface was developed during the project and
the results are reported at the end of the thesis. The Form and included presentation
styles may probably change in the future, but this prototype should be rich enough
to illustrate the capabilities of the system. In Chapter 6 we briefly discuss the
educational aspects and the purpose of the system from the teacher and the student
points of view. Finally, in Chapter 7 we will draw the conclusions and describe
the contributions by discussing the features of the framework and the application
front-end in terms of the taxonomy in detail.
Chapter 3

Theoretical Point Of View

In this chapter, we are going to give descriptions for all the necessary terms and notions in order to fully understand the design and architecture of the implemented system. We use several different kinds of techniques to illustrate our point of view. From the theoretical point of view, it is natural to formally define those concepts and examples used in this thesis. On the other hand, from the technical point of view, we have decided to lean on object-oriented modeling, and therefore it is natural to use real engineering tools, diagrams and methods to illustrate those very same examples in the context of how this theory is applied in practise. 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 [1, 11, 43]. 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 [15, 31].

On the other hand, this chapter also defines a completely new generalization called fundamental data type. As far as I can see, this concept does not refer, despite the name, to any concept known so far.

The constructed model is used as a framework for the design of binary search trees, as described in the case study at the end of this chapter.

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, notations while handling sets [15]. 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 \( x R y \) instead of \( (x, y) \in R \). Additionally, 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. \( \forall x, y \in A, (x, y) \in R \lor (y, x) \in R \) (comparability).

An ordered set is an ordered pair \((A, R)\) in which \( R \) is an order on \( A \). Additionally, 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 additionally \( R_1 \) be the transitive extension of \( R \) such that \( R_1 \) contains \( R \), and moreover, if \( \{(a, b), (b, c)\} \in R \), 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 function. The function \( F \) could 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 $\forall(a,b_1), (a,b_2) \in F \rightarrow b_1 = b_2 = F(a)$. Additionally, if $\forall(a_1,b), (a_2,b) \in F \rightarrow a_1 = a_2$ 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 underlying theory. Most of these terms and notions are used further in the text in case we wanted 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 undirected graph each edge in $E$ is an unordered pair of vertices.

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

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

A path is said to be simple if all 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 could 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. $I$ is the set of input values,
2. $O$ is the set of output values and
3. for all instances of $I$ algorithm terminates with $O = f(I)$.

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 $f : I \to O$ efficiently, the set of input values $I$ should be well organized. We need some ordering and structure for the elements. In general, we refer to such a structure as a data type.

3.3 Data Types

Sets are as fundamental to computer science as they are to mathematics [1, 11]. Whereas mathematical sets are unchanging, the sets manipulated by algorithms may change over time, so they are dynamic.

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 there always exist one or more attributes, which form an 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.

In computer science dynamic sets are special cases of data types. Data type is a particularly important concept because it determines the input and output sets for an algorithm.

**Definition 3.3.1** 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.

We refer to the instances of data types simply as variables. Furthermore, primitive data types are built-in to a programming language. In the Java [12] programming
language they are boolean, char, byte, short, int, long, float, and double. For example, a variable \( b \) of type boolean is defined as Boolean \( b \in \{\text{true}, \text{false}\} \). Thus, it can assume either the value true or the value false, but no other values.

The union of all possible data types are called the generic data type. An instance of such a generic data type is called generic type variable.

### 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 similar manner.

**Definition 3.3.2** An abstract data type (ADT) is a mathematical model with a collection of operations defined on that model.

The expression mathematical model is essential here. By the definition, it is impossible to define abstract data types without clear and explicit data models. However, the term ADT is a little misused in the literature. This is why we have reserved the word interface here to mean an abstract data type without an explicit mathematical model. So, an interface is a skeleton of an ADT without any kind of hint of the actual design. Moreover, there should be a way to represent the model in terms of mathematics. This is where dynamic sets comes to the picture, as they are generalizations of mathematical sets.

There exist several methods to define the mathematical model of an ADT. Maybe the most natural way is to use some actual programming language and its syntax. For example, the Java programming language has its own way to declare not only interface types but also abstract classes, which are partially design and partially implementation. On the other hand, there are also formal methods to use, which we would like to call as elements of discrete mathematics. The idea of using formal methods is rooted in the fact that formally defined abstract data types should include only the design and no implementation. This is an important matter if one wants to focus on important design issues while ignoring unnecessary implementation details. However, in this thesis, we are going to use all of these methods to illustrate our examples.
3.3.2 Fundamental Data Types

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

Definition 3.3.3 Data Model is the collection of data types connected with (binary) relations in order to serve as the mathematical model for an abstract data type.

Moreover, fundamental data types 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 fundamental data types are:

1. array,
2. linked list,
3. binary tree,
4. tree, and
5. graph.

For example, array is 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, array can also refer to the fundamental data type composed of a (homogeneous) collection of variables, each variable identified by a particular index number. From this point of view, it is possible to implement arrays in many different ways. Most of the programming languages do not support this latter form of definition, thus making the term somewhat ambiguous. Nevertheless, in this thesis, we use the latter form.

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

Example 3.3.1 Let us consider the abstract data type AVL tree. Obviously, a natural choice for the data model to this particular search tree is the binary search tree. This abstraction could be defined formally. On the other hand, this data model assumes that the keys are drawn from some totally ordered set of keys, and thus causes some type constraints for keys. Hence, a better choice for the (fundamental) data model could be just an ordinary binary tree.
Fundamental data types do not have any type constraints. 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 object-oriented programming paradigm this kind of type could be seen as an empty interface type. Second, an FDT could store an element of any type. Although this requirement is essential it is not sufficient. Let us consider the abstract data type stack, which do not necessarily have any type constraints for elements to be stored. We do not want to see the stack as a fundamental data type in the 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 it has its chronological partial ordering of elements stored into it (LIFO).

We are now ready to define the concept of fundamental data type as follows.

**Definition 3.3.4** Fundamental data type is the static part of the data model in which all data types are generic.

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 composition\(^1\). In other words, FDTs can be nested for creating more complex structures. An example of composite FDT is an adjacency list, which is an assembly 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 could 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 could be found from the linked list headed at array position \(i\).

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 distinction between FDTs and ADTs. If the dynamic set has its characteristics of dealing with objects which could be compared to each other, we are always dealing with an abstract data type, not a fundamental data type.

### 3.3.3 Data Structures

In order to implement an abstract data type (or fundamental 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.

**Definition 3.3.5** A data structure is a collection of variables of certain data types, connected some specific way.

\(^1\) Object composition is a technique in which new functionality could be obtained by assembling or composing objects to produce more complex functionality.
Separate definitions for the logical (abstract data type) world and for the physical (data structure) world is essential because of 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 kind of hiding implementation details is known as encapsulation. Second, the concept of an ADT is one important principle that is used for managing complexity through abstraction [37].

“A central theme of computer science is complexity and techniques for handling it. Humans deal with complexity by assigning a label to an assembly of objects or concepts, and then manipulating the label in place of the assembly. Cognitive psychologists call such a label a metaphor. A particular label may be related to other pieces of label, forming a hierarchy of concepts and labels. This hierarchy of labels allows us to focus on important issues while ignoring unnecessary details.”

Third, reusability [16] is one of the key principles in software engineering. In this thesis, we are going to fine tune the concept of data types in order to create a model which supports both the white-box reuse and black-box reuse in natural way. This is done by introducing the new abstraction called fundamental data type.

### 3.4 Case Study: Binary Search Tree

The dictionary is an interface which characterizes the structure in which we can store, delete and retrieve elements of a certain type. The binary search tree is an abstract 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 the 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.

The 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. The heap is an abstract data type that implements this interface. However, heaps are also binary trees, with the exception that they are complete².

At this point, it should be clear for the reader that both of these abstract data types are special cases of binary trees. And once we have a binary tree implementation, it should be possible to reuse this functionality in both cases separately. However, a binary tree is also a special case of even more generic fundamental data type **directed**

²A 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
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 also for 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. $(a, x) \in R \land (b, x) \in R \rightarrow a = b.$

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

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 fundamental data type that allows hierarchical relationship between various objects in such a manner that it is either empty or it contains a distinguished root node, and the remaining nodes form an ordered tuple of disjoint ordered trees, called the subtrees. Particularly, for a given node, the root of the nonempty subtree is called a child of the node and contrary the node itself is referred as the father of the root of the nonempty subtree.

**Definition 3.4.2** Let $T = (S, R)$ be a directed rooted tree, and let additionally $R'$ 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 $C_s = \{c | (s, c) \in R\}$.

We are now almost ready to formally define a very important special case of ordered trees called the binary tree. The binary tree is an ordered tree in which no node may have more than two subtrees that mean each node has degree at most two. In addition, if $d = 1$, 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 [11], to completely handle 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 additionally $f : S \times S \rightarrow \{1, 2, 3, \ldots\}$ 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 \( i \)th child of a node is absent. Additionally, by a \( k \)-ary tree we shall understand the positional tree in which for every node, all children with labels greater than \( k \) are missing. 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 \( R_a = \{(a, b) \mid (a, b) \in R\} \). 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. Additionally, 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 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 \( f(r, r_2) = 2 \). Additionally, let \( < \) be a total order of keys. Then \( M = (T, <) \) 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 which defines the operations

1. search() – find and return element of predefined type \( T \),
2. insert() – store and return element of predefined type \( T \), and
3. delete() – remove and return element of predefined type \( T \).

We formalize this by using an algebraic system [31]:
**Definition 3.4.6** Let $\mathcal{A}$ denote the set of all nonempty strings of letters from $A$ and let $S$ denote the subset of $\mathcal{A}$. Then algebraic system $O = (\mathcal{A}, ?, +, -)$ is called a binary search tree operations, if

1. $?$ is the binary operation search, such that for any string $s$ in $\mathcal{A}$,
   $$S?s = \begin{cases} 
   \text{true} & \text{if } s \in S \\
   \text{false} & \text{otherwise}
   \end{cases}$$

2. $+$ is the binary operation insert, such that for any string $s$ in $\mathcal{A}$,
   $$S + s = S \cup \{s\}.$$  

3. $-$ is the binary operation delete, such that for any string $s$ in $\mathcal{A}$,
   $$S - s = S \cap \overline{\{s\}}.$$ 

As mentioned in definition 3.3.2, an ADT is a mathematical model with a collection of operations defined on that model. Here the mathematical model is the Definition 3.4.5 (binary search tree model) and the operations are described on Definition 3.4.6 (binary search tree operations).

**Proposition 3.4.1** A binary search tree is an abstract data type $A = (M, O)$, where $M$ denotes the binary search tree model and $O$ denotes the binary search tree operations.
Chapter 4

Programmer Point Of View

Usually, the functionality of any abstract data type is derived from some more
generic underlying concept. For example, the binary search tree could be seen as
a special case of the binary tree. Giving an array implementation (implicit binary
tree) and the dynamic counterpart (explicit binary tree) implementation for
the binary tree, a programmer could reuse these components 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 chapter. More generally, fundamental data types are defined for ADT
implementations instead of implementing all of the ADTs from the scratch.

The challenge is, however, to determine the complete set of fundamental data types
to satisfy all possible needs that may arise in constructing new abstract data types.
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 fundamental
data types needed in the case. However, if no convenient FDT has been implemen-
ted, it should be clear that implementing one together with the actual ADT is at
most as difficult task to achieve as implementing the ADT directly. After all, while
implementing an ADT, it is good programming practise 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, in this thesis, reusability has a 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 into components called memory structures as
discussed in Section 4.2\(^1\). 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 4.3.

\(^1\) Actually, memory structures are needed for storing and playing the animation sequence back
and forth as discussed in Section 5.3.
4.1 Methods and Tools

Some people find it cumbersome to use formal methods to define a specification for a computer system, even though such definitions are mathematically rigorous and allow no ambiguity. Most of the object-oriented methods have very little rigor. The notation appeals to intuition better than formal definitions. However, these methods may be informal, but many people still find them useful.

In this thesis, we are going to use both of these strategies to illustrate our construction. Formal definitions are mainly used in the context of framework. After that, the technical point of view represents many of these ideas by using some of the well-known diagrams introduced in literature [12, 14, 16].

The diagrams are described as they appear, so we do not give any overall guidance how to interpret them. In addition, we assume the reader has some prior knowledge of object-oriented programming, notions and terms, such as objects, classes, instances, etc.

We have decided to use the Java programming language [12] as our primary implementation language, because of its many good characteristics. For example, Java provides the possibility to design and implement software that runs on many different platforms. Therefore, Java provides not only a meaningful environment for object-oriented design, but also a useful tool for developing suitable software for heterogeneous architectures and front ends. The quality and features of Java in open distributed learning environments has been discussed before [24], thus we do not repeat that discussion here.

4.2 Memory Structures

The simplest data structure is a linear, one-dimensional array. By a virtual array, we mean a list of objects referenced by a set of consecutive numbers. Every primitive data type can be seen as a special case of virtual array, namely an array of size \( s = 1 \).

The virtual array is an implementation for a memory structure. The characteristics of the memory structure are 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 neither have upper nor lower bounds for the virtual array, since the structure is implemented so that it allocates more memory if (and only if) there is no room for additional elements. The implementation of the virtual array is based on the idea of dynamic tables as discussed in the book of Cormen, Leiserson, and Rivest’s [11].

The following describes the interface and amortized time complexity of the operations on the virtual array.

1. constructor(); creation of an initially empty virtual array – \( O(1) \)
Figure 4.1: Relationships between inner object structures and the animator

2. void put(int i, Object o); store object o into index i - O(1)

3. Object get(int i); retrieve object at index i - O(1)

Every put operation is registered into 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 4.1 shows the meta-level relationships between memory structures, storage structures and the animator. The animator object is described more thoroughly in Section 5.4 (Algorithm Animation in page 69).

Because all primitive types have the same properties as memory structures, they could be animated, too.

4.3 Storage Structures

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

\(^2\)By *derived* we mean any mechanism that uses the virtual array as the only storage for instance variables.
Figure 4.2: Array representation of a linked list

In the following, we focus on implementational issues concerning the most common fundamental data types.

4.3.1 Table

The most common storage structure is an ordinary table. Most programming languages include an array structure which could be indexed. We have reserved the word array to denote the most common implementation of such a structure as described in the previous section. The storage structure table has therefore some additional restrictions. We have adopted the C programming language style to allocate arrays indexing from zero. Implementing any kind of storage structure table is trivial by using the virtual array and thus it will not be discussed here any further.

4.3.2 Linked List

There are at least three approaches to implement a linked list: the array implementation and the dynamic implementation which both represent the references explicitly and the static implementation which has implicit references. The dynamic implementation and the implicit representation of references are both discussed very thoroughly, for example, in the book of Shaffer’s [37], thus we describe only the first one here.

The array implementation (see Figure 4.2) stores the elements of the linked list and references to the next element (and possibly to the previous element) at contiguous array positions. The reference to the head of the linked list is at position 0. In a single linked list all elements are stored in odd positions and the corresponding links are stored in array cells immediately after each element.

The freelist could be used while inserting new elements in the list. The new element occupies the head of the freelist (element Y at position 13 in Figure 4.2) which is set to point at the next element. The appropriate next links should be updated and insertion is therefore completed. If the freelist is empty, the head of the freelist should be pointing at the end of the array (the next unused position in the array). Occupying the next unused position will not, however, cause any overflow since we additionally assume that our list is implemented by using the virtual array described in Section 4.2.
4.3.3 Trees

One of the elementary data structure is a 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 cases of trees.

We recall from Chapter 3 that a tree is a structure which allows hierarchical relationship between 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 $< T_1, T_2, \ldots, T_N >$. 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 abstract data type 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 of all, the last Definition 3.4.5 of the binary search tree model is far too declarative in 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 on the order of the subtrees. Thus, it does not preserve the order of the children as should be required in our declaration.

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 an 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 used for some special purposes in the future. Therefore proper naming of the objects and interfaces is essential.

In other words, it depends on the case what kind of functionality would be the most generic. In this case, we argue that there exist two generic objects that could be considered as 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\footnote{Note, however, that even though a binary search tree is a positional tree, which is an ordered tree, which is a directed rooted tree, it is the case that $B \subset P$ and $R \subset O \subset P$, where the sets of all possible binary search trees, positional trees, ordered trees, and directed rooted trees are denoted by $B$, $P$, $O$, and $R$, respectively.} because any ordered tree could be represented in terms of positional trees. On the other hand, the binary tree is so 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 which is stored at \( A[0] \). All subtrees of the node are then stored at the remaining positions of \( A \). Thus, here the binary tree is a special case of a tree in which there is at most two subtrees in each node. It should be noted, however, that this would be just the opposite, too. Namely, a tree could be defined in terms of a binary tree as discussed, for example, in the book of Lipschultz’s [30].

**Binary Tree implementation**

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 entity called node in which we have two references into the left and into the right subtree. We have to 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, has to allocate two times more space for the references than it is necessary. This is because we have to 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 from a node to 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 exists the corresponding mapping also for relations \( R \) and \( R' \). We can now implement the binary tree by an array \( A \), where the root of the tree is stored into \( A[1] \), and given the index \( i \) of a node, its father, the left child, and the right child can be computed as follows:

1. \( \text{father}(i) = A[\lfloor i/2 \rfloor] \),
2. \( \text{left}(i) = A[2i] \), and
3. \( \text{right}(i) = A[2i + 1] \).
Thus, the relation
\[ R = \{(A[i], A[j])| j = 2i \lor j = 2i + 1, i \geq 1, j \leq |S|, \{A[i], A[j]\} \subseteq S\} \]

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 this 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.

### 4.3.4 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.

### 4.4 Extended Structures

As the reader has noticed, the set of storage structures described here, is very much the same introduced as the “archetypes” in Section 3.3.2 (p. 31). 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 \( j \)): \( A[i \ldots j], |A| = (j - i + 1) \) is the size of the table
2. Linked List; explicit array implementation: \( L^a = A[0 \ldots 2n] \), where \( A[k] = \) element and \( A[k + 1] = \) pointer to the next index \( k', k \in \{1, 3, 5, 7, \ldots\} \).
3. Linked List; explicit dynamic implementation: \( L^d = A[0 \ldots 1] \), where \( A[0] = \) element and \( A[1] = \) pointer to the next node \( L' \).
4. Linked List (static list); implicit array implementation\(^4\): \(L^i = A\), where \(A[i] = \text{element}\), \(i \geq 0\).

5. Tree (k-ary tree); dynamic implementation: \(T^d = A[0 \ldots k]\), where \(A[0] = \text{element}\) and \(A[i] = \text{pointer to i'th child node} T'\), \(1 \leq i \leq k\).


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 4.4.1** Let \(A[u \ldots v]\) be an array and let additionally \(L\) be a set of linked lists. Then, \(A^L = A \circ L\) is called an adjacency list if for every position \(p \in \{u, \ldots, v\}\) there exists a unique list \(L_p = A[p] \in L\), and for every element \(e \in L_p\), \(e_{\text{key}} \in \{u, \ldots, v\}\).

The definition above does not explicitly state which linked list to use. The choice may dramatically affect the efficiency of an implemented data structure. However, algorithm analysis is out of the scope of this thesis, thus we do not discuss this question any further.

As the reader has already 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 4.4.2** Let \(A[u \ldots v]\) be an array and let additionally \(A\) be a set of arrays. Then, \(A^2 = A \circ A\) is called an adjacency matrix, if for every position \(u', v' \in \{u, \ldots, v\}\) there exists a unique array \(A_{u'} = A[u'] \in A\), and an element \(e = A[u'][v'] = A_{u'}[v'], e_{\text{key}} \in I\).

Other extended structures could be defined in a similar manner.

### 4.5 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

---

\(^4\)Although some text books [37] describes this implementation it is not recommended because in the average case insertion and removal of an element takes \(\Theta(n)\) time.
tree, as illustrated in Figure 4.3. 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 which could be introduced as an interface type dictionary. From now on, it is possible to define the operations of the abstract data type binary search tree in terms of the fundamental data type binary tree. For example, we could implement the operation search() by using the elements of the binary tree as follows.

**Example 4.5.1**

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. Object-oriented programming languages, such as Java, provides other features to support encapsulation, too.

Similarly, an abstract 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 subclasses from the binary tree and the set of operations are defined in the interface priority queue.

As we noticed, abstract data types form a hierarchy. The abstract data type AVL tree is a special case of the abstract data type binary search tree which in turn is

---

5Some authors [38] 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.
a special case of the dictionary. There are also other abstract data types which 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 4.3 or the Example 4.5.1 shows only
one possible way to construct relationships between objects. For example, also
dlegation (see Figure 4.4 on page 47) is 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 have to be expli-
citly made visible. Thus, it should be kept in mind that the framework represented
here has no restrictions concerning object-oriented design methods.

4.5.1 Reusability revisited

One of the key ideas of introducing the concept fundamental data type, is behind
the fact that well-defined FDTs may serve as reusable [16] building blocks for more
Figure 4.4: Delegation of binary search tree `getElement()` method to a binary tree instance

advanced data types.

We suggested that a fundamental data type could be inherited by an abstract data type. Thus the implementation of the abstract data type is defined in terms of the fundamental data type. 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 abstract data types by assembling fundamental data types.

**Example 4.5.2** Instead of subclassing a binary search tree (BST) from a binary tree (BT) (as in Figure 4.3) a binary search tree might black-box reuse the functionality of a binary tree. This is illustrated in Figure 4.4 in which the diagram shows the binary search tree delegating its `getElement()` operation to the 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. But as we saw in Example 4.5.1 the inheritance has very little to offer, thus making delegation a very attractive choice.

Finally, abstract data types could also be defined in terms of other ADTs. This is
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 4.5: Implementation of a queue by using two stacks.

also an example of black-box reuse. Let us consider the following definitions of the
stack and the queue.

Definition 4.5.1 Stack is an abstract data type in which an element may be inserted
or deleted only at one end, called the top of stack. Particularly it has the last-in-
first-out (LIFO) queue discipline. The set of operations defined on this model are
the following.

1. push() – insert an element into a stack,

2. pop() – delete and return an element from a stack, and

3. isEmpty() – return TRUE if stack is empty, FALSE otherwise.

Definition 4.5.2 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. Particularly it has the first-in-first-out (FIFO) queue discipline. The set of
operations defined on this model are the following.

1. put() – insert an element into a queue,

2. get() – delete and return an element from a queue, and

3. isEmpty() – return TRUE if queue is empty, FALSE otherwise.

Now, we can implement all operations of queue with two stacks as illustrated in
Figure 4.5.
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. This is one of the key challenges in visualizing data structures and algorithms as discussed in the next chapter.
Chapter 5

Visualizer Point Of View

5.1 Overview

In this chapter we describe the technical framework of the system called Object-TRAKLA. The overall goal is to design and implement a system which is capable of

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

of exercises in field of data structures and algorithms.

In order to achieve all these four goals, we have to develop techniques for handling the complexity of this kind of system. The first step was to introduce the formal framework for data structures and algorithms as discussed in Chapter 3. We use heavily 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 chapter) 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 object 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. The (user controlled) simulation is just an extension to the animation and visualization. These 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.

### 5.1.1 Two approaches

Before we go into the details, it should be noted that the system supports two different kinds of approaches to develop a software visualization from the visualizer point of view.

Algorithms can be completely emulated (user controlled simulation for a particular algorithm or program) as with TRAKLA, or the visualization can be achieved by conforming the proper application interface for smart components as with the JDSL Visualizer. In both cases an actual data structure is created which could be manipulated through the graphical user interface. The user manipulates the GUI which in turn delivers the command to the actual data structure by invoking proper method. After the operation is completed, the display is updated and the user may perform another command. On the other hand, if an actual algorithm is run to modify the structure, the display is updated and the algorithm animation is played for the user. This approach requires usage of ready-made visualization components or disciplined creation of user-made components.

### 5.1.2 Organization of this chapter

In Section 5.2 we will revisit the theoretical framework and present the simulation model needed in the rest of this chapter. In Section 5.3 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 5.4 summarizes the framework developed so far by briefly discussing the algorithm animation in terms of the designed framework. In Section 5.5 we extend the construction to support user controlled simulation of algorithms and data structures. Finally, in Section 5.6 we summarize the discussion as a whole.

### 5.2 Simulation Model

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

5.2.1 Theoretical framework revisited

The numerical values of a given data type could be considered as the numerical data to be visualized. Visualization of any given primitive data type is trivial\(^1\), because they 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.

**Definition 5.2.1** Primitive visual data type \( T_P \in \mathcal{N} \times \mathcal{R}_\mathcal{N} \) is a visualization for primitive data type built into a programming language, where

1. \( \mathcal{N} \) is the set of primitive data types, and
2. \( \mathcal{R}_\mathcal{N} \) is the range of given data type \( N \in \mathcal{N} \).

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, a visual primitive data type behaves like a token, which could be delivered from one point to another as in animation. On the other hand, this kind of single delivery could be considered as an instruction as described in Definition 3.2.1 (the definition of an Algorithm on page 28).

In order to visualize more complex data types such as arrays, lists, trees, etc. we need a well defined model. It 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 some 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 to what position or order to put all of its components. In addition, the container may also include a set of visual references. A

---

\(^1\)Ford [13] shows that authors visualize even an integer type in several possible ways. We argue that discussion **how** a given primitive type should be illustrated, is a customization issue, not a visualization issue.
visual reference is a binary relation between two visual components made 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 5.2.2** 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 $n \in N$ has its key attribute, denoted by $n_{\text{key}}$. The key attribute is the reference into the visual data type $\mathcal{T}$ which the visual component is representing.

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

**Example 5.2.1** Let $M = (T, <)$ be the abstract 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 $p_{\text{root}}$ 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 $n \in N$ is pointing to some AVL tree node $s \in S$ and for each relation $p = (a, b) \in P$ 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 $s_{\text{key}}$ should be a primitive type, e.g. a string, that could be illustrated as a primitive visual data type.

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, some 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.
5.2.2 The Model

So far, we have defined the term visual data type and described the outline of it. We propose that this sketch is rich enough to be able to emulate any data structure known so far.

Usually, a simulation model is a metaphor to some 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.

**Definition 5.2.3** A simulation model is the implementation of a visual data type which emulates 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 allowed that the structure may have many possible simulation models to illustrate its behavior. It is the visualizer's choice to make the decision on which metaphor to use. It may be even possible to have several visualizations active at the same time. Thus, we argue that the framework should be extensible and customizable in order to serve also the future requirements.

5.3 Data Structure Visualization

In this section we will discuss the visualization of fundamental data types. The simulation model, presented in the previous section, formally defined the visual components needed to visualize an abstract data type. It also gave the basis for algorithm animation and user controlled simulation as discussed later in this thesis. In this section we will briefly describe some technical details how this formal framework could be applied for visualizing a single state of a data structure. We also define the most common fundamental data types in terms of the simulation model.

5.3.1 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. If there is no chance of 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 illustrated in Example 5.2.1 at the end of the previous section. A component may have a key, which in turn could be either another container or a piece of data (primitive data type). Thus, any visual representation may be as complex as required.

Example 5.3.1 Figure 5.1 shows the object structure of a parse tree. The surrounding box titled at $a = (b > c) ? b : c$ represents the container of the tree. The nodes of the tree are the components of the representation illustrated by circles. The connecting lines between components are the references and finally, the keys of the nodes illustrate the stored data represented as rounded rectangles labeled with \{$=, \text{a}, ?, >, \text{b}, \text{c}\}$.

The Definition 5.2.2 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. In Figure 5.1 the title 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 container above has its title name “$a = (b > c) ? b : c$”. On the other hand, a node or a reference may also have a name, illustrated as a label beside the object. Second, coloring is another possible method to decorate an object. Thus, also the color could be set depending on the usage of the object. These, and other decoration issues, are discussed further in Section 5.3.3.
Implementation

As discussed above, the system and all of its visual objects should support several methods to function properly. Thus, we briefly describe the overall design and architecture of the visual objects here. The top level design is illustrated in Figure 5.2 together with an example of object structure relationships of the VisualTree.

All components described in definitions 5.2.1 and 5.2.2 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; and 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 some structure is already shown in some place on the display or not. This is essential, for example, in order to prevent infinite loops in some 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\(^2\) 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 5.2 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 5.1.

**Example 5.3.2** The VisualTree in Figure 5.2 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 which is a storage structure. Thus, the elements may be some complex data structures or just some primitive data types such as the keys in Figure 5.1.

\(^2\)There exists only one reference type, at the moment.
Figure 5.2: Simulation model and the design of VisualTree
5.3.2 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 fundamental data types. These ready-made visualizations could be nested to produce more complex visualizations. Thus, the ready-made visual container also behaves like the corresponding fundamental data type or its implementation (storage structure) in which composition was introduced for the first time.

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

At the moment, we have visualizations for just arrays and trees. We will refer to these visualizations as VisualArrays and VisualTrees, respectively.

If we would like to implement a new abstract data type, 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 section. In fact, this is exactly the case. The reuse of the storage structures does not only give us the benefits of reusability, but also the visualizability. In other words, the visualization does not know the actual subtype of the structure it is representing. It only knows the supertype (storage structure) which is actually enough\(^3\). Thus, a single visualization may represent many possible storage structures.

**Example 5.3.3** 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 5.3. The AVL tree may additionally reuse the binary search tree functionality by not subclassing the binary tree directly, but 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 also 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 any 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

\(^3\)Enough with respect to fundamental data types. Of course, we possibly cannot cover all details a data structure may have. This is, however, possibly fixed by decoration, as discussed in Section 5.3.1.
Figure 5.3: Extended hierarchy of abstract data types
Figure 5.4: Explicit and implicit binary heap representations

visualizations to choose from. The following example illustrates our approach of having many possible visualizations for a single storage structure.

Example 5.3.4 The static binary tree implementation (the class BinTre) 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 of them are represented in Figure 5.4. It should be noted, however, that both of these representations in the figure illustrate the same storage structure. Thus, while making an update for one representation also affects the corresponding change to the other.

As we can see, the visualizability could be achieved by implementing an interface. Thus, every visual representation should have a similar interface as the visual tree representation did. This also implies that we could implement this interface without reusing any existing containers. It should be noted, however, that doing that 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\(^4\) the structure. This kind of animation is quite a close to the visual debugging as some authors [33] prefer to call it.

\(^4\)Simulation constraints are discussed later.
5.3.3 Interfaces

The following interfaces define the methods a storage structure should implement in order to gain a visualization. Note, however, that the customization issues are not discussed here. For example, one might argue that the explicit binary heap in Figure 5.4 should be drawn from the left to the right instead of from top to bottom. We consider these questions as customization issues. For any representations here, a skilled programmer could easily construct a dual representation in which such properties are taken into account, thus they are not discussed any further here.

Decoration

Before defining all the actual representational interfaces we briefly describe how to decorate the representation, if required. For example, the binary heap in Figure 5.4 could be decorated by providing the indices also for the explicit binary heap representation and by placing each corresponding index beside the nodes. 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 5.3.5 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 the node H the balance condition \( |b(H)| = 2 > 1 \) is not valid, thus violating the AVL balance property.

Similarly we could decorate other visualizations, too. This requires, however, an additional interface to be implemented in the data structure implementation. For example, the previous decoration in Example 5.3.5 could be achieved by implementing the following LabelDecorator interface.

1. boolean isLabelEnabled(); - enable/disable labeling
2. void setLabel(String s); - set label
3. String getLabel(); - get label
4. boolean isReferenceLabelEnabled(); - enable/disable labeling of references
5. void setReferenceLabel(int i, String s); - set reference label
6. String getReferenceLabel(int i); - get reference label

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
Figure 5.5: AVL tree single rotation
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 (see Section 5.5 User Controlled Simulation for more details).

Other interfaces could be introduced and defined similarly. Of course, the corresponding implementation should be added to the visualizations.

**Array Interface**

One of the elementary visual structures is an array. As we recall from Section 4.2 we have implemented the virtual array to be the very basic building block for more complex constructions. Thus, every fundamental data type implemented here as a storage structure should include the virtual array in some form. From now on, it is very easy to declare an interface that corresponds to all necessary methods the virtual array implements. Thus, for every storage structure we already have the implementation needed to represent the structure as an array.\(^5\)

As a VisualType the VisualArray is a representation in which we only have VisualComponents (positions called VisualArrayComponents) but no VisualReferences. Additionally every position has an attribute index that is shown as an integer at the bottom of the VisualArrayComponent.

Any data structure can be represented as an array by implementing the following interface **Array** as described above. Most of the cases, only the first method is actually needed and the implementation for all the other methods could be omitted. However, there should be a way to disable arbitrary manipulation of the virtual array. Therefore, if the getArray() method returns the data structure itself, it is possible to gain the full control over the representation process by implementing the rest of the methods.

1. public Array getArray();
2. public Object getElement(int i);
3. public void setElement(Object o, int i);
4. public int getFirstIndex();
5. public void setFirstIndex(int i)
6. public int getLastIndex();
7. public void setLastIndex(int i);

The lower representation of the binary heap in Figure 5.4 is an example of the array representation.

\(^5\)Recall the definition of the storage structure in which we required that the structure is derived from the virtual array.
Figure 5.6: Directed acyclic graph

Tree Interface

In Figure 5.1 (page 55) we saw an example of the parse tree that is represented by using the VisualTree representation. Any object structure implementing the following interface Tree could have this representation.

1. public int getSubTreeCount();
2. public Tree[] getSubTrees();
3. public Object getElement();
4. public void setElement(Object v);

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. This does not, however, imply that it has to be an instance of the same class as its father node. It implies only that both of them are of (interface) type tree. Finally, the key stored at a node should be type of Object, thus satisfying the definition of the fundamental data type.

However, even graphs may implement this interface. The implementation of the VisualTree 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 as original nodes (illustrated by circles). This has, however, other implications, too. For example, the cross arcs of a graph are not drawn pointing to the actual node, but a terminal node. This situation could be illustrated in the representation by highlighting also the original node while highlighting a terminal. Obviously, terminals do not have any empty subtrees as leaf nodes may possibly have.

Example 5.3.6 Figure 5.6 shows a labeled digraph with 10 vertices and 20 arcs. Figure 5.7 shows the corresponding Depth First Search Traversal Tree (DFST-tree).

\footnote{This depends on the implementation of the actual tree.}
Figure 5.7: DFST-tree of the directed acyclic graph in Figure 5.6

Using the VisualTree representation. The graph is acyclic, thus there are no duplicates along any path from the root to a leaf. There are, however, cross arcs. This 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, 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). 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 value. Thus, for two distinct nodes the keys may be equal. Let us say we have the same key B in two distinct nodes b1 and b2 in the tree. Changing the value of b1 does not affect the value of b2 or vice versa.

Such a situation could be avoided by labeling the keys as B1 and B2. 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 done, for example, in Figure 5.5 on page 62. The AVL tree balance factor is represented as an integer (label) beside the node. The label could be any other string, too. This allows to make distinction between two separate nodes with equal keys by labeling nodes with distinct names.

**Linked List Interface**

Actually, we could see the linked list representation as a special case of the tree representation, because any linked list could be seen as a tree in which for all internal nodes the number of subtrees equals one. An example of such a list representation is in Figure 5.8.
5.3.4 Building higher-level representations

Interestingly, from now on, we could use the array representation and the list representation (tree representation) discussed above to produce an adjacency list representation. With similar manner we can build more complex representations.

Adjacency List representation

Let us assume we have a set $V$ of nodes and a set $E \subseteq V \times V$ of references. The adjacency list representation consists of an array $A[0\ldots k-1]$ of $k = |V|$ lists, one for each node in set $V$, and a function $f : V \rightarrow \{0, 1, 2, \ldots, k-1\}$. 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 5.3.7 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 5.9 we see this construction as an adjacency list.

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 5.9. We denote this construction by $A \times L$.

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 $M = A \times A$. 
Figure 5.9: Adjacency list representation of the tree in Figure 5.4.

Case Study: how to represent $B^+$-trees?

A $B^+$-tree is a commonly used balanced search tree structure in which all data are stored at the leaves, and the branching factor of the internal nodes is maximized, thus internal nodes store only simple keys and child references [11].

In a 2-3-4 tree every internal node has either 2, 3 or 4 children (maximum of 3 simple 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 we have to have an inner structure in the nodes to store all simple keys the internal nodes may contain. However, for representing the overall branching structure, a simple tree representation should be adequate.

**Proposition 5.3.1** 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. Thus, 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, the decoration information could be put away, thus leaving just the plain sequence of consecutive keys.

**Example 5.3.8** Figure 5.10 illustrates the 2-3-4 tree. The nodes are indicated as ellipses and rectangular boxes inside them with one, two or three value fields. Leaf nodes contain the data illustrated by capital letters and internal nodes store the keys illustrated by small letters.
5.3.5 Implementation

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 as a valid case where the proper representation is the DFST-tree of the graph, as illustrated in Figures 5.6 and 5.7. Thus, 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, the good design and implementation of these kinds of visual components is essential. The superclass VisualType offers several techniques to handle a situation like above, where we avoid to enter infinite loop. However, functionality like this, among other features of the class VisualType, is not discussed here any further.

5.3.6 Summary

As we have shown above, we can represent many different kinds of data structures by two carefully designed and implemented representations, namely by 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.

### 5.4 Algorithm Animation

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, some kind of control of the changes to these representations is required. Obviously, this control could 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 the 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 is working together with all the other structures around in order to function properly. Thus, there is a special animator object which takes care of the animation sequence storage. The animator object has to be informed every time an assignment takes place. It is implemented as a singleton [16] (pattern), thus there exists only one instance of it in which all actions are stored by invoking the method \( \text{store (object, index, old_value, new_value)} \) as illustrated in Figure 4.1 (on page 39).

The parameter object refers to the invoking object, the index is the (virtual) array position that is storing 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 backward the assignment. This is done simply by invoking \( \text{obj->set(index, old_value)} \). This requires that the invoking object conforms to the interface Memory Structure in which the method \( \text{set(int i, Object o)} \) is declared.

The most convenient way to enable the animation for the structure is the reuse of the virtual array that was introduced in the previous chapters. Thus, declaring all instance variables as virtual arrays (a single variable could be seen as a virtual array of size \( s = 1 \)) encapsulates the animation process from the application. There are, however, a few exceptions. Because we possibly cannot store primitive types (such as int, float, double, ...) as objects we have to 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, does not make any good without a representation. Particularly, 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 4.3. However, redesign of the actual data structure is possibly needed. This is done in terms of fundamental data types as described in Section 3.3.2.

However, the result should be a highly visual dynamic algorithm animation. In Figure 5.1 we see 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 representations (array and tree).

The control of the algorithm animation process is discussed further in more details in the next section together with the idea of user controlled simulation.

5.5 User Controlled Simulation

Simulation is by definition an experimental technique where the data generated by each experiment has to be collected, summarized and represented in a meaningful way. We have demonstrated the use of storage structures for implementing fundamental data types to update and report their values in algorithm simulation. In addition, a number of visualizations were defined for representing model compon-
In the case of user controlled simulation of algorithms we are interested in the
detailed study of the dynamic behavior of data structures. Sequences of data struc-
tures are directly represented through event procedures. This paradigm permits
the representation of systems at an essentially unlimited level of detail.\footnote{For
example, Stern et al.\cite{Stern1980} describes 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
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.}

Simulation experiments are performed under the control of an animator entity with
event process orientation. Model executions are guarded by some terminating al-
gorithm or by human interaction. Furthermore, we are especially interested in the
human controlled simulation, which we refer to as user controlled simulation.

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. Object’s behavior can then be illustrated by
sequence of discrete states or by continuous process. Either the algorithm or the
human controlling the model can be the dominant participant. In order to integrate
all of this, the animator must continually check for both state and control events.

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

1. moveBackward()
2. moveForward()
3. moveBeginning()
4. moveEnd()
5. play()

The set of state events may vary between the models to be simulated. On the
other hand, there exists only a limited set of actions a user may perform during
the execution of a simulation. Thus, the simulation model should be responsible for
mapping these actions to particular state events. The very 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, 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.
1. click() - read this object
2. double click() - write read object to this
3. drag() & drop() - insert dragged object to this
4. popUpMenu() - delete, customize, change object's representation, etc.

Of course, there could exist also much more sophisticated actions, such as select
a set of objects or drag & drop of selected set. It should be noted, however, that
the primary use of the user interface is intended for students and educational pur-
poses. Thus, there is usually not much time to use for learning how to use the tool.
Therefore, it should be kept as simple as possible.

Naturally, the set of actions above could be targeted to any of the visual objects;
VisualContainer, VisualComponent, VisualReference, or VisualData; and the func-
tionality may differ from object to object. However, the simulation model gives
a meta-level proposition for the actions as described above. There is, however, a
number of exceptions which are briefly discussed in more detail here.

5.5.1 Read and Write Operations

From the animation point of view, the only operations we are interested in are those
of which somehow manipulate the data structure to be visualized. In procedural
programming languages this manipulation takes the form of assignments. An as-
signment is the operation that stores the value of an expression in a variable. Thus,
a visual component is considered to be the expression above; and the variable refers
to some other visual components key attribute. Moreover, we may assign any FDT
into the key because the data structure that the visual component is pointing to
(representing) is defined by definition as the fundamental data type.

A single click on an object assigns an internal reference to point to 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 cannot handle
the write operation neither, the action is again forwarded to the container holding
the component with the additional information where the write operation occurred
 esp, 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 [2] as demonstrated in the following example.

Example 5.5.1 In Figure 5.11 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 $B_0 \ldots B_5$ that are represented as the nodes of the tree. The node is decorated
with label if the corresponding block has the label $L_0 \ldots L_3$. The basic block $B_4$
is represented in detail by embedding the corresponding linear code. The block contains
two expressions which are represented in the virtual array $B_1$. Furthermore, the first
code line is illustrated 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 the time. Even
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

1. any object itself is responsible of taking care of the validity of its keys and
2. no object structures (other than visual objects) are created during the read
   and write actions.

For example, if we are trying to take an action 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 give a cause for the same change in the dual visualization and vice
versa.

5.5.2 Insert Operation

The drag-and-drop action is sometimes referred as an insert operation. This is
especially true in the case of abstract 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
Basic Blocks in Linear Code:

L0: if low >= high goto L3

B0: 

try = (low + high)/2

if key >= table[try] goto L1

B1: 

high = try - 1

goto L0

B2: 

L1: if key <= table[try] goto L2

B3: 

low = try + 1

goto L0

B4: 

L2: return try

B5: 

L3: return low

B6: 

Figure 5.11: 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 is illustrated in the expression tree at array position 0; and the second expression tree at array position 1 is minimized.
the frame. Thus, the target has to implement a special insert method that takes
the source to be inserted as an argument. This method is declared in the special
interface ADT that indicates that the visualized structure is an abstract 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 an ADT, the drag & drop action is
interpreted as the write action.

5.5.3 Miscellaneous Operations

There is, however, no natural way to express the delete operation declared in the
interface ADT. Thus, such miscellaneous operations are hidden in pop-up menu's
attached to all visual objects.

There exists a set of very powerful tools in superclass VisualType which increase
the functionality of any visual object by declaring a new entry for its pop-up menu.
By default, the delete operation is included there and should be implemented for all
visual objects.

Similarly almost any kind of additional operations could be inserted and implemen-
ted here. For example, the AVL tree implementation may include a special set of
rotate operations to illustrate the behavior of this particular balanced search tree.

Decoration

There exist also some predefined decoration operations in default pop-up menu.
For example, the label (name) of a component could be enabled, disabled and set
by selecting the proper operation. If the component is type of LabelDecorator,
as discussed in the Section 5.3.3, the actual setLabel method is invoked for the
visualized data structure. On the other hand, implementing this interface is not
necessary in order to visualize component labels, i.e., any visualization could be
decorated by labels. Note, however, that in this case the label is not a property of
the actual visualized data structure but rather a property of the visualization and
thus does not appear in every instance of the visualization of the data structure.

5.6 Summary

Recently, the research community has paid much attention to the concepts of pro-
gram 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 imple-
ment, for example, in the Java programming language. However, the primitives
necessary for representing a data structure and the handling of dynamic relation-
ships 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 ready-made visualizations for these object structures.

The design of a new abstract data type is defined in terms of fundamental data types. After that, the implementation is straightforward by reusing the existing implementations of these FDTs. The FDT instances include the possibilities to animate, visualize, and simulate the new abstract data type implementation.

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 on 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.
Chapter 6

Educational Point Of View

6.1 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 how the framework developed could be used for educational purposes in this sense.

6.1.1 Teacher and Student Points of View

The educational point of view includes two separate viewpoints. In Section 6.2 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 6.3 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 a task for practice. Moreover, an example shows how to provide feedback to the student on his/her performance in terms of automatic assessment.

6.2 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 [17] demonstrated the usefulness of flowcharts. After that, several systems have been developed to generate flowcharts automatically [23, 36]. This kind of “static” technique (the opposite to dynamic) has a role to play even today. On the other hand, some of the early-stage approaches towards “dynamic” techniques were those of Knowlton’s [21, 22] and Baecker’s [5] films. These were also the first to address the
visualization of running programs.

### 6.2.1 Visual Examples

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 can use the tool to illustrate data structures and algorithms as done in Figures 4.2 (Linked List, p. 40), 5.1 (Parse Tree, p. 55), 5.4 (Binary Heap, p. 60), 5.5 (AVL Tree Rotation, p. 62), 5.7 (DFST-Tree, p. 65), 5.8 (Tree Representation of List, p. 66), 5.9 (Adjacency List, p. 67), 5.10 (2-3-4 Tree, p. 68), 5.11 (Dominator Tree, p. 74) or Table 5.1 (Buildheap, p. 70).

### 6.2.2 Classroom demonstration

A 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. One possible use for Object-TRAKLA is that of a demonstration tool in a computer science class to help illustrate and represent algorithms. It could be used in teaching to help students understand how algorithms work or to help illustrate how the data structure is constructed.

One obvious way is to use a sequence of visual examples, as described above (see, for example, Table 5.1). These pictures could be represented with the overhead projector as a sequence of slides. On the other hand, Object-TRAKLA can also be used for showing the whole animation live, still having the possibilities to:

1. define the example by running an actual algorithm, or create user controlled simulation of an algorithm on line;
2. play the animation sequence back and forth;
3. have several windows (views) representing the same data structure in different representations etc.

### 6.3 Electronic exercise book

As we have seen, the framework can be used for demonstration purposes. The examples have mostly concentrated on how to illustrate data structures and algorithms
to the students or to some other audience. Furthermore, many papers have been published on how to teach data structures and algorithms. It is actually surprising that there has been so little discussion about how to learn data structures and algorithms.

TRAKLA system has been a “learning environment” since the very beginning, because it was implemented as a student project at the Helsinki University of Technology. On the other hand, the whole WWW-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 the learning process.

### 6.3.1 Laboratory Experiments

The course material for laboratory experiments 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, to the 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 the experiments. The idea is that when studying a set of new algorithms with the aid of the framework the user could observe strengths and weaknesses of the algorithms.

One example could be the research study in which the student is trying to find out the worst cases of several sorting algorithms, let us say, Shellsort, quicksort and insertion sort. As it may turn out to be very hard to construct an example of the worst case of Shellsort, the framework can be used for determining the worst case by means of simulation. Running the algorithm with all possible permutations of a fixed set of keys can easily determine the worst case. At least in the case of very small input sets. This experiment can be a valuable learning aid for the student to verify that his or her original mental image of the complexity of the algorithm was correct. Moreover, in the case of quicksort, in which the construction of the example is quite an easy thing to do, the tool could be used for illustrating the results; and the worst case example could be easily illustrated by the student in terms of user controlled simulation.

### 6.3.2 Object-Oriented Programming and Design

The framework introduced to permit automatic animation provided a starting point for object-oriented design. For example, one of the pedagogic goals could be to teach how to use application interfaces within the object-oriented paradigm as discussed by Baker, Boilen and Goodrich [6]: “Instead of writing single-use, throw-away implementations (data structures) for a specific task, students write generic, reusable
implementations that conform to given APIs. Students then implement algorithms assuming the availability of data structures that realize these APIs”.

The same idea could be adopted to Object-TRAKLA in which the API is one of the visual component interfaces. We will even provide a set of reusable fundamental data types as a starting point for the implementation.

Because all of the components are designed to conform to interfaces, this modular design should make it is easy to expand the current prototype. Thus, also the further development of the framework could be possibly done as a student project.

### 6.3.3 Automatic Assessment

One possible use for the whole system is to produce exercises to be solved by students in the class of algorithms and data structures. This idea is adopted from the TRAKLA system and implemented as one of the main applications for the framework.

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, T, L, M, A, C, D, and X”. The tools needed for this exercise include two fundamental data types. 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. The student should drag & drops the keys into the heap and perform the swap operations illustrated in Table 5.1 (on page 70). 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 5.1), it is also possible to create the model solution for the exercise. In fact, it is also 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 an interface Exercise. The interface defines several important methods. First, we need a method to determine all the data structures the exercise algorithm needs during its execution. These data structures are expected to be fundamental data types and ready-made visualizations for them should exist. Second, we need a simple method to return the string in which the assignment is described. Third, 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 [43].

From now on, the framework is capable of visualizing the exercise by showing the
assignment in some text window and by illustrating the fundamental data types 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.
Chapter 7

Discussion

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. In addition, a set of visualizations has been developed for these FDTs in order to illustrate the behavior of algorithms. These implementations provide an extensible set of learning modules and a common application framework for software visualization.

As an application for the framework we have illustrated how to apply it for an “electronic exercise book”. The core of each exercise is the exercise algorithm and the data structures the algorithm manipulates. The tool provides the way and means to visualize and animate the data structures. The tool is also capable of user controlled simulation of the exercise algorithms, obtaining immediate feedback on student’s performance, visualizing the data structures while the algorithm is executed automatically (model answer), or using the package to visualize student’s own algorithms. Thus, we have provided a very effective and flexible modeling framework for this specific application area.

The programmers have to reuse the memory structures in order to obtain all the functionality needed to properly animate objects. This might become an obstacle to produce 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, but this person’s duty is very different than that of the current version of TRAKLA. The content expert can focus on tutoring some students implementing new features into the system and is also needed to provide all the necessary fundamental data types in order to accomplish a given task to produce more advanced data types. There should be enough flexibility to construct new data types based on reuse of existing ones. Thus, the visualization, animation, or even the simulation do not require extensive programming by the students. This is because they can reuse the existing code that we 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 about 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 [28] 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 simulation exercises where the topic indeed is the illustration of very clear basic structures, thus stereotypes!

7.1 Contributions in terms of Taxonomy

The overall application framework provides TRAKLA system with a set of improvements. We summarize these improvements here in terms of the taxonomy represented in Chapter 2.

7.1.1 Scope

Most of all, the new system will provide us with a capability of visualizing a completely unrestricted set of algorithms. This is because of the user controlled simulation that does not require any actual programming language to be involved. On the other hand, there is an option to code the algorithm to be visualized in the Java programming language. Moreover, the Java programming language does not have any operating system nor hardware restrictions\(^1\). Thus, we argue that we gain the maximum possible generality in contrast to the other SV systems developed so far.

Also, the scalability qualifies for the highest possible level. Because there is no code annotation involved, there are no limits to the size of the code. On the other hand, the graphical representations of fundamental data types should be scalable, thus also allowing very large examples. At least we can say that the system does not set any fundamental limitations to the size of the data, but it is the hardware (the size and resolution of the display, the amount of memory, etc.) that might limit some actions.

7.1.2 Content

We speculate that the idea of fundamental data types should improve the data visualization capabilities and provide programmers with a very flexible and easy to

\(^1\)It runs on any modern platform.
use environment to create visualizations of any kind. Thus, at least the ranking of data visualization should increase. On the other hand, it is possible to develop the system further by adding some program animation features. In this respect, we expect to have better ranking also on code visualization. Thus, the real challenge, in the near future, will be to combine the program animation and the concept animation.

7.1.3 Form

The new presentation style is based on the fundamental data types and their visual counterparts. The representations could be improved separately from the actual data structures visualized.

At the moment, the user may change the size of the visible structure from 1 point to 256 points, thus the system also has zooming capabilities. Moreover, the whole design of FDTs provides the designer with a set of components, and the flexible framework in which it is possible, for example, to manage content complexity using a stepwise refinement technique.

There are also features for elision control implemented in some of the ready-made visual components. For example, in the tree representation, the user may elide a subtree by shrinking it to a triangle representing a whole subtree. On the other hand, it is possible to open a new window for a given detail, let us say for a subtree.

7.1.4 Method

The new framework includes several possible specification methods. It is still possible to simulate the algorithm, but from now on it is also possible to specify the visualization in terms of interfaces, or by using some proper library structures.

The interface approach requires that the user’s code conforms to a simple application interface. On the other hand, using library structures simply requires reusing of some existing library component(s). None of these techniques do not require any modifications to the student’s code in order to facilitate visualization. On the other hand, in order to be able to simulate user-made code, it is recommended that the ready-made library components are used\(^2\). As a result, the framework has the highest possible code ignorance allowance (CIA), and the System-Code Coupling (SCC) is the lowest possible, indicating that the SV system is loosely tied to the program it is visualizing (in simulation mode).

In addition, many new improvements to the tailorable have been introduced.

\(^2\)Although this is not required because it is still possible to achieve the full simulation capabilities by using virtual arrays. Anyway, this requires careful design and implementation, and additional knowledge about the framework.
As we have stated before, the tool provides full control of resizing, scrolling, and zooming, not to mention the possibility to change the layout of a data structure live by direct manipulation.

7.1.5 Interaction and Effectiveness

Examples of interaction include buttons, menus, and scripted programs. However, user controlled simulation is also one of the interaction styles. This also includes full support to navigate through the visualization.

It is not clear whether the elision control properties discussed in Section 7.1.3 should be discussed here again, but in the case of this environment, the boundaries of Form and Interaction get blurred. This is because of the nature of the user controlled simulation. In Form we are dealing with the characteristics of the output of the system (the visualization). On the other hand, Interaction describes the characteristics of the input of the system. In the case of user controlled simulation the input directly modifies the output, thus putting the person interacting with the system in place of “visualizer” and “user” at the same time.

However, the elision control capabilities are greatly improved and the system should qualify as a tool to software engineering (see Section 7.1.3 for more details). The temporal control features are also improved because of the much more sophisticated animator component. These techniques allow the user to change the mapping between execution time and real time (stop and start animation, rewind to the beginning or to the end, etc.). Reversing the direction of time is also possible, so that the algorithm runs backwards.

Serious scripting support is required to fully demonstrate the system, and it is particularly useful in classroom situations where the animation can be run like a videotape or the students can go through it at their own pace. The framework supports several techniques for this by providing

1. ready-made exercises,
2. user-made animations using visualized FDTs and user controlled simulation, and
3. the possibility to include user-made algorithms in the system.

However, at the moment the framework is missing to support any method to record the user-made animation sequence to be animated later on. Some kind of standard format for many visualization systems would be a feasible solution.

Because effectiveness is a highly subjective measure, we leave it to the students to evaluate how well the system communicates information to the actual users. In addition to the subjectivity, it is clear that this kind of evaluation cannot be
completed until the system is applied to a real world example. Therefore, this topic
is not discussed any further.

7.1.6 Summary

This thesis has tried to lay a foundation for simulation framework of data structures
and algorithms. We conclude the main characteristics of the framework as follows.

1. The framework provides fast prototyping of user controlled simulation applica-
tions.
2. Implemented prototype enables visualization, algorithm animation and user
controlled simulation for user-made code without any restrictions concerning
object-oriented design methods.
3. The specification methods allow separate implementations for actual data
structures and for visualizations.
4. One visualization could be used for displaying many different data structures.
5. One data structure could have many visualizations.
6. The prototype demonstrates the exercise interface and the possibility to auto-
matically assess the exercises.

7.2 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,
2. the recording facilities for the user controlled simulation,
3. including the system in the WWW-TRAKLA learning environment,
4. global library of exercises and demonstrations (requires web file system),
5. implementation of internal pseudo-programming language interpreter (in order
to provide on-line programming without compilation),
6. study of program animation capabilities,
7. package for automatic transformation of pictures to some latex compliant, and
8. further customization of representations.

The first four tasks are the primary research topics that complete the work needed to create the electronic exercise book and the learning environment. Moreover, the framework itself could be developed further to provide 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


Index

2-3-4 tree, 67

abstract data type, 30
actions, 71
adjacency list, 32, 43, 44
adjacency matrix, 43, 44
ADT, 30
algorithm, 28
algorithm animation, 1, 8, 15, 69
algorithm simulation, 1, 8, 19, 70
algorithm visualization, 8
analogy, 54
animation
  algorithm, 15
  concept, 14
  program, 15
Animator, 39
array, 31, 40
  virtual, 38
assignment, 72
attribute, 29, 53, 55
  key, 29
  non-key, 53
  visual, 53, 55
automatic assessment, 2, 9, 80

binary search tree, 33, 35, 44
binary search tree operations, 36
binary tree, 35

composition, 32, 47
concept animation, 14
control event, 71
dad-link, 42
data model, 31
data structure, 32
  explicit, 43
  first order, 53
  implicit, 43
  k'th order, 53
  second order, 53
data type, 29
  generic, 30
  primitive, 29
decoration, 55, 61, 75
degree, 34
delegation, 47
delete operation, 75
depth first search traversal tree, 64
DFST-tree, 64
dictionary, 33, 35
directed rooted tree, 34
dynamic set, 29
dynamic table, 38
dynamic tree, 42
electronic exercise book, 3, 78
emulation, 18, 51
encapsulation, 33
event
  control, 71
  state, 71
exercise, 80
extended structures, 43, 44

FDT, 31
  composite, 32
FIFO, 48
function, 28
fundamental data type, 31, 32
generic data type, 30
generic type variable, 30
graph, 28, 43
  acyclic, 28
  connected, 28
  directed, 28
  undirected, 28
insert operation, 73
instrumentation, 17
interface, 30, 61
- ADT, 75
- Array, 63
dictionary, 33, 45
- Exercise, 80
LabelDecorator, 61
- Linked List, 65
MemoryStructure, 69
- priority queue, 33, 45
Tree, 64
virtual array, 38
interface type, 45
key, 29
- key attribute, 29, 53
label, 75
- learning environment, 6
LIFO, 48
linked list, 40
list, 40
matrix, 66
memory structure, 37, 38
metaphor, 33, 54

Object-TRAKLA, 2, 3, 50
operand, 30
operator, 30
order, 27
ordered tree, 34
ordering relation, 27

path, 28
- simple, 28
positional tree, 34
primitive, 52
primitive data type, 52
primitive visual data type, 52
priority queue, 33
probe, 17, 24
- program animation, 15
program visualization, 8
queue, 48

reference, 29
reusability, 37, 46
reuse

black-box, 47
white-box, 47
rooted tree, 34
self-animating programs, 15
simple cycle, 28
simulation, 1, 7
simulation model, 51, 54
software visualization, 8
stack, 48
state event, 71
static tree, 42
storage structure, 37, 39
graph, 43
- linked list, 40
table, 40
tree, 41
table, 40
tailorability, 18
taxonomy, 9
terminal node, 64
TRAKLA, 2
transitive closure, 27
Tred, 2
tree, 41
- $B^+$-tree, 67
binary, 35
binary search, 35
directed rooted, 34
dynamic, 42
ordered, 34
positional, 34
rooted, 34
static, 42

user controlled simulation, 1, 8, 19, 70
virtual array, 38
visual components, 54
visual data type, 53
visual debugging, 1, 19
visual key, 58
visualization, 1, 8, 54

WWW-TRAKLA, 2
Appendix A

Binary Search

The set of basic blocks in linear code, derived from the following binary search function, are illustrated graphically in Figure 5.11 on page 74.

```c
#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;
}
```