Animating Binary Search Trees

Viktor Georgiou

May 2010

Computing Science

Supervisor: Dr Jason Steggles

Word Count: 18,659

Abstract

Binary Search Trees have been developed to make searching easier and more efficient. A range of different types of such trees exists and each individual one has its own algorithms for rebalancing the tree. Many of these algorithms are very complicated and difficult to understand. One way to teach them is to introduce animation, which will graphically visualise each algorithmic step. This project sets out to create a support tool for learning and understanding the algorithms associated with Binary Search Trees using animation. This will allow the users to see how each algorithm runs for a given type of Binary Search Tree. The tool should be easy to use and should provide features to reinforce the pedagogical helpfulness such as pseudo-code, explanation comments for each algorithmic step and stepping the animation backwards.

Declaration

“I declare that this document represents my own work except where otherwise stated”

Viktor Georgiou

Acknowledgments

I would like to thank Dr Jason Steggles for his valuable support and guidance throughout the project. I would also like to thank the students who contributed to this project by answering to the user questionnaire. Finally, I want to thank my parents for giving me the opportunity to study abroad and obtain a Bachelors degree in an area that interests me a great deal.

Table of Contents

1 Introduction

A *Binary Search Tree* [23] is a powerful data storage structure used to make searching efficient. Binary Search Trees are simply Binary Trees [5](i.e. trees where each node has at most two child nodes) such that each node has the following properties: *All nodes descending on the left have smaller data values than node's data and all descendants on its right have larger data values*. Figure 1.1 shows a Binary Search Tree where nodes are represented as circles and edges are represented as lines connecting the circles.

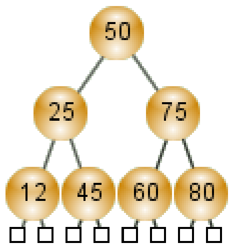


Figure 2.1.1: A Binary Search Tree with root 50

Binary Search Trees work best when they are balanced, i.e. no leaf is much further away from the root than any other leaf [5], gaining a search performance of O(log N) . If they become unbalanced then searching becomes inefficient because the tree acts as a linked list resulting in a worst case performance of complexity O(N).

New variations of Binary Search Trees were developed to ensure that a Binary Search Tree remains balanced. Some of these are *AVL* [12], *Red-Black* [7], *Treaps* [5] and *Splay* [11] trees. These trees are called Balanced Binary Search Trees due to their ability to rebalance (move deep nodes closer to the root) the tree after insertion and deletion. For this reason the worst case performance of O(N) can never occur in this type of trees.

A major problem regarding Binary Search Trees is that their associated algorithms can be very complicated, especially for the Balanced Search Trees. In particular, they are normally defined using pseudo code, which can be difficult to understand and visualize. An effective solution to this problem is the use animation techniques. *Animation* [16] is the time-based alteration of graphical objects through different states, locations, sizes and orientations. More specifically, animation [28] is a set of images (frames) appearing on the screen in a way that users think that something is moving. This technique is much more effective than just reading a pseudo code definition of an algorithm since it allows you to visualise the underlying idea behind an algorithm.

This project sets out to develop a tool that will use animation effects to visualize the algorithms associated with Binary Search Trees while they operate. The idea will be to provide full control over the animation (step forwards and backwards) and in particular, link the animation to the pseudo code in order to help the user understand the algorithm. A range of different types of Binary Search Trees will be selected and included in the tool. In particular, the tool will be designed to allow adding new types of trees to be straightforward.

2 Aims and Objectives

2.1 High level Aim

To develop a tool that enables users to understand how different Binary Search Trees operate by using animation techniques.

2.2 Objectives

1. To identify the key types of Binary Search Trees that should be included in the tool
2. To identify how animation can be used to aid the understanding of Binary Search Trees.
3. To identify the functional requirements of the animation tool
4. To develop a prototype animation tool for Binary Search Trees.
5. To evaluate the effectiveness of the tool

2.3 Description of objectives

Objective 1: A wide range of Binary Search Trees exists so that a subset needs to be chosen for the tool. The idea is to select Binary Search Trees based on their popularity and to ensure that a wide range of technical approaches is included. Then they have to be studied in detail in order to gain a full knowledge of how their associated algorithms for insertion, deletion and searching operate.

Objective 2: To research and study a range of animation techniques and understand how an algorithm can be animated. This knowledge will then be used to design a range of animation features for Binary Search Trees, which will help to highlight the key features of their underlying algorithms.

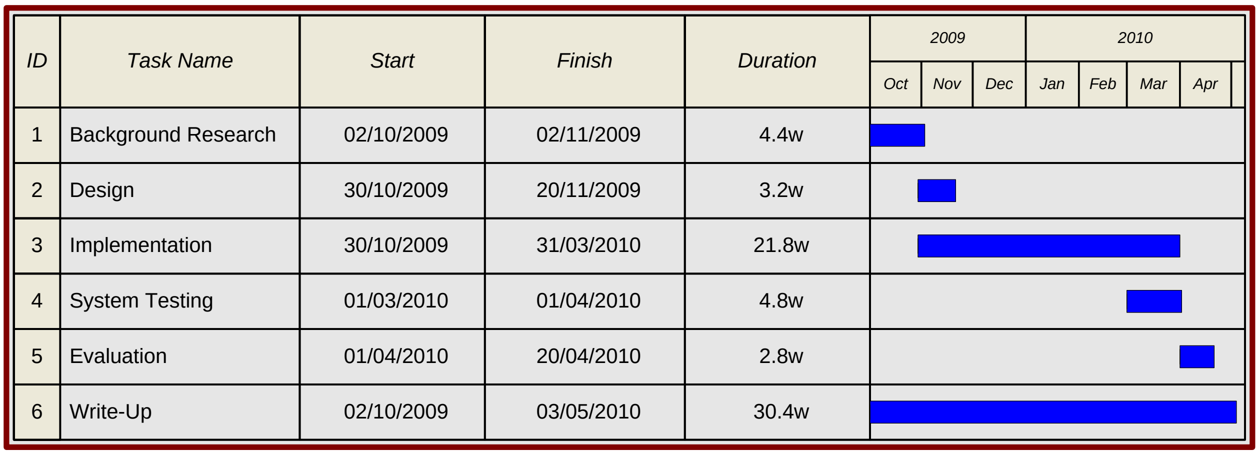
Objective 3: It will be important to decide what features the tool should provide. For example, some important areas include control of the tool; user support and appearance. These features will then be documented in the form of requirements in order to assist the implementation of the tool.

Objective 4: To implement a prototype tool for Binary Search Trees such that it will meet the functional requirements of the tool as identified in Objective 3 above. It will be important here to identify appropriate implementation technologies.

Objective 5: After the tool is developed and tested appropriately, it must be evaluated to gauge how well it supports the learning of Binary Search Trees. This will involve conducting a survey and distributing a questionnaire to an appropriate group of users.

2.4 Project Schedule

The project schedule for the first and second semester with details of the key tasks included is presented in figure 2.4.1 below.



ID	Task Name	Start	Finish	Duration
1	Background Research	02/10/2009	02/11/2009	4.4w
2	Design	30/10/2009	20/11/2009	3.2w
3	Implementation	30/10/2009	31/03/2010	21.8w
4	System Testing	01/03/2010	01/04/2010	4.8w
5	Evaluation	01/04/2010	20/04/2010	2.8w
6	Write-Up	02/10/2009	03/05/2010	30.4w

Figure 2.4.1: Project Schedule

3 Background Research

Background research is a very important part of the project. It is the part where the essential information about the project needs to be gathered in order to drive to its successful completion. Any wrong decisions made or inaccurate material collected might cause serious problems in the future.

3.1 Binary Search Trees

There is a range of different Binary Search Trees. This section will describe some of the most popular kind of Binary Search Trees as Standard, Balanced and Self- Adjusting Binary Search Trees. Both Balanced and Self- Adjusting Binary Search Trees are a refinement of the Standard Binary Search Tree.

3.1.1 Standard Binary Search Tree

A *Binary Search Tree* [22] is a data storage structure used in programming. Such trees are generated by taking elements in a random order and inserting them into an empty tree. It is represented as graph (with nodes and edges) with some unique properties shown below [23].

- Each node has at most two child nodes.
- The top level node is called the root
- The descendant nodes to the left have smaller data values than the node's data value and the descendants to the right have larger data values.

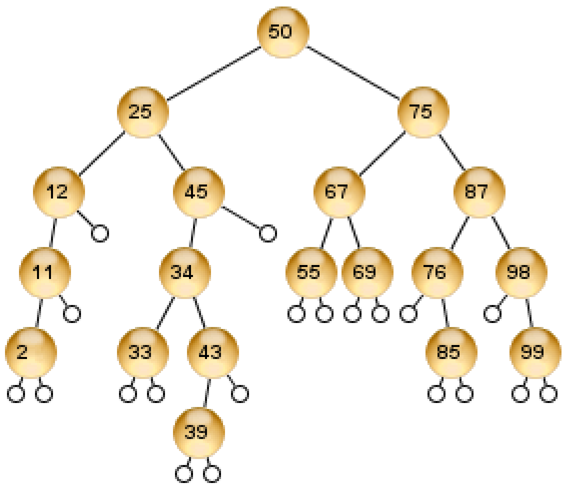


Figure 3.1.1: A Binary Search Tree

Binary Search Tree operations

1. Searching

Searching [23] data in a Binary Search Tree is as easy as in an ordered array. The associated searching algorithm is presented below where `c` initially points to the root of the tree and `key` is the key of the node that we are looking for.

```
Algorithm search
Inputs c:Pointer; key:Data
Returns c: Pointer
Begin
    //if reaches a null node
    If c = NULL then return NULL
    Else
            //if we have found the node
            if c.key() = key then return c; fi
            //if the key is greater than the node's key
            else if key > c.key() then
                    //check right sub-tree
                    return search(c.rightPointer(), key);
            //key is smaller than node's key
            else
                    //check left sub-tree
                    return search(c.leftPointer(), key);
            fi
    fi
End
```

Figure 3.1.2 below illustrates the searching algorithm while is looking for 57.

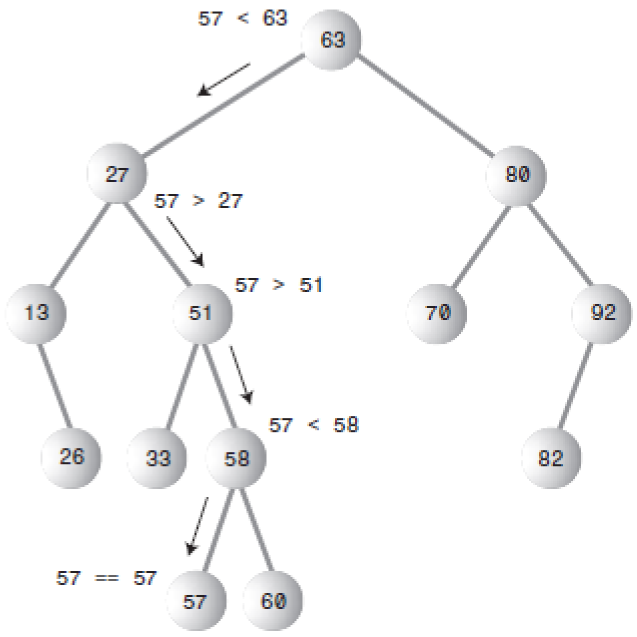


Figure 3.1.2: Searching for 57 [23]

2. Insertion

Insertion [23] in a Binary Search Tree is as quick as in a linked list. The first thing to do when inserting a node is to find the place to insert it. This involves a searching operation, but this time the target node to be found is the parent of the new node (if the node does not already exist). When the parent is found, the new node is connected either on the left of its parent if the new node's data is smaller than its parent is, or on the right side of the parent if the new node's data value is greater than its parent is. Figure 3.1.3 shows how insertion is performed and a pseudo code algorithm for insertion is shown in Appendix A2.

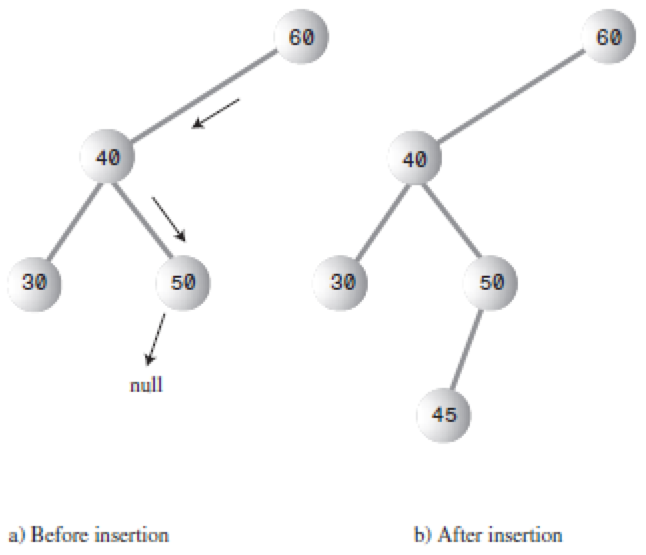


Figure 3.1.3: Inserting 45 in a Binary Search Tree [23]

3. Deletion

Deletion [23] is the most complicated operation in a Binary Search Tree. It first starts by finding the node to be deleted in the tree. If the node is found, there are three cases to be considered.

Case 1: The node to be deleted has no children

This is the simplest case, where the target node is spliced out of the tree and it is replaced with a null node (figure 3.1.4).

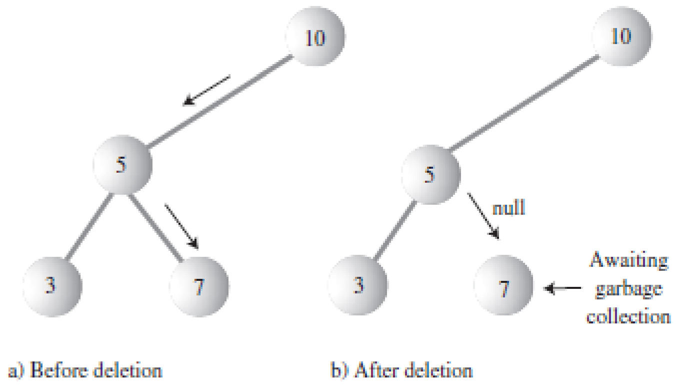


Figure 3.1.4: Deleting a node with no children [23]

Case 2: The node to be deleted has only one child

A node has two connections, to its parent and its child. The node is spliced out by connecting its parent with its child. More specifically, the deleted node is replaced with its child no matter if it is on the left or right side (figure 3.1.5).

Case 3: The node to be deleted has two children

The node to be deleted is replaced with its *successor* as in figure 3.1.8. The successor of a node is the node with the next highest value. It can be found using the following path starting at the node to be deleted: A step right and then as left as it can as in figure 3.1.6. If the right child of the node has no left child then the successor is the right child itself as in figure 3.1.7.

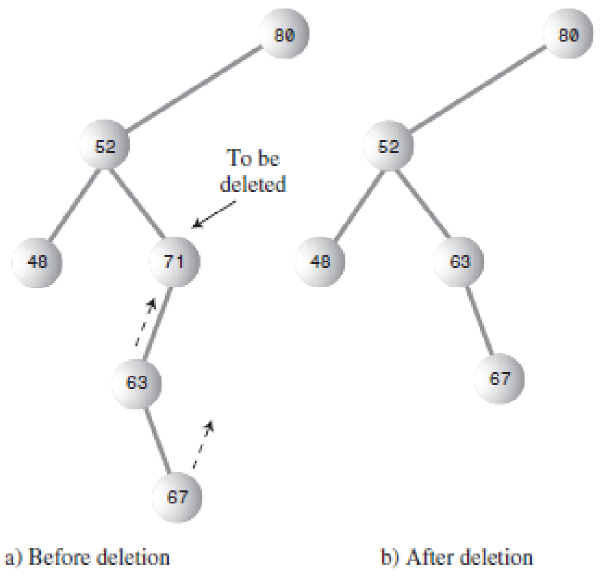


Figure 3.1.5: Deleting node 71 with only one child [23]

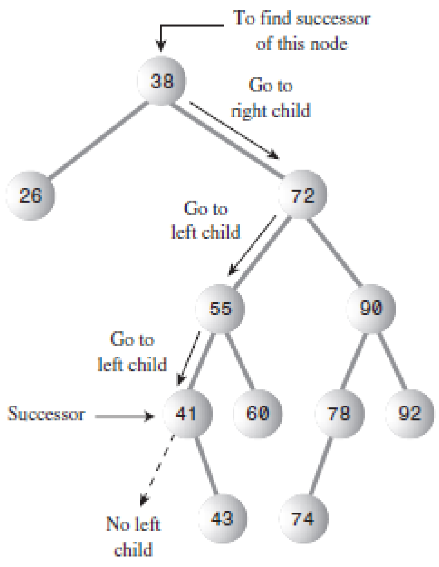


Figure 3.1.6: Finding the successor of 38 [23]

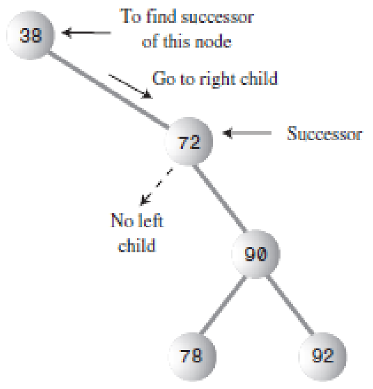


Figure 3.1.7: Finding the successor of 38 where 72 has no left child [23]

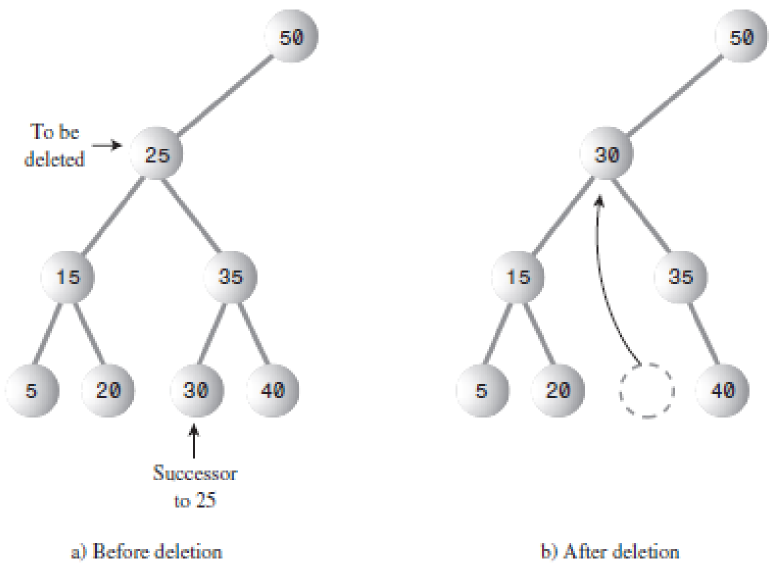


Figure 3.1.8: Deleting a node with two children [23]

The associated pseudo code algorithm for deletion can be found in Appendix A3

3.1.2 Balanced Binary Search Trees

"A *tree* where no *leaf* is much further away from the *root* than any other leaf. Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced."[5]

A basic operation in Balanced trees is *rotations* [11]. There are two types of rotations, left and right rotation which are symmetrical to each other.

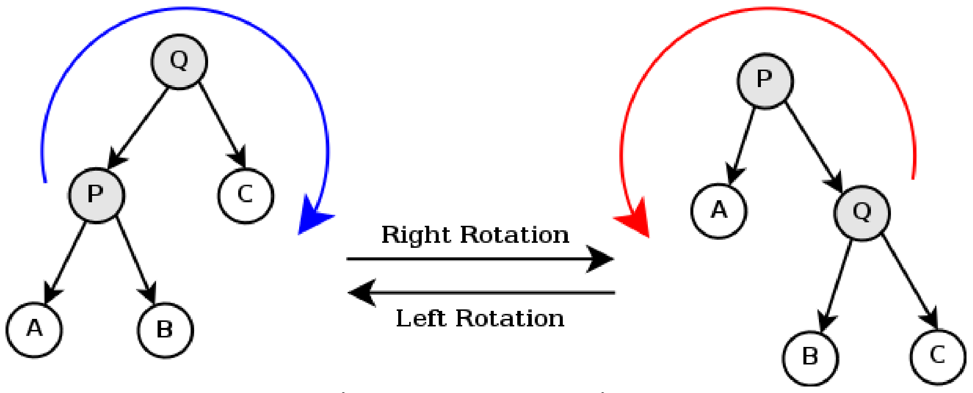


Figure 3.1.9: Tree Rotations

A *right rotation* [11] of a node P over its parent Q is performed according to its grandparent G. If Q is not the root (G is not null) then G becomes the parent of P by replacing Q. The right sub-tree of P becomes the left sub-tree of Q and Q becomes the right sub-tree of P.

In order to return back to the initial state (state before the right rotation) a *left rotation* [11] of the node Q about its parent P needs to be performed. In this case, G becomes the parent of Q by replacing P, as in right rotation. The left sub-tree of Q becomes the right sub-tree of P and P becomes the left sub-tree of Q. Figure 3.1.9 shows how left and right rotations are performed.

1. Height-Balanced trees

Height-Balanced trees [34] are also known as *AVL* trees. Such trees are Binary Search Trees that have the property that any two sub-trees at a common node differ in height by at most one. This balance property can be maintained by counting the difference in height between the right and left sub-trees at each node. Figure 3.1.10 shows an AVL tree where each node is marked with the difference in height between its left and right sub-trees. The algorithm for rebalancing the tree after insertions and deletions in an AVL tree is shown in Appendix A4.

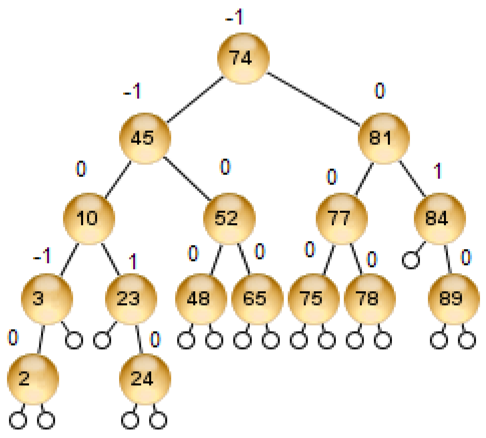


Figure 3.1.10: AVL Tree

2. Red-Black Trees

A Binary Search Tree in which each node is coloured either red or black. As every balanced tree, red black trees have their own rules listed below [6].

a. Each null node (children of each leaf) are known as external nodes and are always marked black
b. The root is always black
c. The red condition: Each red node has a black parent.
d. The black condition: Each path from the root down to a null node must contain the same number of black nodes.

To ensure that the tree meets the rules stated above, a Red-Black tree is rebalanced after insertions and deletions based on the algorithms shown in Appendices A5 and A6. A Red-Black tree is presented in figure 3.1.11

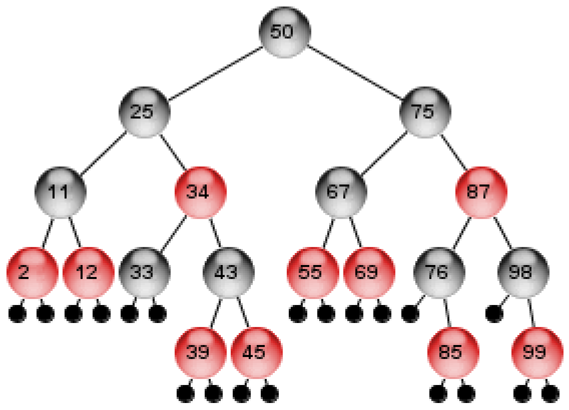


Figure 3.1.11: Red-Black Tree

3.1.3 Self- Adjusting Trees

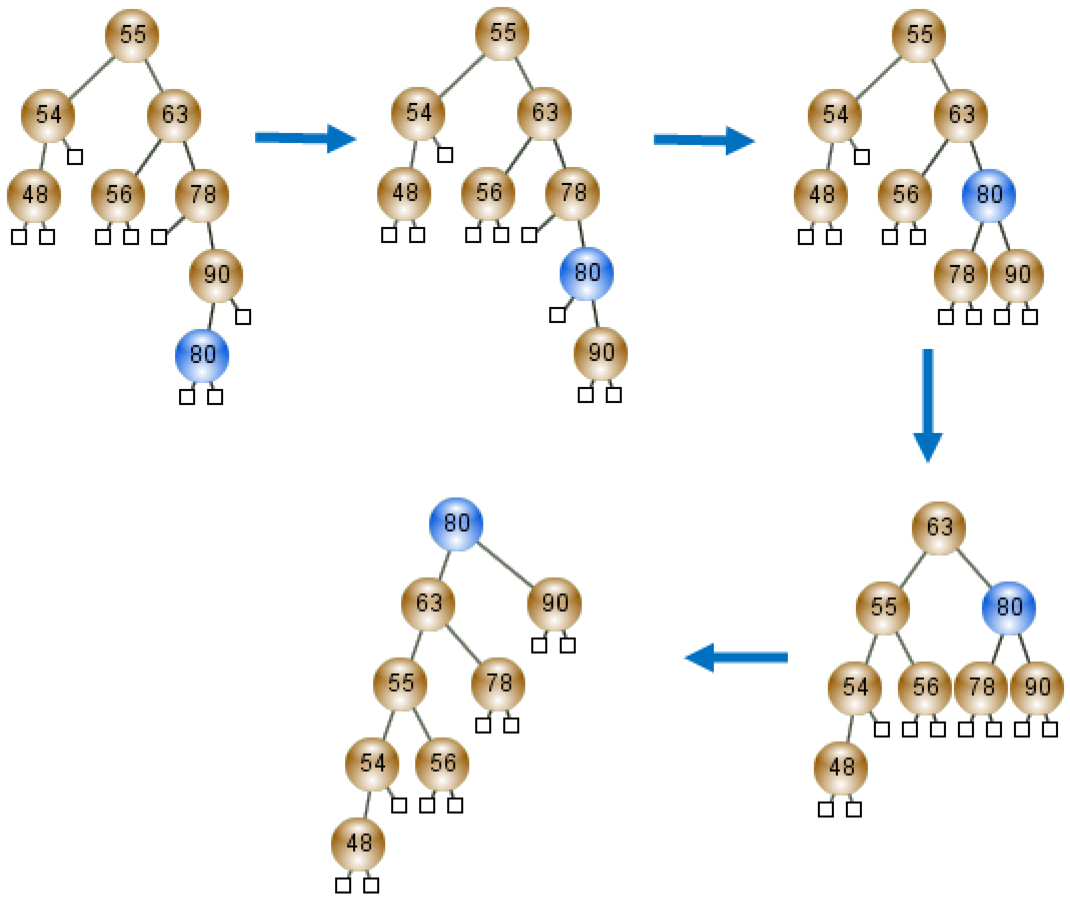


Figure 3.1.12: Splaying 80 to the root of the tree

Self-Adjusting trees are Binary Search Trees, also known as *Splay* trees. The basic approach of *Splay* trees [11] is that not all elements are used with the same frequency. For example if an element is located in a deep level of the tree and is used frequently, then accessing this element will be costlier than if it is located close to the root. Therefore, the strategy of self-adjusting trees is to create a kind of "priority tree" by restructuring the tree each time a node is accessed, by moving it to the root.

A strategy which moves a frequently used element to the root is called *splaying* [11]. This strategy pushes an element to the root by a sequence of left or right rotations. A full pseudo code algorithm for splaying is presented in Appendix A7. Figure 3.1.12 above shows splaying 67 to the root.

3.2 Animation

Animation [28] is a powerful technique that gives life to objects by creating an optical illusion of movement. This approach studies algorithms and programming techniques in order to specify and generate motion of graphical objects. In particular, animation adds time dimension to computer graphics by defining how computer objects change their shape and appearance over time.

3.2.1 How animation works

The basic idea behind animation is to generate a set of images called frames. These frames are then displayed in a rapid sequence in such way that they are perceived by an observer as a single moving image. An animation example taken from *Filthy Rich Clients* [16] is shown in figure 3.2.1 below. It shows a fly's journey as a set of individual images which can be shown in rapid sequence and convince our minds that the fly is moving.

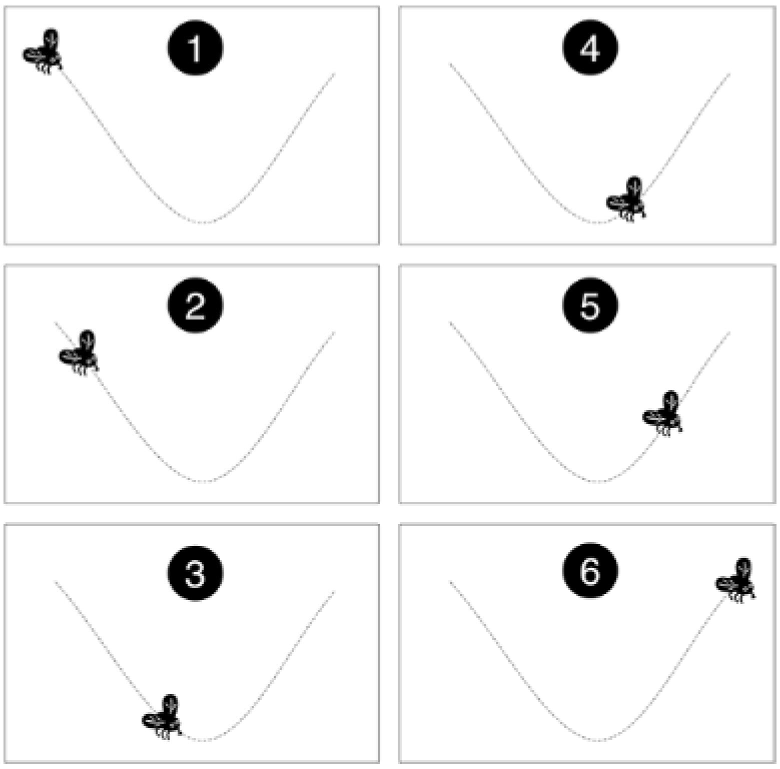


Figure 3.2.1: Individual animation frames for the fly's movement

3.2.2 Algorithm Animation

Algorithm Animation [8] is used for teaching purposes in computer science courses. It is a technique used to explore an algorithm's behaviour in order to bring it to life.

When an algorithm is executed, it leads to a sequence of states, each of them containing abstractions of data, operations and semantics. Each step of an algorithm causes the current state to be transited to the next one. The basic idea of algorithm animation is to map each of these states into a visual representation (view), often visualised as an image, and shows the transitions as animations between these views.

The Figure 3.2.2 shows how each algorithm state is mapped into an image, allowing transitions between these images to produce the algorithm animations.

Sometimes it is more appropriate to map states into visual models represented as graphical objects, which are then rendered to produce an image for the current state. This approach has the advantage that allows each graphical object to render itself in an arbitrary manner, based on programmer's preferences. As a result, algorithm states are not restricted to map into a single image. Figure 3.2.3 shows the mapping of algorithm states into visual models.

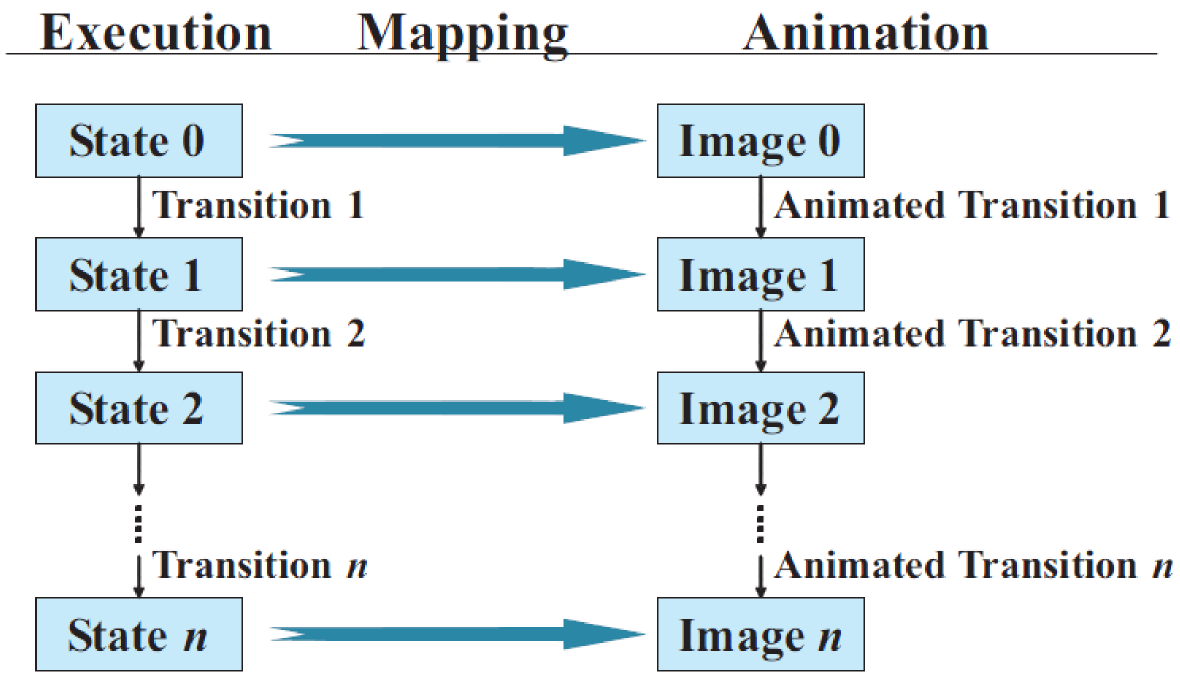


Figure 3.2.2: Algorithm animation: mapping states to images [8]

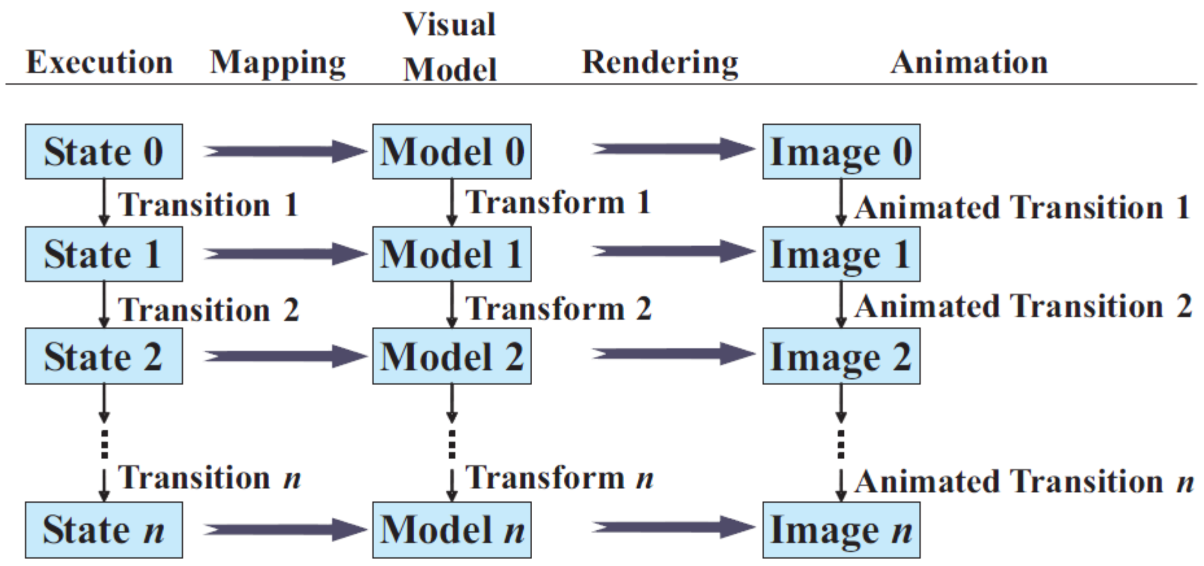


Figure 3.2.3: Algorithm animation: mapping states to models [8]

3.2.3 What makes a good animation

Often, animations are used to assist the learning of an algorithm. To achieve this they have to provide sufficient information to users and draw their interest. What makes a good animation depends on the features included to the animation. Following, is a discussion about some of such features the developers need to be aware in order to create algorithm animations that aid the comprehension.

Animation control

Some algorithms are complex and involve many rearrangements when stepping from one state to another [26]. For example, a Red-Black tree rebalance operation involves many colour flips and rotations. In such case, users get confused and might want to pause or slow down the animation speed. Even when slowing down the animation or pausing it, users might miss out some important information. For this reason, animations often need to provide a rewind facility or stepping backwards. The advantage of such feature is that users will be able to examine particular events of an algorithm by stepping backwards and forwards.

Smooth animation

Smooth animation is one of the fundamentals of algorithm animation. When an algorithm steps from one state to another, it is easier to see the change if it appears as a smooth transition, rather than an abrupt erase and repaint [18]. Smooth motions are very important and need to be taken into consideration when constructing animation tools.

Highlight areas of interest

Regularly, animations require focusing the attention on a specific area of interest where a particular event takes place. Therefore, they paint a specific region using a different colour to draw the eye to them. AVL trees for example, need to highlight an inserted node, which requires several rotations. Rudolf [30] mentions that this feature can also be used to link the animation to the pseudo code by highlighting the currently executed piece of code for education purposes. The highlighting process should be temporal and quickly removed after the event of interest ends otherwise it can become distractive. In addition, the choice of the colour to highlight is very important and needs to consider disabilities such as colour blindness.

Sound

The use of sound as part of the animation effects plays a vital role for a good animation as it conveys information that is difficult to visualize. In particular, sometimes it is hard to associate an event with a particular animation effect. Therefore, the use of a sound could make the task easier and probably more appropriate for some events. In addition, sound is sometimes used to reinforce what is currently being displayed visually, making the animation even more exciting [18].

3.3 Implementation Technology

A crucial decision that needs to be made before the tool's development is the choice of a programming language for developing the tool. A huge range of different programming languages exists. For this reason, we will focus on the most popular ones that can be used to create an animation tool, taking into consideration factors such as visualization features, portability and extendibility. Based on these factors, the following languages are selected and analysed in order to choose the most appropriate one.

3.3.1 Adobe Flash

Adobe Flash [2] was previously known as *Macromedia Flash* [25] and is a multimedia platform used to create animations, videos and rich internet applications that are integrated into a webpage. Adobe Flash is not a programming language but it provides a scripting language called *ActionScript* that is primarily used to develop strong and dynamic websites and software using the Adobe Flash platform. The advantages with such platform are that it is portable due to its integration into web pages and provides a good animation support easy to use. However, programs developed in Adobe Flash platform require the Adobe Flash player to be installed, making them difficult to be extended. In addition, it may require much processing power and a good graphics card in order to perform well and run smoothly.

3.3.2 C/C++

C [21] is a powerful functional programming language developed in 1972 [20] by Dennis Ritchie and is used in many computing environments such as UNIX, PC and MAC. C was designed to provide direct access to memory and requires negligible run-time support, making it ideal for applications that were previously coded in Assembly language.

In 1983, C was extended to C++, a higher-level language that introduced object oriented features. C++ is one of the most popular programming languages and is widely used in the software industry. It provides much support for graphics and animation features and is one of the fastest languages during run-time that exploits as much as possible the usage of memory, allowing more algorithms to be added to the tool without slowing down the system. Although, it requires careful control over the memory as the developer is responsible to free the allocated memory, making the coding harder and more complicated than other languages. In addition, C++ is not much portable due to its compilation into machine code which can only be executed in particular platforms [31].

3.3.3 C#

C# [33] is a modern object-oriented programming language that has its roots in the C family of programming languages. It was designed for the Common Language Infrastructure, a specification that defines an environment that allows multiple high-level languages to be used on different computer platforms running Microsoft's .NET Framework. C# syntax is simpler than C++ allowing developers to debug a program easily. It is also the best programming language available for .Net Framework and is easily extendable. However, C# was designed for Microsoft Windows environments and there might be issues regarding the portability with other platforms as Mac and UNIX.

3.3.4 Java

Java was developed by James Gosling in 1995 at Sun Microsystems and is a powerful strongly typed programming language that follows the object-oriented paradigm [15]. As in C#, Java can run on any platform that runs a Java Virtual Machine (counterpart of Microsoft's .Net Framework) regardless of the computer architecture (platform independent). An advantage that Java has over many other languages is that it provides comprehensive Application Programming Interface (API) allowing developers to reuse or even extend existing classes of the API making their job easier and simpler. Many graphical classes as Swing and Graphics2D exist that allow the development of robust graphical user interfaces and animation tools using a range of IDEs as Eclipse, NetBeans and BlueJ. Java is considered as one of the most influential programming languages of the 20th century, and it is widely used in the Software Engineering industry including many universities in UK. A major drawback regarding the implementation in Java is that programs run slower than in other languages such as C/C++ [4].

3.3.5 Conclusion

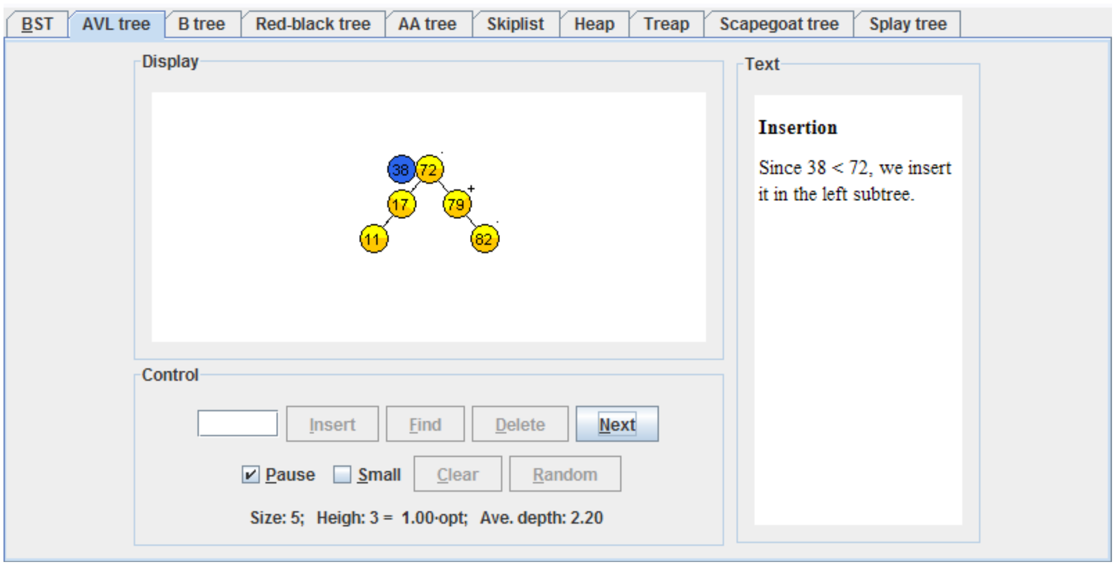
Adobe Flash provides a strong animation environment using scripting languages but it requires much processing power and it may not be extendable enough. On the other hand, C/C++ is the fastest language during run-time providing much support for graphics and animation. However, it is a very complicated language and it might not run normally on some platforms. Finally, C# and Java are platform independent languages, both following the object-oriented paradigm and allow programs to be extendable and simpler. They also provide a rich graphics environment that can be used for the implementation of this tool. On the other hand, C# has some issues regarding the portability to other platforms. Alternatively, Java meets all the consideration factors stated above and as it has already been taught in depth during the last three years in the University, it is an ideal language for the development of this tool.

3.4 Existing Systems

A range of existing tools is selected and evaluated based on the requirements of the project and mainly on the ability to aid the understanding of Binary Search Trees.

1. Binary Search Trees Applet

This tool [1] visualizes a numerous data structures including Binary Search Trees as AVL, Red-Black, Splay, Treap etc. It provides the basic operations of Binary Search Trees and describes each algorithmic step. A screenshot of this tool is shown in fire 3.4.1.



Insertion

Since 38 < 72, we insert it in the left subtree.

Size: 5; Heigh: 3 = 1.00-opt; Ave. depth: 2.20

Figure 3.4.1: Screenshot of the Binary Search Trees applet

2. Java Models: Binary Search Trees Applet

This applet [19] demonstrates the operations of AVL, Red-Black and Splay Binary Search Trees. It provides some extra operations such as finding the minimum key, deleting all the nodes and traversing the tree using different traversing algorithms. Figure 3.4.2 shows a screenshot of this applet.

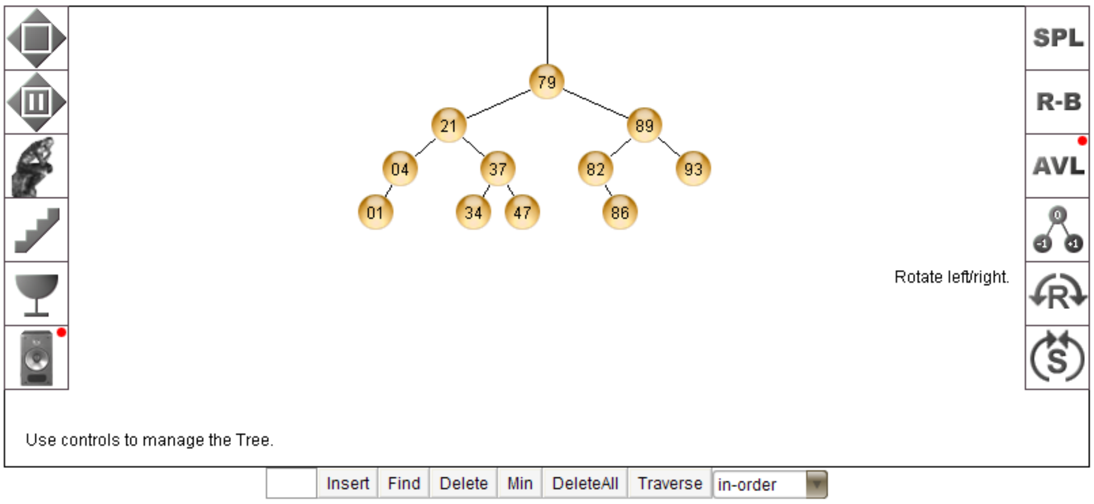
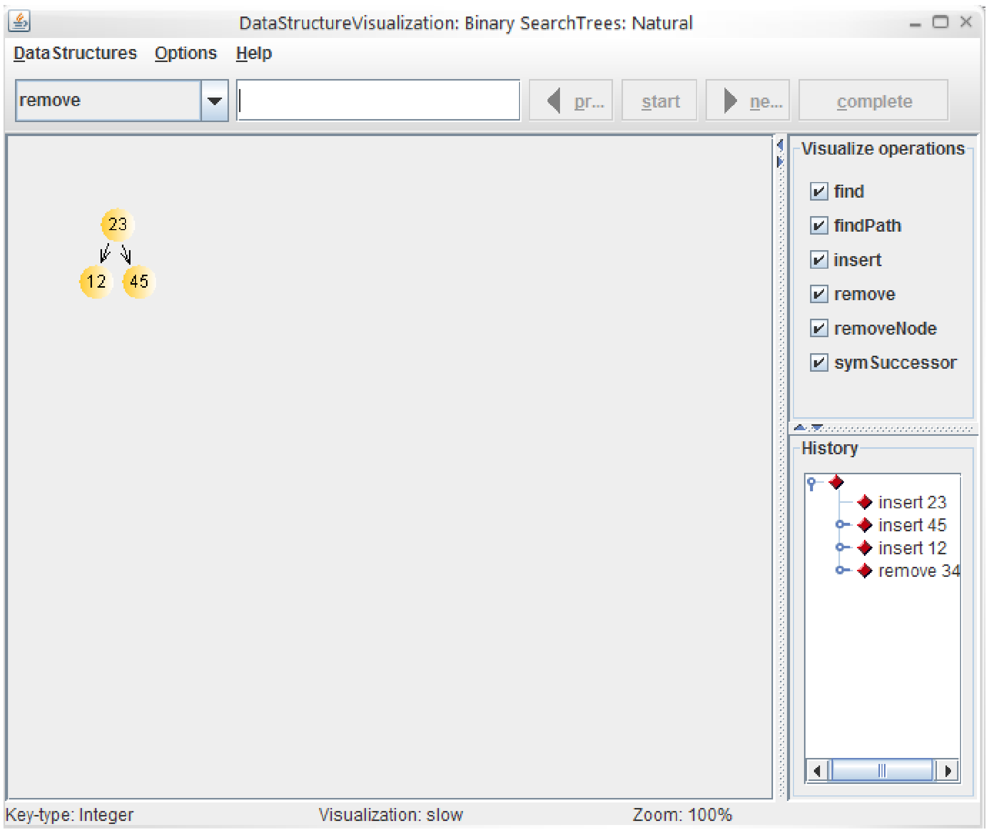


Figure 3.4.2: Screenshot of the Java Models Binary Search Trees applet

3. Data Structures Navigator (DSN)

A tool [9] demonstrating a range of sorting algorithms, Lists and Binary Search Trees. A new feature provided by this tool is that it records the operations that took place and stores them in a history panel.

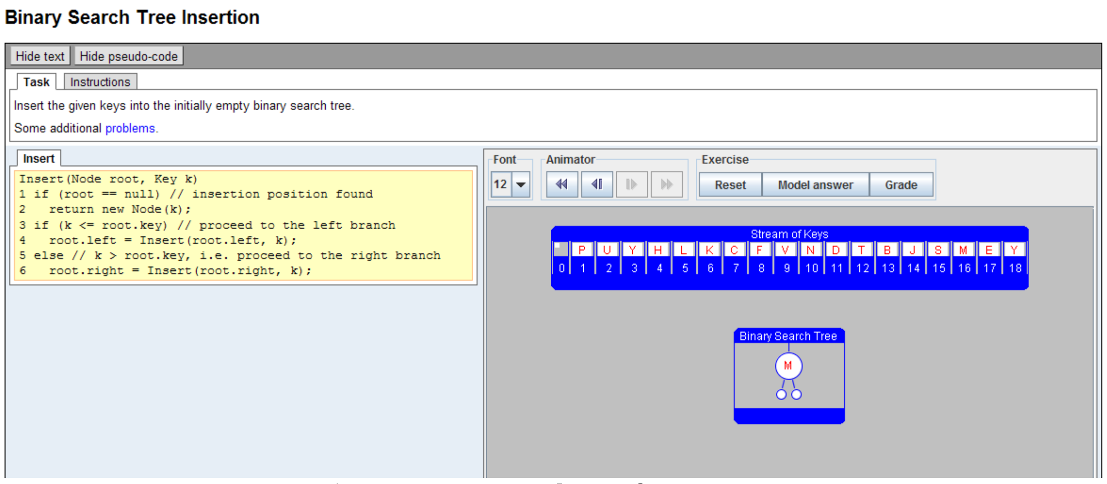


Key-type: Integer   Visualization: slow   Zoom: 100%

Figure 3.4.3: Screenshot of the Data Structures Visualization tool

4. TRAKLA 2

TRAKLA 2 [17] is a very well structured animation tool supporting almost any algorithm and data structure. It was developed to assist the understanding of these algorithms and provides exercises to be solved by the user. Screenshots of TRAKLA 2 are presented in figures 3.4.4 and 3.4.5 below.



Insert the given keys into the initially empty binary search tree.

Some additional problems.

```
Insert(Node root, Key k)
if (root == null) // insertion position found
  return new Node(k);
if (k <= root.key) // proceed to the left branch
  root.left = Insert(root.left, k);
else // k > root.key, i.e. proceed to the right branch
  root.right = Insert(root.right, k);
```

Figure 3.4.4: Screenshot 1 of TRAKLA 2

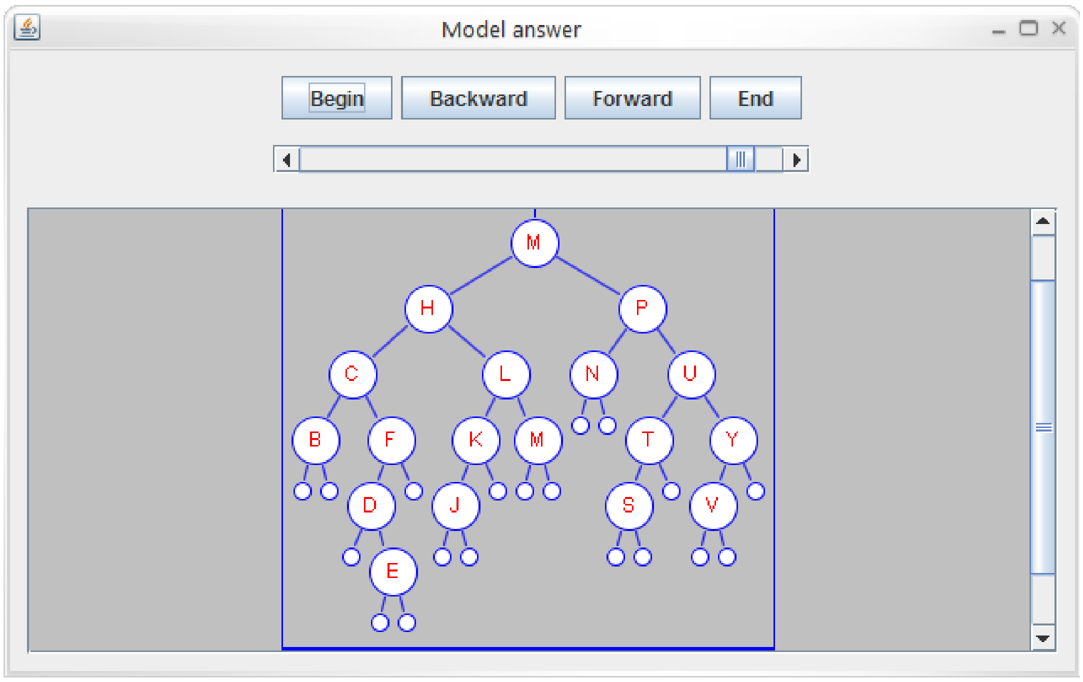


Figure 3.4.5: Screenshot 2 of TRAKLA 2

3.4.1 Evaluation of existing Systems

Below is a table that evaluates the existing tools using eight criteria, considering in detail their strengths and weaknesses.

Tool	**1**	**2**	**3**	**4**
Range of animation effects	√	√	√	X
User friendly graphical user interface	√	√	√	√
Structure can be modified	X	√	X	√
Description of algorithm steps	√	X	X	√
Pseudo code linked to the animation	X	X	X	√
History panel	X	X	√	X
Step forwards	√	X	√	√
Step backwards	X	X	X	√

Table 1: Evaluation of Existing Systems

3.4.2 Conclusion

The existing systems evaluated in table 1 are good animation tools for Binary Search Trees. They have user-friendly graphical user interfaces and provide a range of animation effects. In particular, the animation effects of the second tool were very interesting and we intend to include similar effects in this project's tool. A major drawback for some of these is that they do not provide a link to the pseudo code. Since understanding Binary Search Trees algorithms is the aim for this project, the tool must provide such a feature. In addition, only TRAKLA 2 provides full control over the animation, including backward steps. This evaluation has provided some important insights, which will be taken into consideration for the development of this tool.

3.5 Human-Computer Interaction

3.5.1 Overview

Human computer Interaction [32] is the study of the interaction between humans and computers. The user initiates the interaction by giving to the computer some commands, which often require some output. The response is typically displayed on the screen and it must provide adequate information to the user. This interaction continues until user receives the desired output. During the interaction, the user must identify the displayed information and select the appropriate response. Therefore, the interface between user and computer must be designed according to user's processing capabilities.

3.5.2 User Interface Design Principles

A user interface has to follow some principles of human perception and information processing in order to create an effective display design. Such principles are based on usability guidelines. Following, there are some of the most important principles for a usable interface design taken from [10].

1. Learnability

The user must be easily able to begin an effective interaction with the system and achieve maximal performance.

Predictability: The interface should be able to predict future actions based on the history of previous interactions and provide the user with visible potential operations

Synthesizability: The user should be able to consider the consequences of previous operations on the current state

Familiarity: The user interface must be designed in accordance to the knowledge the users already have. The more familiar the user is with the system, the easier is to effectively start interacting with the interface and predict similar capabilities

Generalizability: The interactive system must support the user for extending his/hers knowledge of specific interaction to other similar situations that did not come across

Consistency: Likeness in input/output behaviour arising from similar situations or task objectives

2. Flexibility

The interface should allow the user to manipulate the system easily in multiple ways in order to be able to exchange information.

Dialogue initiative: The system must allow the user to interact with it by granting him a full access, without imposing any constraints by the input dialogue, for example, when login in to the system

Multithreading: The interface must support the user to interact with more than one task at the same time

Task migratability: The user and the system must be able to pass the control of a task execution to each other. Therefore, a task that is internal to the user or the system can be either internal to the other part, or even be shared between both of them

Substitutivity: Equivalent values of input/output forms must be allowed to substitute. For example, a form for adjusting the margins of a letter must allow the user to enter the value in inches or centimetres, or even to explicitly define it without using the form choices

Customizability: The user should be able to modify the user interface. For example, the user could change the input/output form to display only centimetres

3. Robustness

The interface must provide the user with a maximal support level, allowing him/her to assess whether the current task was successfully performed.

Observability: The user must be able to evaluate the internal state of the system by observing the representation displayed on the interface

Recoverability: The interface must allow the user to achieve its goal even when a mistake has occurred. This involves corrective actions that take place once an error has been recognized. Forwards and backwards recoveries are two different types that can be used to resolve an error, and change an erroneous state to an error free one

Responsiveness: The user must always be informed about the status by receiving some feedback from the system. In particular, the feedback must be received within a reasonable time. In cases where an immediate response cannot be obtained due to a long task that is in execution, the user must receive an indication (i.e. progress bar) that the task is being executed

3.6 Evaluation Techniques

A range of techniques exist that can be used to evaluate the overall performance of the tool and assess whether it has met the initial objectives and requirements. Some of the most important evaluation techniques are listed below.

3.6.1 Survey/Questionnaires

A questionnaire is a set of questions related to a specific subject and it is distributed to a set of users. It is used for gathering information based on the responses obtained. The advantages of survey/questionnaires is that they are cheap and do not require much effort to be built. They can also be distributed over the Internet making respondents work easier. However, questionnaires are limited because not every respondent is able to read the questions and reply, and in some cases this might be impractical [3].

3.6.2 Interview

Interview is a technique where the interviewer is having a conversation with one or more people and asks questions related to a subject in order to get some information from them. Interview is considered one of the most useful evaluation techniques as it allows the interviewer to ask a variety of questions and receive immediate responses. However, in some cases interview might be time consuming.

3.6.3 Silent Observation

The evaluator silently observes the users while they interact with the tool and records information on how easy the tool is to use and how long a task takes to complete. A main problem of this approach is that the evaluator does have an insight into the user's decision process or attitude.

3.6.4 Think Aloud Method

Similar to silent observation technique with the difference that the evaluator gives a deep insight into what the user is about to do. However, such technique may alter the way the user is performing a task or it might be hard to talk to him/her while is concentrating on the task.

3.6.5 Retrospective Testing

The evaluator tests the users understanding after they have experienced with the tool in order to clarify events that occurred during the tool's usage. This can be done by performing observational tests, but such thing might be time consuming.

4 System Design

This chapter will describe the process of analysing the requirements of the system in order to produce design solutions for the problem. Such solutions will be taken into consideration for the implementation of the tool, which will be discussed, in the next chapter.

4.1 Requirements

The requirements identified for the project are listed below:

1. The tool should provide animations for four types of Binary Search Trees.
2. It should animate the key algorithms including searching, insertion and deletion
3. A pseudo-code should be displayed for each algorithm.
4. The tool should provide clear animation based on a range of animation effects
5. It should provide smooth transitions from one algorithm state to another.
6. The animation should be linked to the pseudo-code.
7. Explanation comments should be displayed for each step of the algorithm during the animation.
8. It should provide full control over the animation including step backwards.
9. It should provide a friendly graphical user interface.
10. It should allow two or more Binary Search Trees to be compared.
11. It should be extendable, allowing new Binary Search Trees to be easily added.
12. It should provide a description of each Binary Search Tree.

4.2 High-level design

In order to start designing the system we need to identify its major components and their connectivity and produce a high-level design. A key requirement for the system is to provide full control over the animation. Thus, the design must separate the algorithm logic from the animation logic. This will allow the animation to be independent from the algorithm. To accomplish this, some information for each algorithmic state must be recorded during an algorithm's execution. The recorded information should then be used to produce a set of animation states. Next, the animation will act as a player that executes the animation states and produce the required visual effects on the screen. Figure 4.2.1 shows the high-level system design indicating the system's identified main components.

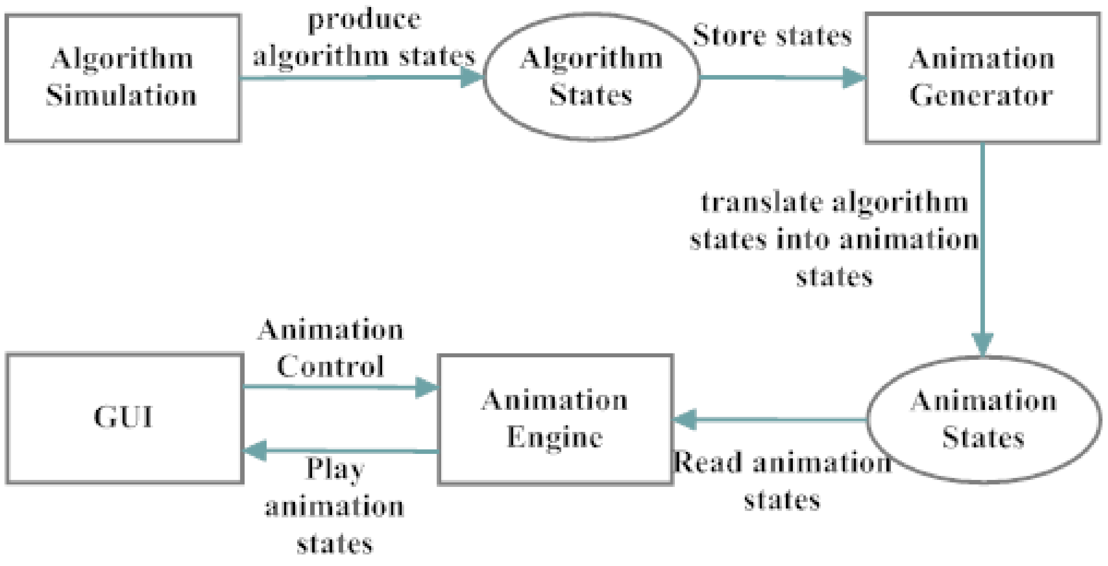


Figure 4.2.1: High level system design

A brief description of the high-level components is now given below.

Algorithm Simulation: It represents the algorithm that is being executed. During its execution, it records state information for each step and produces the algorithm states

Algorithm states: The states produced by the Algorithm Simulation during its execution. Each state contains sufficient information about the data needed for the visualization

Animation Generator: An engine that receives the produced algorithm states and translates them into animation states

Animation states: The states produced by the Animation Generator. Each animation state is a visual model that is transformed into an image displayed on the screen

Animation Engine: The animation player takes each animation state produced by the Animation Generator and transforms it into an image that should be displayed on the screen

GUI: Displays the images produced during the transformation of the animation states by the Animation Engine to the user. It also allows the user to interact with the application and have full control over the animation

4.3 Design of the Algorithm Simulation

A key design decision has been made that an algorithm's execution will be recorded as a series of states. These states are received by the Animation Generator that converts them into animation states. Therefore, each algorithmic state must contain sufficient information about the data that will be modified during each state.

The set of algorithm states that need to be produced by an algorithm depends on the type of tree and operation that is currently executed. Each tree may produce its own states depending on the currently executed operation. First, we identify the basic operations that are common to all the Binary Search Tree types that will be included to the tool. Such operations are searching, insertion, deletion and traversals that are operations of a Binary Search Tree inherited by both Balanced and Self-adjusting trees. The tool should assist the understanding of Binary Search Trees, thus, a pseudo-code linked to the animation as well as explanation comments must be essential during an algorithm's execution. The design of these features will require some additional information to be stored in an algorithm state when an algorithm is executed. The following is the data information that needs to be recorded for each identified algorithm state.

1. Searching state

During searching, the algorithm compares each node that is passed through until either it finds the node that is looking for or it reaches the end of the tree (described previously in section 3.1.1). For this reason, each searching state will be designed as a pair of node data described below:

Parent: The node that is currently compared with the key
Child: The next node to be compared if the parent's key does not match

2. Insertion state

The insertion state will first perform a searching operation in order to find the parent of the node to be inserted (If the node does not already exist). As a result, the data to be recorded for an insertion state is as follows.

Parent: The parent of the node that will be inserted to the tree
New node: The node that is to be inserted
New node index: The position the new node will be inserted (This is either on the left or the right of the Parent).

3. Deletion state

Similarly, deletion requires a searching operation to find the node that will be deleted. In addition to that, deletion might require an additional state for swapping the node that is to be deleted with its successor in the case when it has two children. The data for a swap state are shown below.

Node: The node to be deleted
Successor: The successor of the node to be deleted

After swapping is finished, the node to be removed will have at most one child and can be safety removed from the tree. The data to be recorded for this state are listed below.

Node: The node to be removed
Node Index: The index of the node to be removed (indicates either a left or a right child)
Parent: The parent of the node to be removed
New Child: The node that will replace the removed node
New Child Index: The index of the new child (either left or right)

4. Balanced and Self-Adjusting tree states

Balanced and Self-Adjusting trees have the ability to rebalance the tree after searching, insertion, and deletion. A basic operation that is common to this type of trees and results to the tree being balanced is tree rotations (discussed in chapter 3.1.2). In addition to tree rotations, Red-Black trees involve colour flips in order to rebalance the tree. A list of the identified algorithm states for these trees is shown below:

4.1 Rotation state

In this state, a node should be rotated over its parent. The data that should be recorded for a rotation state is given below:

Node: The node to be rotated over its parent
Rotation type: Indicates a left or right rotation of the node over its parent

4.1 Colour Flip state

A colour flip changes the colour of a Red-Black node from red to black and vice versa. The recorded data is:

Node: The node that its colour will change
Old Colour: The current colour of the node
New Colour: The new colour of the node

5. Highlighting and Explanation state

In addition, there should be a state for highlighting the current line in the pseudo-code that is being executed and an English explanation about that line. The recorded data information for this state is as follows.

Line: The line index on the pseudo-code that will be highlighted
Comments: The explanation comments to be displayed during the execution of that line of code

4.4 Design of the Animation Generator

The Animation Generator is responsible for receiving the data information for the algorithm states produced by the Algorithm Simulation and translating them into visual models called the animation states. The states produced will then be passed to the animation engine that will generate the essential graphical effects on the screen.

Since the tool is concerned with algorithm animation, we will have to provide control over the animation. The Animation Generator must use the algorithm states in an appropriate manner such that the animation states produced can be easily undone and redone. When the animation state is undone, it should go back to the pervious state and when it is redone, it should go to this state. The key idea for the design of this feature is to introduce two primitive methods that will be exported by each animation state. We call these operations `undo,` and `redo` and their functionality is described in table 2.

Operation	**Functionality**
`Redo`	Moves the animation from the current state to this state
`Undo`	Moves the animation from this state to the previous one

Table 2: The primitive operations for each algorithm state

Essentially, an undo operation of an animation state Si will reverse it to the previous animation state Si-1. In state Si-1, we can go to the next state Si by performing a redo operation on Si.

The idea behind the design of the animation states is that each primitive operation of an animation state will execute a smooth transition from one animation state to another as well as the reverse of this transition. This is done by invoking the `redo` and `undo` operations. Based on the received algorithm state information the following animation states have been identified. The design of these states is concerned with the graphical effects that each of these performs when its `redo` and `undo` operations are invoked.

1. Searching State

When searching, a search node should start at the root moving downwards until either reaches the node that is searched for or a leaf node that does not matches the key(indicates that the node does not exists in the tree). The idea is to record all the nodes in the searching path and create a set of node pairs from the visited nodes. Each pair should represent an individual searching state showing the node that is currently visited because of a transition from a parent node to its child. Initially, the search node should be displayed on the root of the tree before the searching starts. The following animation states are the states produced during searching and the effects of their undo and redo operations.

1.1 Search Node state

A search node should be transparent coloured with a different colour than the nodes of the tree such that when is passing through the nodes, the key must be visible to the user. The redo and undo operations of this state are shown below.

Redo: Displays the search node at the root of the tree where the searching starts.
Undo: Removes the search node from the display

1.2 Searching State

During searching, a set of states of this type should be produced. The undo and redo operations for each of these states is given below.

Redo: Animates the search node moving from the parent to one of its children. Figure 4.4.1 shows how this operation is visualized
Undo: Moves the search node from the child back to its parent (exactly the opposite of redo)

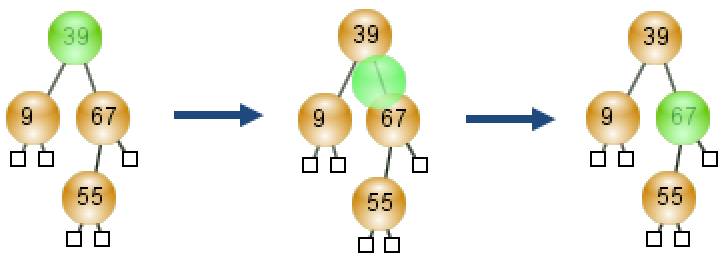


Figure 4.4.1: A searching state that moves the search node from node 39 to node 67

2. Insertion State

The key idea for this state is to make use of the searching operation before inserting a node into the tree. An unsuccessful searching for the node to be inserted will stop at a leaf node. Apparently, this node will be the parent of the node to be inserted, as the node we want to insert does not exist in the tree. Having found the parent of the node to be inserted an insertion can be performed according to the node's key. The effects of the redo and undo operations of an insertion state are shown below.

Redo: Inserts the node to the tree by animating the node being moved from its parent to its final position as in figure 4.4.2
Undo: Removes the inserted node from the tree

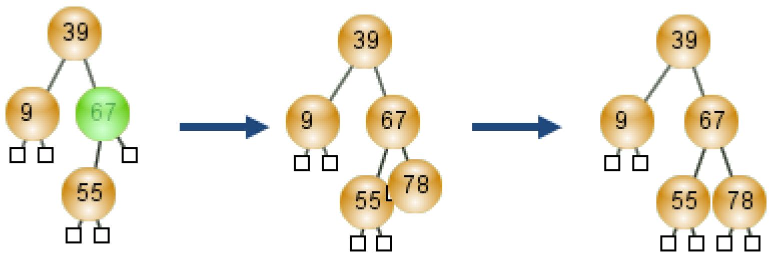


Figure 4.4.2: Inserting node 78 into the tree

3. Deletion state

Similar to insertion, the searching operation can be used to find the node that has to be deleted. If the searching is successful, it will find that node so there are two cases to consider. The first case is when the node has two children where the node to be removed must be swapped with its successor. Next, a deletion is performed, as the node will have at most one child, which is the second case of the deletion. Therefore, deletion will produce the following animation states.

3.1 Swap State

An animation state that shows the keys of a node and its successor being swapped

Redo: The node to be removed being swapped with its successor
Undo: Swaps the successor with the node to be removed (same as Redo)

3.1 Deletion State

Shows the tree after the node has been deleted

Redo: Animates the node being spliced out and falling down the tree. Figure 4.4.3 shows how deletion would be performed
Undo: Inserts the removed node back to its previous position

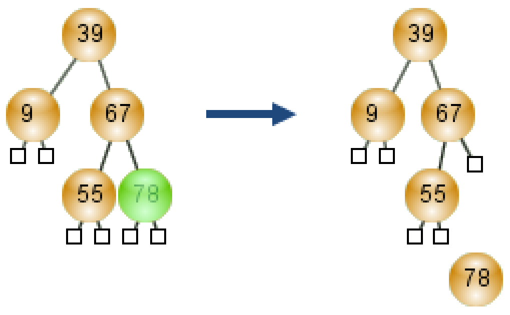


Figure 4.4.3: Removing node 78 from the tree

4. Balanced and Self-Adjusting tree states

The design of the primitive operations for the animation states of Balanced and Self-Adjusting trees is described next.

4.1 Rotation State

Shows the Balanced tree after a node has been rotated either left or right.

Redo: Animates a node being rotated left/ right over its parent (see section 3.1.2)
Undo: Rotates the parent right/left over the node (reverse rotation over the node)

4.1 Colour Flip State

Shows a Red-Black node having its colour changed

Redo: Changes a Red-Black's tree node colour from red to black or vice versa
Undo: Colours the node with its old colour

5. Highlight and Explanation state

A highlight and explanation state should display a highlighted line of code in the pseudo code and display explanation comments for that line. Below is the design of the primitive operations for this state.

Redo: Highlights the line of code that is being executed as well as to display explanation comments for that line of code
Undo: Highlights the previous line of code that was executed and replaces the current explanation comments with the comments associated with the previous line

4.5 Design of the Animation Engine

Once the animation states are produced by the Animation Generator, they need to be passed to the Animation Engine, which will execute the primitive operations on these states accordingly. The idea is to have a *back* stack for storing the states that will be undone and a *next* stack storing the states to be redone. Initially, the *back* stack should be empty with all the states stored in the *next* stack. Each time a state in the next *stack* is redone it is pushed onto the top of the *back* stack that always contains the current animation state on its top. To go back to the previous state we pop the animation state on the top of the *back* stack, execute *undo* and then push it onto the *next* stack. Figure 4.5.1 demonstrates how the animation engine performs a transition from the initial state, to state1 and vice versa.

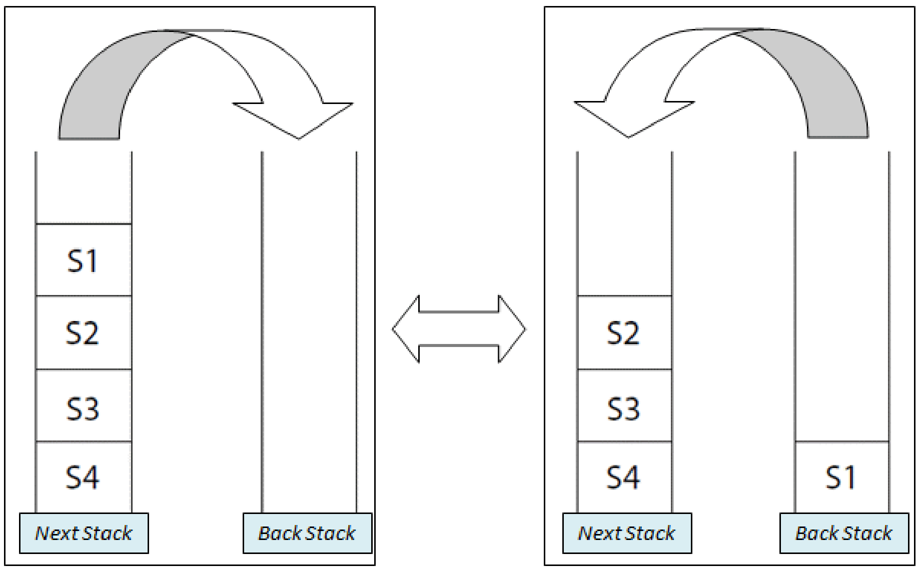


Figure 4.5.1: Going from the initial state to state 1 and vice versa

The initial state is when the *back* stack is empty. Going from the initial state to state1 requires a redo operation on state1 that is currently on the top of the *next* stack. State1 is then pushed onto the top of the *back* stack. Going back from state1 to the initial state requires an undo operation on state1 that is now on the top of the *back* stack. After executing the undo operation, state1 is pushed back onto the top of the *next* stack.

In general, going from state(n) to state(n+1) we execute a redo operation on state(n+1) which is then pushed onto the top of the *back* stack. To go back to state(n), we execute an undo operation on state(n+1) and push state(n+1) onto the top of the *next* stack. Figure 4.5.2 indicates a similar operation that results to a transition from state1 to state2 and vice versa.

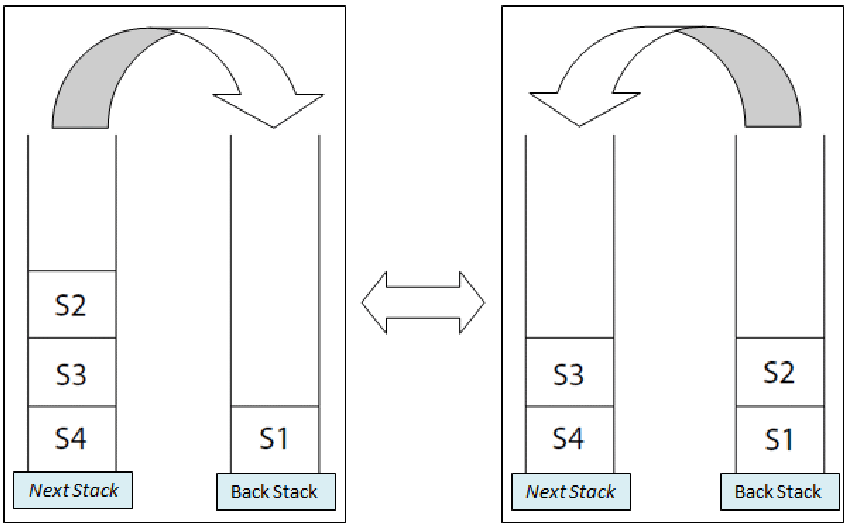


Figure 4.5.2: Going from state1 to state2 and vice versa

This approach was followed under the consideration that each animation state contains information only about the data that need to be modified and not information about the global data of the entire visualization. Going from one state to another, only the data of the current state has to be modified by executing the `undo` and `redo` operations. The main advantage of this approach is that we do not have to record all the data of the visualization. Instead, we record only the required data, which means that we avoid much storage consumption. A main drawback of this approach is that we can only go from one state to the next or the previous one. For example, going from state1 to state4 we have to execute the redo operations of states 2, 3 and 4 in sequence. Similarly going from state4 to state1 we have to execute the undo operations on states 4, 3 and 2 in sequence.

The idea of this design decision is to introduce a Tree Visualization Engine that will have direct access on a Binary Search Tree and all the global data information. Depending on the changes performed by the `undo` and `redo` operations of the current animation state, it visualizes the modified tree on the screen. Because the structure of a Binary Search Tree changes dynamically due to insertions and deletions, creating a tree visualization that has fixed positions for each node is inefficient. The problem is that the left and right sub-trees can be of different heights. The following are three different design solutions considered during the design of the tree visualization engine.

Solution 1: Starting at the root and for each node, place its left and right child into symmetrical positions as in figure 4.5.3

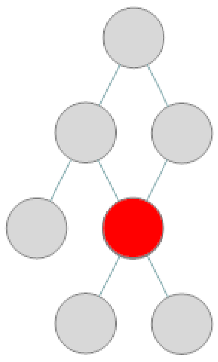


Figure 4.5.3: Binary tree with conflict nodes

Problem: The main problem of this approach is that we have node conflicts. This means that there will be nodes placed one on top of the other. This will hide information from the user, as he will not be able to observe the entire set of nodes of the tree.

Solution 2: Pre-calculate each node position before inserting the nodes as in figure 4.5.4.

Problems: This solution works well for complete Binary Search Trees (each left and right sub-tree at each node have exactly the same height). The problem here is that we cannot predict how many nodes a tree will have. In addition, non-complete Binary Search Trees can become very wide consisting of a few nodes only. This will result in some space to be wasted. Figure 4.5.5 shows a case where the right sub-tree has height 1 and the left sub-tree has height 3.

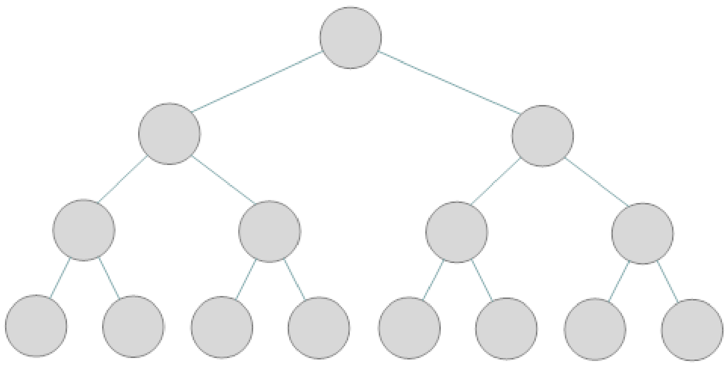


Figure 4.5.4: A complete Binary Search Tree

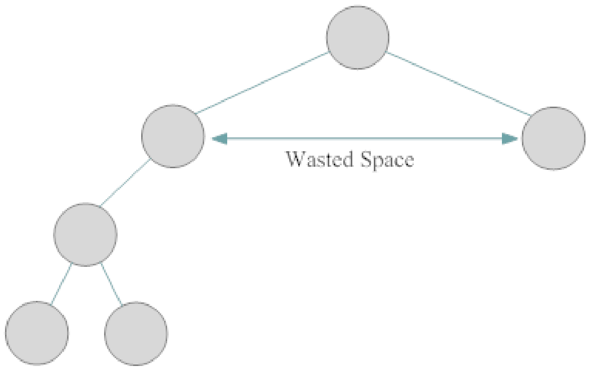


Figure 4.5.5: A Binary Search Tree with wasted space between nodes

Solution 3: An ideal solution would be to exploit the wasted space shown in figure 4.5.5 and place the nodes as close as they can be so that the tree will not be too wide. Figure 4.5.6 shows an ideal visualization of a tree where the empty rectangles denote a null node which is a leaf node's child.

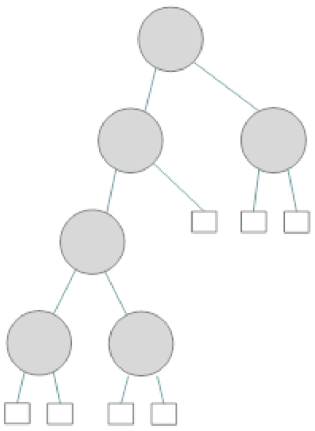


Figure 4.5.6: A refinement of the tree in figure 4.9

A way for achieving this design is to first calculate the total width of each node such that the total width of a leaf node must be equal to its diameter, and the total width of the root must be equal to the total width of the tree. The recursive algorithm in figure 4.5.7 shows how the total width for each node is calculated where `NODE_WIDTH` is a constant number representing the diameter of a node. Similarly, a null node's width is equal to `NODE_WIDTH / 2`.

```
void calculateTotalWidths(Node node)
{
    if(!node.isNull)
    {
        node.width = NODE_WIDTH;
        calculateTotalWidths(node.left);
        calculateTotalWidths(node.right);
        node.width = node.left.width + node.right.width;
    }
    else
    {
        node.width = NODE_WIDTH / 2;
    }
}
```

Figure 4.5.7: Algorithm for calculating the total width of each node in a Binary tree

Once the total width of each node is calculated, they have to be positioned on the screen. The idea is to place the root of the tree on the top middle position of the screen and adding the rest of the nodes relative to their parent's position and based on their total width. An algorithm for calculating the position of a node is shown in figure 4.5.8 where `MIDDLE_OF_SCREEN` is a number representing the middle of the screen that the tree is displayed on and `HEIGHT_DIFFERENCE` is a constant number representing the difference in height between a child and its parent.

```
void calculatePosition(Node node)
{
    if(node.isRoot())
    {
        node.position = (MIDDLE_OF_SCREEN, 0)
    }
    else
    {
        Node parent = node.parent;
        int x = parent.position.x - (parent.width - node.width) / 2;
        int y = parent.position.y + HEIGHT_DIFFERENCE;
        if(node.isRight())
        {
            x += parent.left.width();
        }
        node.position = (x, y);
    }
}
```

Figure 4.5.8: Algorithm to calculate the position of each node in a Binary tree

Finally, after calculating the position of each node, what is left is to connect each child to its parent. The points of a line connecting a child to its parent are shown in figure 4.5.9.

```
int x1 = parent.position.x + NODE_WIDTH / 2;
int y1 = parent.position.y + NODE_WIDTH / 2;
int x2 = child.position.x + NODE_WIDTH / 2;
int y2 = child.position.y + NODE_WIDTH / 2;
```

Figure 4.5.9: Algorithm for connection points between a child node and its parent

4.6 Design of the Graphical user Interface

The user interface is a vital part of the system design because it must allow the interaction between the user and the system to be easy. In particular, during the development of the user interface we need to take into consideration the design principles identified in section 3.5.2.

The user interface of the tool has been designed to include a single "main menu" that will be displayed to the user when the tool is loaded. It should provide sufficient information in order to interact with the application. Figure 4.6.1 shows an outline of the "main menu" which is divided into five main displays. Each of these displays is described next.

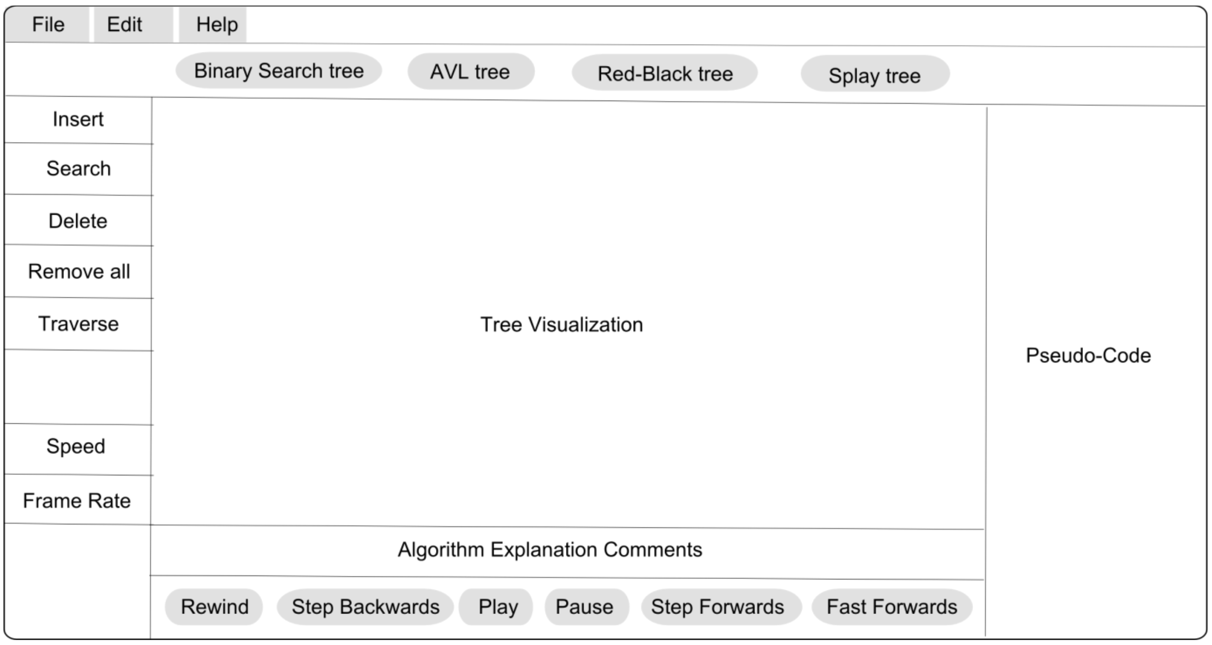


Figure 4.6.1: The graphical user interface

Tree selection display: Displays the types of Binary Search Trees and allows the user to select one of them

Tree visualization display: Displays the visualization of the selected tree during the animation

Operations display: Displays the operations a user is allowed to perform on the selected tree such as searching, insertion, deletion and traversals

Animation control display: Displays the buttons for controlling the animation such as play, pause, step backwards, step forwards, rewind and fast forwards

Pseudo-code display: Displays the pseudo-code for the algorithm that is being executed, and allows lines to be highlighted

Algorithm Explanation Comments display: Displays explanation comments for the highlighted line

Menu Bar: Provides help support, animation configurations and an option to terminate the program.

An important interaction attribute for the tool is to have the ability to receive input from the user. For example, when the user wants to insert a node into the tree, it would be good idea to be authorized to enter its own value in a text field or even let the tool to generate a random value on its behalf. Figure 4.6.2 shows the window designed to appear on the screen when the user wants to insert a node into the tree.

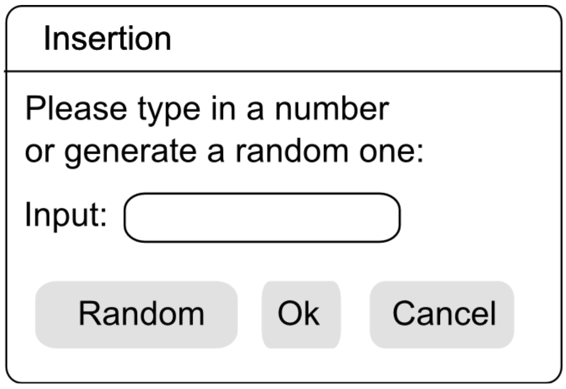


Figure 4.6.2: Insertion window

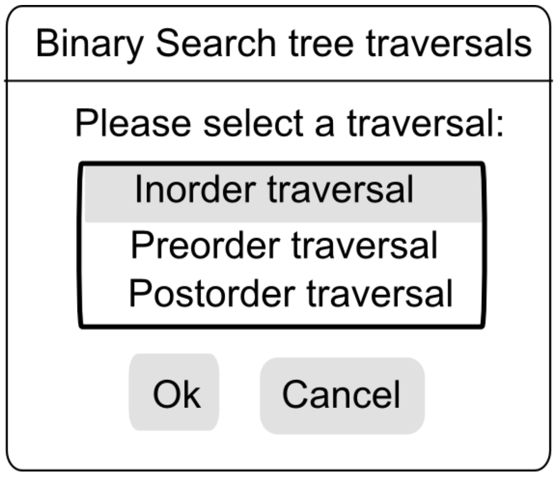


Figure 4.6.3: Binary Search Tree traversals window

A similar window has been designed to be displayed for the deletion and searching. Finally, an input for selecting a tree traversal has been designed and is displayed in figure 4.6.3.

5 Implementation

Implementation is the software engineering process that follows the design and its main concern is to bring the project to life. This chapter will describe in detail how each part of the design is implemented. Then all the implemented parts will have to be integrated in order to form the final system, which should meet the identified requirements. The chosen language for the implementation is Java as decided upon in section 3.3. The first component to be implemented is the Algorithm Simulation and then the Animation Generator, which will produce a list of animation states. Next, the Animation Engine will be implemented in order to control the primitive operations of the animation states. Finally a final system that should provide full control over the animation and aid the understanding of Binary Search Trees will be developed.

5.1 Implementing the Algorithm Simulation

The key idea for the trees implementation is to implement an abstract class called `BinaryTree` that contains all the common methods used in a Binary Tree. Next, we have extended this class into a `BinarySearchTree` class for representing a Binary Search Tree. For the Balanced Trees, a new class called `BalancedBinaryTree` was implemented. This class inherits all the properties of a Binary Search Tree and provides some additional methods for tree rotations that are common to all Balanced and Self-Adjusting Trees. Finally, a separate sub-class of `BalancedBinaryTree` was created for each type of Balanced Tree as AVL, Red-Black and Splay. The following class diagram in figure 5.1.1 shows the inheritance of the tree class developed.

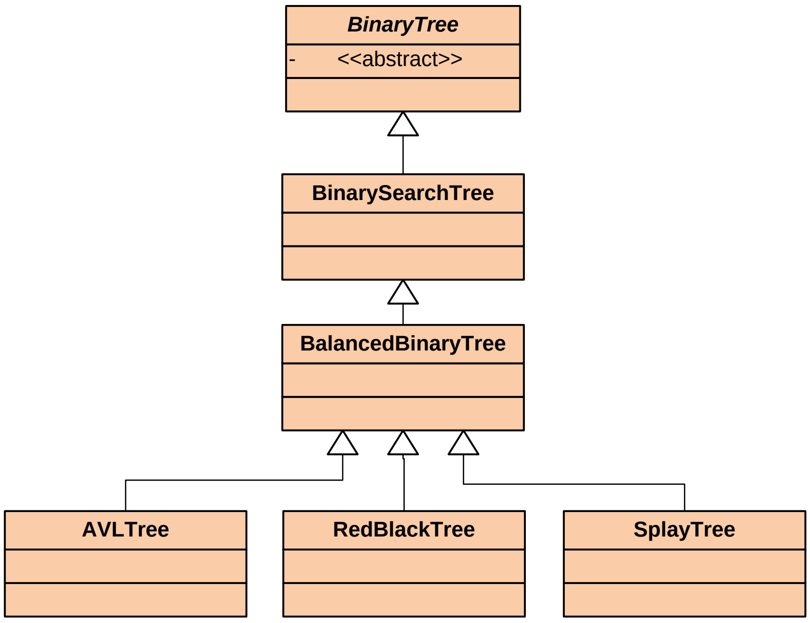


Figure 5.1.1: Binary trees inheritance

For the tree nodes, a similar decision was made to implement a separate class to represent a node of each type of tree. The decision was made under the consideration that each node has some unique properties. For example, an AVL node has a `balance` value that is the difference in height between its left and right sub trees. Similarly, a Red-Black node has either a red or a black colour. For this reason, we have implemented an abstract class called

`BinaryTreeNode` that contains operations as getting the left and right child that are common to each Binary Tree node so that it can be extended by other classes that need to implement their own node properties. A class diagram for the node classes is shown in figure 5.1.2.

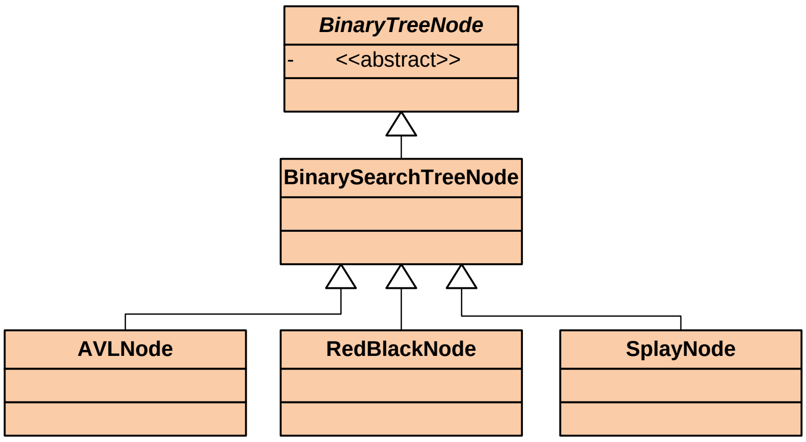
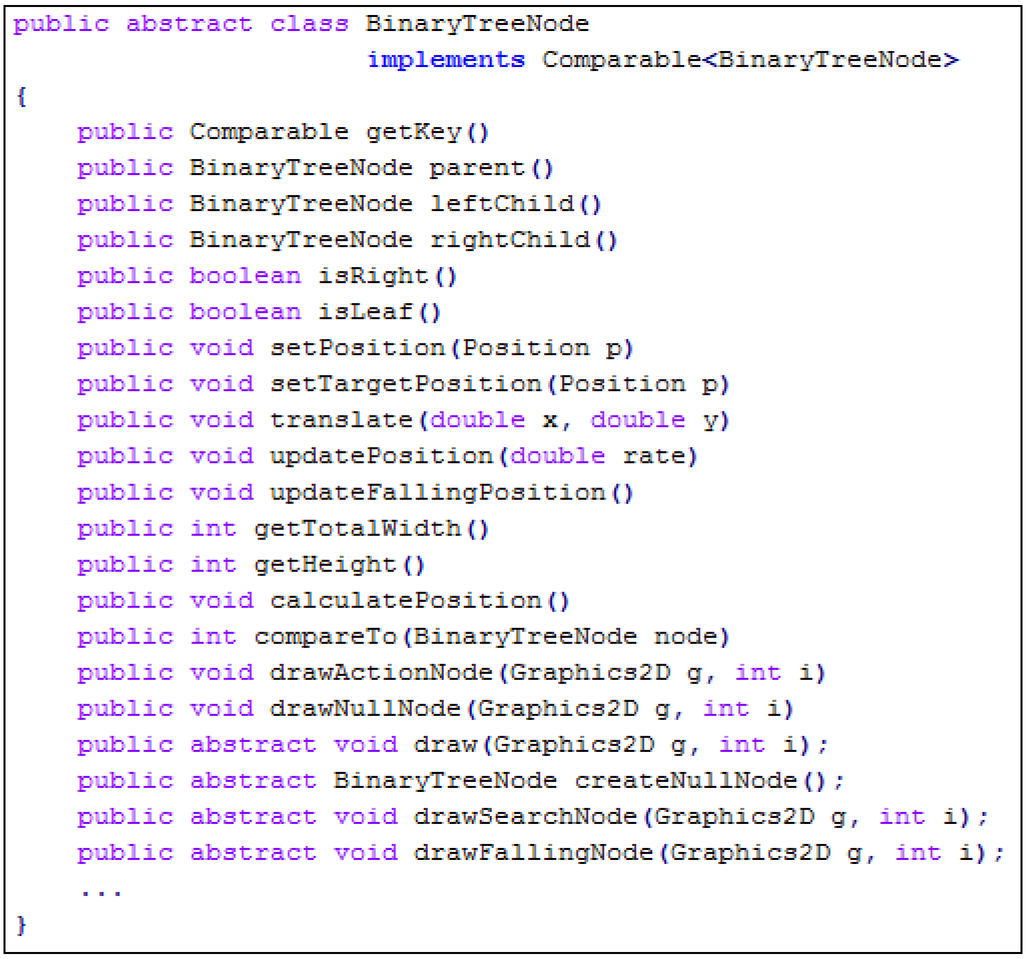


Figure 5.1.2: Binary Tree nodes inheritance

Apparently, the inheritance of the trees allows the tool to be easily extended by adding new types of Binary Search Tree. Each of these classes provides methods for performing operations on the tree as insertion, deletion and searching. In addition, some methods are used by the Animation Engine in order to display the tree on the screen and show any changes performed by the animation states. Some of the methods provided by the `BinaryTree` and `BinaryTreeNode` abstract classes are shown in figures 5.1.3 and 5.1.4 respectively.

```
public abstract class BinaryTree
{
    public boolean isEmpty()
    public BinaryTreeNode getRoot()
    public  BinaryTreeNode find(Comparable key)
    public BinaryTreeNode insert(Comparable key)
    public BinaryTreeNode delete(Comparable key)
    protected BinaryTreeNode spliceOut(BinaryTreeNode node)
    private BinaryTreeNode getSuccessor(BinaryTreeNode node)
    public void traversal(int i)
    public int getHeight()
    public void calculateTotalWidths(BinaryTreeNode node)
    public void calculateSubTree(BinaryTreeNode tnode)
    public void setAnimationGenerator(AnimationGenerator aG)
    public void updateTreePosition(BinaryTreeNode node, double rate)
    public void draw(Graphics2D g, int i)
    public String getPseudoCode(int i)
    public abstract String getExplanationFile();
    public abstract BinaryTreeNode createNode(Comparable data);
    public abstract BinaryTreeNode createNullNode();
    ...
}
```
Figure 5.1.3: Snippet of the BinaryTree class implementation



```
public abstract class BinaryTreeNode
                      implements Comparable<BinaryTreeNode>
{
    public Comparable getKey()
    public BinaryTreeNode parent()
    public BinaryTreeNode leftChild()
    public BinaryTreeNode rightChild()
    public boolean isRight()
    public boolean isLeaf()
    public void setPosition(Position p)
    public void setTargetPosition(Position p)
    public void translate(double x, double y)
    public void updatePosition(double rate)
    public void updateFallingPosition()
    public int getTotalWidth()
    public int getHeight()
    public void calculatePosition()
    public int compareTo(BinaryTreeNode node)
    public void drawActionNode(Graphics2D g, int i)
    public void drawNullNode(Graphics2D g, int i)
    public abstract void draw(Graphics2D g, int i);
    public abstract BinaryTreeNode createNullNode();
    public abstract void drawSearchNode(Graphics2D g, int i);
    public abstract void drawFallingNode(Graphics2D g, int i);
    ...
}
```

Figure 5.1.4: Snippet of the BinaryTreeNode class implementation

The full implementation of these classes can be found in Appendices E1 and E2. The implementation of the trees represents the Algorithm Simulation that needs to record data information during an algorithms execution and pass this data to the Animation Generator, which will create the animation states. To achieve this communication, some extra lines of augmented code have been added to each algorithm's implementation.

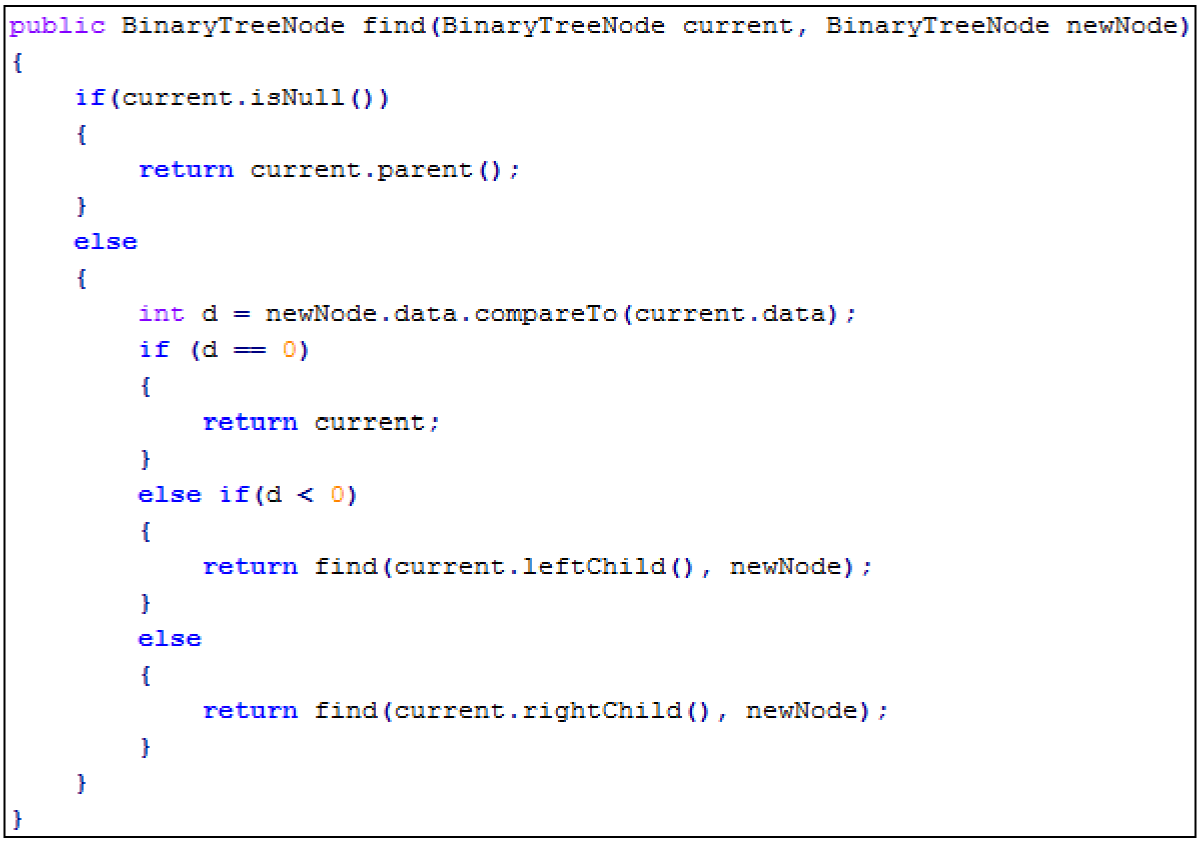
An example of the augmented code added to the searching algorithm implementation is presented in figure 5.1.5. The `animationGenartor` is an instance of the `AnimationGenerator` class and is passed as a parameter to the `setAnimationGenerator` method of the `BinaryTree` subclass that is currently executed. (To be discussed later in "Implementing the Animation Generator" section).

The `addSearchingState` method signals to the Animation Generator that a searching state should be created where `newNode` is the node that is searched for, `current` is the node that is currently compared with `newNode` and `current.parent()` is the parent of the `current` node. Similarly, the `addHighlightAndExplanation` method indicates that a highlighting and explanation state should be created where the number indicates the pseudo-code line that needs to be highlighted and the text in quotes is the English explanation for that line.

```
public BinaryTreeNode find(BinaryTreeNode current, BinaryTreeNode newNode)
{
    ...
    else if(!current.isNull())
    {
        animationGenerator.addSearchingState(newNode, current.parent(), current);
    }
    animationGenerator.addHighlightAndExplanationState(2, "Check if reach the" +
                                                       " end of the tree");
    if(current.isNull())
    {
        animationGenerator.addHighlightAndExplanationState(3,"Key not found!");
        return current.parent();
    }
    else
    {
        animationGenerator.addHighlightAndExplanationState(4, "Comparing "+
                                                    newNode+" with "+current);
        int d = newNode.data.compareTo(current.data);
        animationGenerator.addHighlightAndExplanationState(5, "Check if "+
                                                    newNode+ " = "+current);
        if (d == 0)
        {
            animationGenerator.addHighlightAndExplanationState(6, "Key found!!");
            return v;
        }
        ...
    }
}
```

Figure 5.1.5: The augmented searching algorithm implementation

Having augmented all the algorithms with code for recording the algorithm states, we can now move on to the next part of the implementation, the Animation Generator.



```
public BinaryTreeNode find(BinaryTreeNode current, BinaryTreeNode newNode)
{
    if(current.isNull())
    {
        return current.parent();
    }
    else
    {
        int d = newNode.data.compareTo(current.data);
        if (d == 0)
        {
            return current;
        }
        else if(d < 0)
        {
            return find(current.leftChild(), newNode);
        }
        else
        {
            return find(current.rightChild(), newNode);
        }
    }
}
```

Figure 5.1.6: The implementation of searching algorithm

A class diagram for the Algorithm simulation is shown in figure 5.1.6 below.

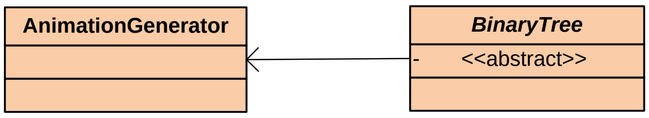


Figure 5.1.7: Class diagram for the Algorithm Simulation

5.2 Implementing the Animation Generator

For the Animation Generator, a class called `AnimationGenerator` was implemented that contains methods used by the Algorithm Simulation in order to pass data information for each algorithmic state. Some of these methods are shown in figure 5.2.1. Each time one of these methods is invoked by the Algorithm Simulation, the Animation Generator creates the appropriate animation state depending on the method that is called.

The main concern of this approach is how the recorded data (algorithm state) will be translated into an animation state. The key idea is to create an interface called `AnimationState` (shown in figure 5.2.2) that represents an animation state and exports the two primitive operations, `undo` and `redo` as mentioned in section 4.4. In addition to these methods, two extra methods have been defined called `isModifying` and `canBeDisable`. The `isModifying` method denotes whether an animation state modifies the structure of the tree. The `canBeDisabled` method indicates whether an animation state can be disabled, in such case it will not become visible to the user. The last method was introduced to allow the pseudo-code and explanation comments to be easily disabled and enabled.

```
public void addSearchingState(BinaryTreeNode searchNode,
                              BinaryTreeNode from,
                              BinaryTreeNode to)
public void addInsertionState(BinaryTreeNode parent,
                              BinaryTreeNode newNode,
                              int index)
public void addDeletionState(BinaryTreeNode parent,
                             BinaryTreeNode toBeDeleted,
                             BinaryTreeNode newChild,
                             int index,
                             int childIndex )
public void addHighlightAndExplanationState(int line,
                                            String expls)

public void addSearchNodeState(BinaryTreeNode searchNode,
                               BinaryTreeNode node,
                               boolean setSearchNode)
...
```

Figure 5.2.1: `AnimationGenerator` class methods

```
public interface AnimationState
{
    void undo();
    void redo();
    boolean isModifying();
    boolean canBeDisabled();
}
```

Figure 5.2.2: The AnimationState interface

Having the animation state interface defined, a class for each of the animation state types has to be developed. Such classes will have to implement all the methods of the `AnimationState`, based on the data received from the Algorithm Simulation. An example of an animation state class is the `InsertionState` whose redo method is implemented to provide a transition from the currently visible state to this animation state where a node is inserted in the tree. Similarly, the undo method of this state is used to reverse this transition to the previous animation state where the node was not in the tree.

The following is a list of all the identified animation states, which are divided into 3 categories.

1. States that result by modifying the tree data structure
    a. `InsertionState`
    b. `SwapState`
    c. `DeletionState`
    d. `RotationState`

    e. `ColorFlipstate`
2. States that result by highlighting areas of interest on the tree
    a. `SearchingState`
    b. `SearchNodeState`
    c. `ActionNodeState`
3. States that result by highlighting the line of code that is currently executed as well as displaying explanation comments that line.
    a. `ChangePseudoCodeState`
    b. `HighlightAndExplanationState`

The key idea during the implementation of the animation state classes was to allow each state to cooperate with the Animation Engine in order to perform the appropriate changes on the screen. For this reason, the constructor of the `AnimationGenerator` takes as parameters an instance of each of the following classes of the Animation Engine.

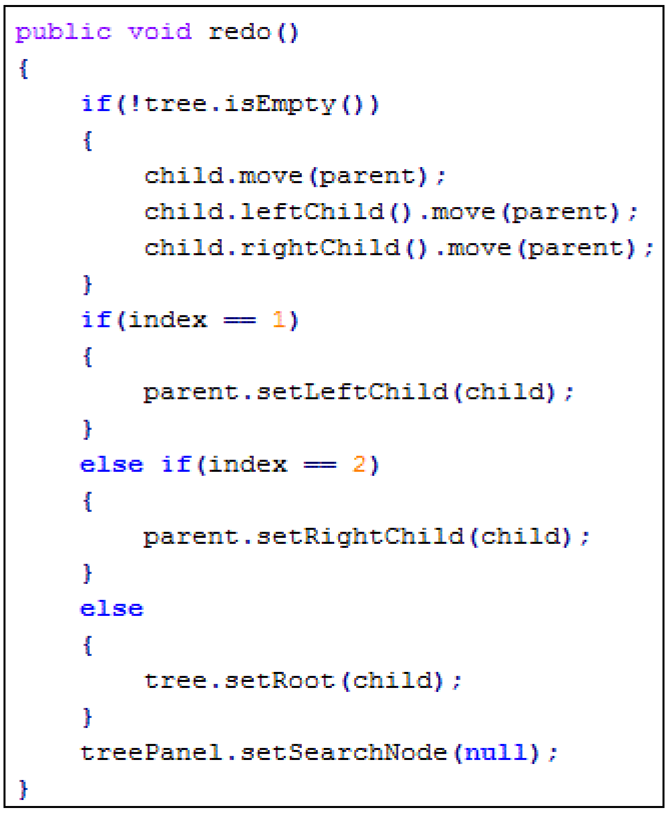
- `BinaryTree`: The tree to be animated
- `TreePanel`: The Tree Visualization Engine that displays the tree
- `PseudoPanel`: The panel that displays the pseudo-code for each algorithm
- `ExplanationPanel`: The panel that displays the explanation comments for each algorithmic step

These instances are used by the animation state classes in order to perform the appropriate changes on the screen. For example, the `HighlightAndExplanationState` uses the `PseudoPanel` and `ExplanationPanel` objects in order to highlight a line of code that is being executed and display explanation comments for that line respectively.

The `TreePanel` class cooperates with the animation states that modify the tree and the states that highlight areas of interest on the tree. Its main task is to calculate the target position of each node in the tree, which is the position that each node must be placed. Based on their current positions it slightly moves each node periodically, until it reaches its target position. This provides smooth transitions from one node position to another. This approach has a great advantage that the animation states do not have to be concerned about the position a node will be placed. For example, when a node is set as a left child using `parent.setLeftChild(node)` statement, the `TreePanel` will animate this node to the position where it should be placed. Even if a state modifies the position of a node, the node's position will be re-calculated and moved to the position where it should be. This was the main reason that each state focuses only on the data that it modifies and not on any other data of the tree. The implementation of the `TreePanel` class is very crucial and will be discussed in the next section.

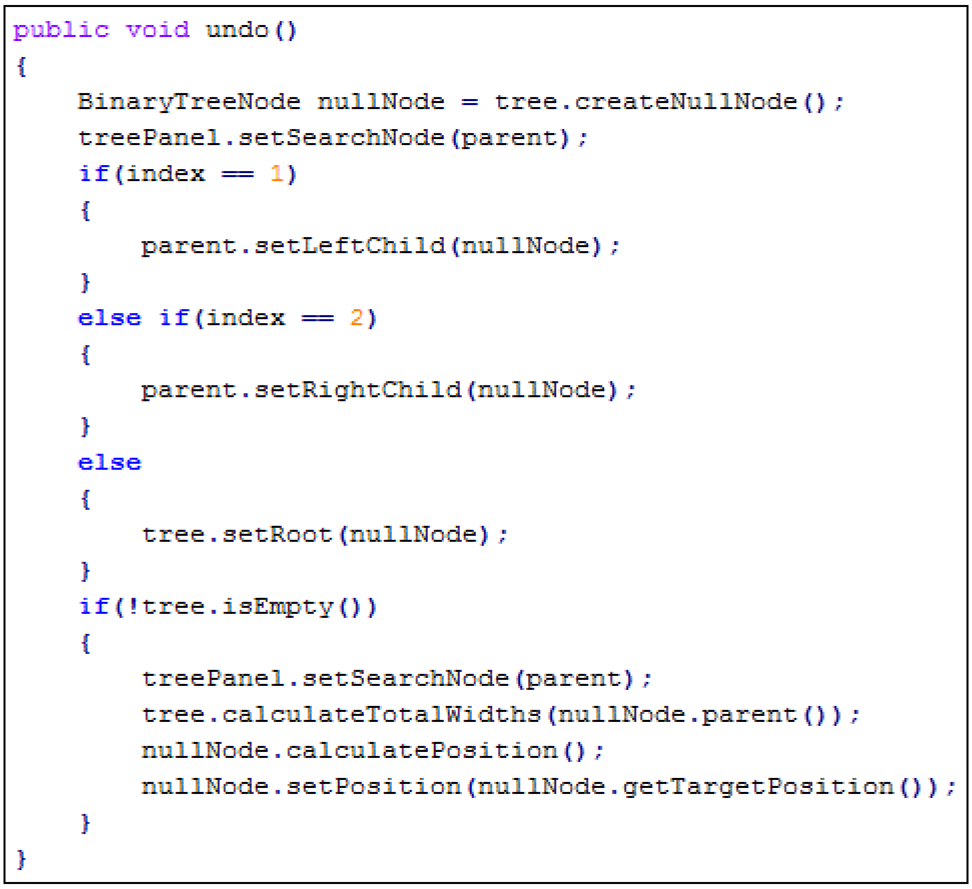
The following discussion presents the implementation of the `redo` and `undo` methods for one of each of the three animation state categories stated above.

An animation state that modifies the data structure is the `InsertionState` class. The implementations of its `undo` and `redo` methods are shown in figures 5.2.3 and 5.2.4 respectively.



```java
public void redo()
{
    if(!tree.isEmpty())
    {
        child.move(parent);
        child.leftChild().move(parent);
        child.rightChild().move(parent);
    }
    if(index == 1)
    {
        parent.setLeftChild(child);
    }
    else if(index == 2)
    {
        parent.setRightChild(child);
    }
    else
    {
        tree.setRoot(child);
    }
    treePanel.setSearchNode(null);
}
```

Figure 5.2.3: The redo method of the InsertionState class



```java
public void undo()
{
    BinaryTreeNode nullNode = tree.createNullNode();
    treePanel.setSearchNode(parent);
    if(index == 1)
    {
        parent.setLeftChild(nullNode);
    }
    else if(index == 2)
    {
        parent.setRightChild(nullNode);
    }
    else
    {
        tree.setRoot(nullNode);
    }
    if(!tree.isEmpty())
    {
        treePanel.setSearchNode(parent);
        tree.calculateTotalWidths(nullNode.parent());
        nullNode.calculatePosition();
        nullNode.setPosition(nullNode.getTargetPosition());
    }
}
```

Figure 5.2.4: The `undo` method of the `InsertionState` class

The `redo` method first moves the node to be inserted on the position of its parent and then connects it as a left or right child. As it was mentioned above, the transition of the inserted node from its parent's position to its final position is a task performed by the Tree Visualization Engine.

The `undo` method simply creates a null node represented as an empty rectangle, which replaces the newly inserted node. The last if statement helps the Tree Visualization Engine to re-calculate the position of the nodes on the tree in order to provide a smooth transition from the state where the node was inserted to the previous state where it was not existed in the tree.

The next category of animation states are the states that highlight areas of interest on the tree visualization such as the searching state. Similar to the insertion state, the `SearchingState` class implements the `redo` and `undo` methods of the `AnimationState` interface. The implementation of these methods is shown below in figures 5.2.5 and 5.2.6 where `from` is the parent and `to` is the child node.

```
public void redo()
 {
     double dx = to.getPosition().x - from.getPosition().x;
     double dy = to.getPosition().y - from.getPosition().y;
     searchNode.setPosition(new Position(from.getPosition()));
     treePanel.setSearchNode(searchNode);
     from.translate(-dx * 0.2, -dy * 0.2);
     searchNode.setTargetPosition(to.getPosition());
 }
```

Figure 5.2.5: The `redo` method of the `SearchingState` class

The idea behind the method in figure 5.2.5 above is to move the search node from the "**`from`**" node towards the "**`to`**" node. For more advanced animation effects, when the search node starts moving it jabs the "**`from`**" node. This is done in the following statement:
`from.translate(-dx * 0.2, -dy * 0.2)`
The idea is that when a node is jabbed (forced to be slightly moved from its current position), the Tree Visualization Engine will smoothly move this node back to its previous position. Next, the target position of the search node is set to the position of the "`to`" node. The tree visualization engine performs the transition based on the search node's target position.

```
public void undo()
{
    double dx = from.getPosition().x - to.getPosition().x;
    double dy = from.getPosition().y - to.getPosition().y;
    searchNode.setPosition(new Position(to.getPosition()));
    treePanel.setSearchNode(searchNode);
    to.translate(-dx * 0.2, -dy * 0.2);
    searchNode.setTargetPosition(from.getPosition());
}
```

Figure 5.2.6: The `undo` method of the `SearchingState` class

The undo operation of this state does exactly the opposite. It moves the search node from the "**to**" node towards the "**from**" node including a jabbing of the "**to**" node.

The last category of the animation states highlights the line of code that is currently executed, and displays explanation comments for that line. One such state is the HighlightAndExplanationState and its redo and undo methods are prsented in figures 5.2.7 and 5.2.8.

```
public void redo()
{
    pseudoPanel.highlightLine(line);
    explanationPanel.addExplanation(explanation);
}
```

Figure 5.2.7: The redo method of the HighlightAndExplanationState class

The redo method simply calls the highlightLine method of the PseudoPanel class that highlights the line number given as parameter. Similarly, the addExplanation method of the ExplanationPanel class displays the explanation String given as parameter.

```
public void undo()
{
    pseudoPanel.highlightLine(oldLine);
    explanationPanel.addExplanation(oldText);
}
```

Figure 5.2.8: The undo method of the HighlightAndExplanationState class

The undo operation invokes the same methods as in the redo operation with the difference that the parameters for the PseudoPanel and ExplanationPanel classes are the previous line of code and explanation comments respectively.

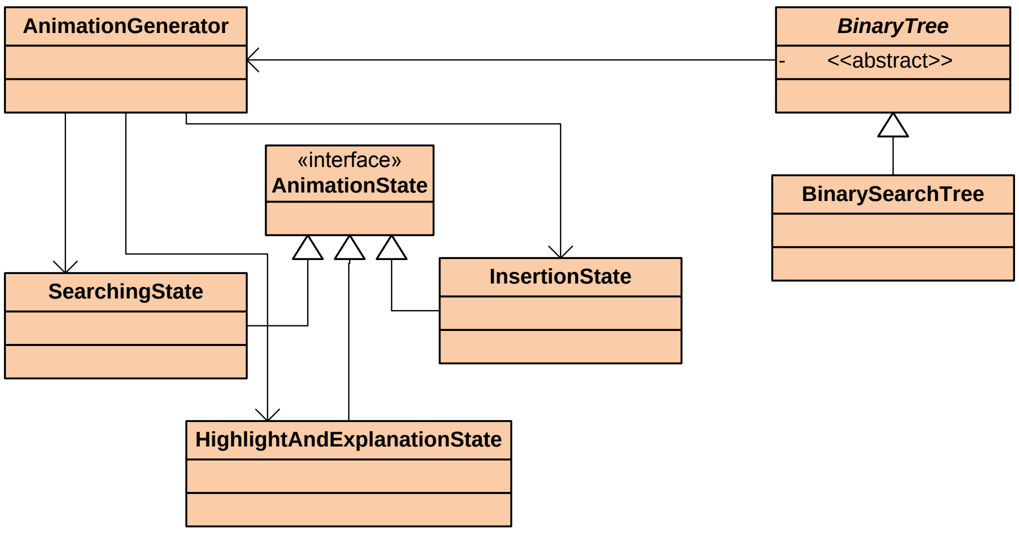


Figure 5.2.9: Class diagram for the Animation Generator

Finally, a class diagram for the Animation Generator is shown in figure 5.2.9 above. Having implemented the animation states, we will have to provide a mechanism that will be used to control them. This mechanism is the Animation Engine and will be explained in detail in the next section.

5.3 Implementing the Animation Engine

An animation state performs operations that modify specific data on the `BinaryTree`, `TreePanel`, `PseudoPanel` and `ExplanationPanel` classes. In particular, the animation states that modify the tree hold references of the tree nodes that they modify and not a copy of them. This means that they act directly on the tree data structure that is being visualized by the `TreePanel`. Consequently, during an algorithm's execution, the tree modifications become visible to the user.

The idea is to make use of the `undo` method of each animation state in order to reverse the animation to its initial state, which is the state before the changes by the algorithm's execution were performed. To achieve this, each time an animation state is created is pushed onto a stack such that the lastly created state should be on the top of the stack.

The first task of the Animation Engine is to access the animation states stack. We call this stack the `back` stack that contains the states to be undone. Next, for each animation state on the top of the `back` stack, its `undo` method is invoked and the state is pushed onto another stack called the `next` stack, until the `back` stack becomes empty. This will roll back the system from its final state to the initial state, which is the state before the algorithm has been executed. For example, when a node is inserted in the tree, the final state is the state where the algorithm finishes with its execution and shows the tree containing the inserted node. To prevent this from becoming visible to the user, the states created by the algorithm's execution are used to roll back the tree to the state before the node was inserted. At this state, the user will be prompted to start playing the animation. This task is performed every time an algorithm finishes with its execution and is done in rapid sequence such that the user is not able to observe the changes, having the illusion that nothing has changed on the tree.

The main classes representing the animation engine are the `TreePanel` and the `AnimationController`. Their full implementation can be found at Appendices E3 and E4.

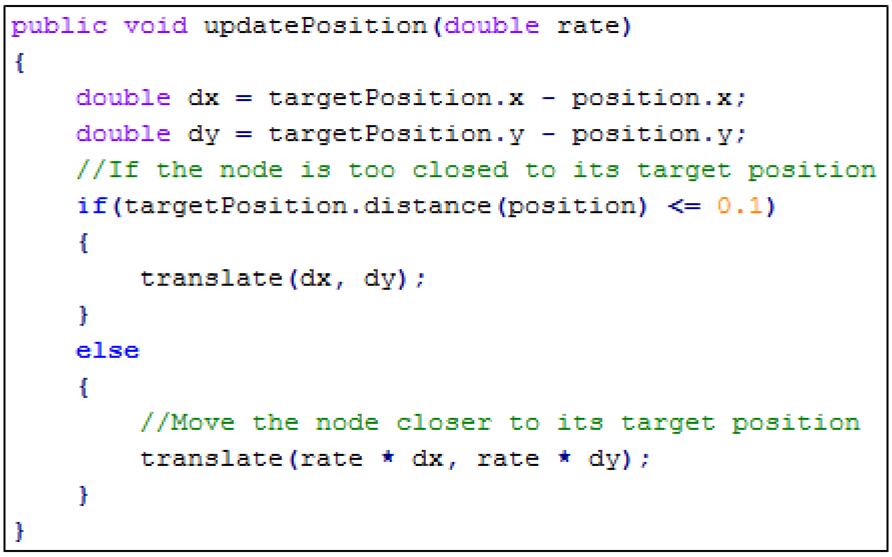
Beginning with the first class, the `TreePanel` is a sub-class of the `JPanel` that implements the `Runnable` interface and is used to visualize a tree on the screen. The `TreePanel` is called the Tree Visualization Engine and its main task is to periodically calculate and update the positions of each node in order to draw the tree. This is one of the most important aspects of the animation engine. The implementation of the `TreePanel's` run method is shown below in figure 5.3.1.

The way the `TreePanel` calculates the position of each node is based on the two algorithms for calculating the total width of each node and then their positions based on their total widths as described in section 4.5.

```
public void run()
{
    while(!terminated)
    {
        //Calculate the total widths for each node in the tree
        tree.calculateTotalWidths(tree.getRoot());
        //Update the position of each node in the tree
        tree.updateTreePosition(tree.getRoot(), frameRate);
        //For searching operation
        if(searchNode != null)
        {
            searchNode.updatePosition(frameRate + 0.02);
        }
        //For remove operation
        if(fallingNode != null)
        {
            fallingNode.updateFallingPosition();
            if(!fallingNode.isTarget())
            {
                fallingNode = null;
            }
        }
        //Draw the tree
        repaint();
        try
        {
            Thread.sleep(speed);
        }
        catch (InterruptedException ex){}
    }
}
```

Figure 5.3.1: Snapshot of the implementation of the visualization engine

The idea is that each node has a current and a target position. The current position depicts the actual position of the node whereas the target position indicates the position where the node must move. The role of the `updateTreePosition` method is to calculate the target position for each node. Then it slightly moves each node from its current position closer to its target position. As a result, each time this method is invoked it moves the nodes of the tree closer to their target position until they reach their final position. The algorithm that computes the distance a node will move each time towards its target position is a method in the `BinaryTreeNode` abstract class and is shown in figure 5.3.2. The `rate` parameter is a decimal number between 0.05 and 0.25 that indicates how far the node is to be moved from its current position. More comprehensively, the `rate` value defines how smooth the transition of a node from its current to its target position will be. The smaller the `rate` value is, the smoother the transition will be.



```
public void updatePosition(double rate)
{
    double dx = targetPosition.x - position.x;
    double dy = targetPosition.y - position.y;
    //If the node is too closed to its target position
    if(targetPosition.distance(position) <= 0.1)
    {
        translate(dx, dy);
    }
    else
    {
        //Move the node closer to its target position
        translate(rate * dx, rate * dy);
    }
}
```

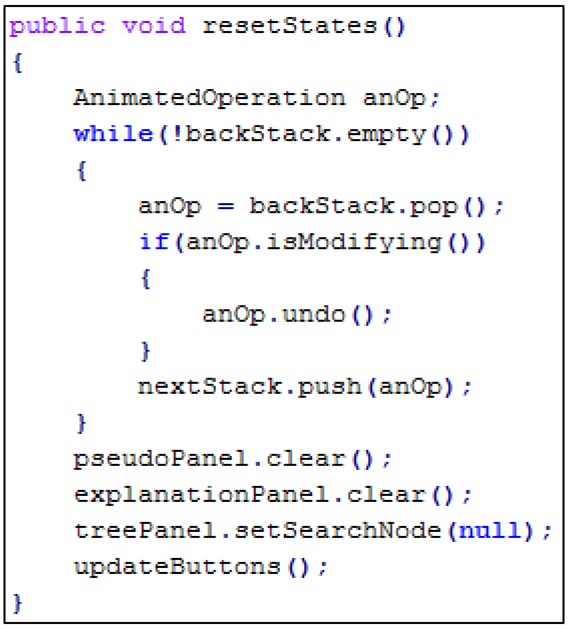
Figure 5.3.2: Implementation of the `updatePosition` method

The implementation of the `TreePanel` class is the most important part of the Animation Engine because each animation state that modifies the tree data structure will not have to be concerned about how the tree will be visualized. The `TreePanel` class is responsible to do this automatically, making the changes performed by the animation states to appear smoothly to the user.

The `AnimationController` class is the component that controls the animation and all the inputs. When an operation is commanded, it calls the relevant method of the chosen Binary Search Tree. When the method is executed, the animation states computed by the algorithm are passed by the Animation Generator to the `AnimationController`. The first task the Animation Controller does is to roll back the animation to the initial state, the state before the algorithm was executed. The algorithm in figure 5.3.3 presents how the roll back is done using a method called `resetStates`.

The algorithm during its execution modifies only the tree data structure and not any other components of the system like the `PseudoPanel` and `ExplanationPanel`. This is where the `isModifying` method of the `AnimationState` is used. Such method indicates whether the current `AnimationState` that is on the top of the back stack modifies the data structure. If it is, then it has to be undone, otherwise, it is just skipped to improve the performance of the system.

After the roll back task is performed, the animation is ready to begin. For this action, a set of animation control buttons were implemented. The buttons and their associated operations are listed in table 3.



```
public void resetStates()
{
    AnimatedOperation anOp;
    while(!backStack.empty())
    {
        anOp = backStack.pop();
        if(anOp.isModifying())
        {
            anOp.undo();
        }
        nextStack.push(anOp);
    }
    pseudoPanel.clear();
    explanationPanel.clear();
    treePanel.setSearchNode(null);
    updateButtons();
}
```

Figure 5.3.3: Implementation of the `resetStates` method

Button	**Operation**
Play	Starts playing the animation
Pause	Pauses the animation
Next	Steps the animation backwards to the previous state
Previous	Steps the animation forwards to the next state
Rewind	Rolls back the animation to initial state
Fast Forwards	Steps the animation forwards to the final state

Table 3: Animation control buttons and their operations

The implementation of the control buttons is typically based on two main methods that are used appropriately by all these buttons apart the `pause`. These methods are the `stepForwards` and `stepBackwards` shown in figure 5.3.4. When these methods are invoked by the `next` and `previous` buttons, they step the animation to the next and previous state respectively.

For the `play` button, a `Timer` was implemented that calls the `stepForwards` method periodically every `ALGORITHM_STEP_DELAY` time units. The implementation of the timer is shown in figure 5.3.5. The animation starts by calling `timer.start()` and is paused by calling the `timer.stop()` method of the `Timer` class.

The `Rewind` button simply calls the `resetStates` method described previously. Similarly a new method called `redoStates` was implemented (used by `Fast Forwards` button) which is similar to the `resetStates` methods with the difference that takes each `AnimationState` from the top of the `next` stack, invokes its `redo` method and then pushes it onto the `back` stack until the `next` stack becomes empty.

```
public void stepForwards()
{
    AnimatedOperation anOp;
    if(!nextStack.empty())
    {
        anOp = nextStack.pop();
        anOp.redo();
        backStack.push(anOp);
    }
    updateButtons();
}
```

```
public void stepBackwards()
{
    AnimatedOperation anOp;
    if(!backStack.empty())
    {
        anOp = backStack.pop();
        anOp.undo();
        nextStack.push(anOp);
    }
    updateButtons();
}
```

Figure 5.3.4: Implementation of the `stepForwards` and `stepBackwards` methods

```
timer = new Timer(Globals.ALGORITHM_STEP_DELAY, new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        stepForwards();
    }
});
```

Figure 5.3.5: Implementation of the Timer class

The last statement of the `stepForwards` and `stepBackwards` methods (`updateButtons`) is used to enable and disable the animation control buttons accordingly, (i.e. `back` button becomes disabled when there is no previous state, which means that the `back` stack is empty). Additionally, all the buttons apart from the `pause` button are disabled during the animation run-time (when the `play` button is pressed). Furthermore, an additional code was added to these two methods in order to skip the Highlight and Explanation states when the user disables them. This is done in a while loop that skips all the states whose value returned from the call to their `canBeDisabled` method is true.

Finally, a class diagram for the Animation Engine is shown in figure 5.3.6 below.

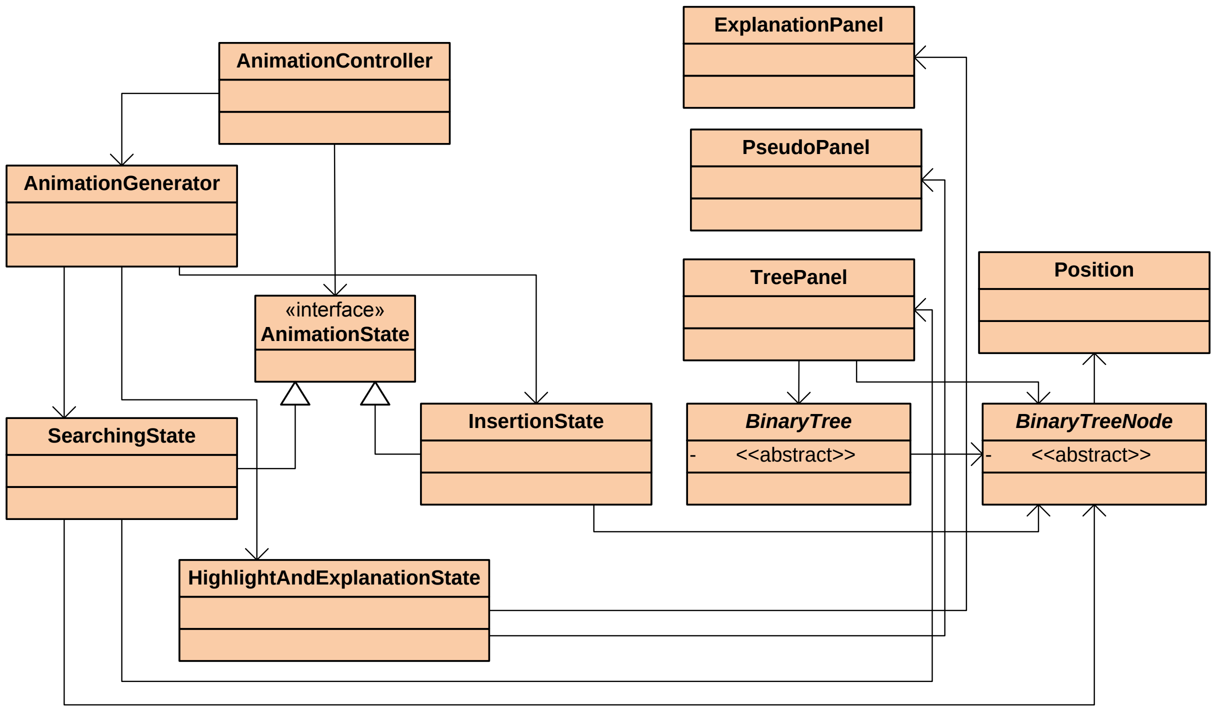


Figure 5.3.6: Class diagram for the Animation Engine

5.4 Implementing the Graphical user Interface

A simple user interface was developed using classes from *Swing* [13], a Java library for developing graphical user interfaces. The initial GUI was developed after the implementation of the Algorithm Simulation in order to check the correctness of the next parts. The implementation of the GUI was based on the design in section 4.6. The classes implemented to represent each identified component of the GUI are listed below.

- Tree selection display → `JPanel` containing buttons for selecting a tree type
- Tree visualization display → `TreePanel`
- Tree operations display → `JPanel` containing buttons for performing tree operations
- Animation control display: `JPanel` containing buttons for controlling the animation
- Pseudo-code display → `PseudoPanel`
- Algorithm explanation comments display → `ExplanationPanel`
- Menu Bar → A `JMenuBar` containing three menu bar items (file, edit, exit)

In order to create a professional GUI we have included an image for each button of the system that it was supplied to the constructor of the `JButton` class. A snapshot of the GUI when the program loads, is shown in figure below.

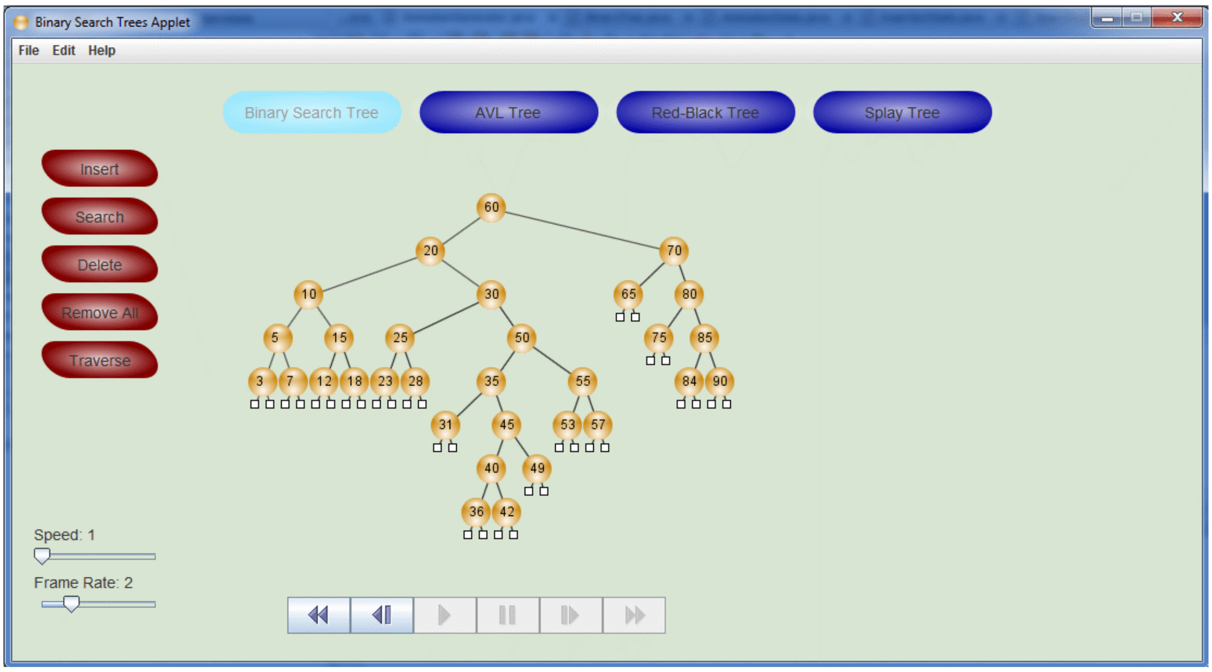


Figure 5.4.1: The Graphical User Interface

During the implementation of the GUI, the usability principles discussed in section 3.5.2 were taken into consideration. An important characteristic was to allow the tool to receive inputs in a friendly way. For this task, a subclass of the `JDialog` class called `OperationDialog` was implemented. This class is loaded and displayed each time the the insert, delete or search buttons are pressed. For example, during insertion, the displayed dialog allows the user either to enter its own number or to allow the `OperationDialog` to generate a random one on its behalf. The implementation of the `actionPerformed` method of the `OperationDialog` that deals with user inputs is shown in figure 5.4.2. The implementation of this method deals with incorrect inputs as characters, a negative number or a number above 99 as this will exceed the area of a node's image.

A similar dialog window was implemented for the tree traversals that allow the user to select a traversal algorithm for the tree. Figures, 5.4.3 and 5.4.4 show how the tool receives input from the user for insertion and tree traversals.

Moreover, the operation buttons on the left hand-side of the tool were implemented in such a way that the operation that takes place at a given time remains highlighted as well as the type of tree in which the operation is performed.

In addition, the animation control buttons are enabled and disabled depending on the remaining animation states in the `next` and `back` stack of the Animation Controller. The pseudo code and algorithm explanation comments are displayed only during the algorithm's runtime. The tool also provides the option to disable/enable the pseudo code and algorithm explanation comments. A snapshot of the final system during insertion is show in figure 5.4.3.

```
public void actionPerformed(ActionEvent e)
{
    if(okButton == e.getSource())
    {
        try
        {
            value = Integer.parseInt(input.getText());
        }
        catch(Exception ex)
        {
            value = -1;
            this.setVisible(false);
            return;
        }
        if(value < 1 || value > 99)
        {
            value = -1;
        }
        this.setVisible(false);
    }
    else if(cancelButton == e.getSource())
    {
        value = -2;
        setVisible(false);
    }
    else if(randomButton == e.getSource())
    {
        input.setText(""+ (random.nextInt(98) + 1));
    }
}
```

Figure 5.4.2: The method of OperationDialog for dealing with user inputs

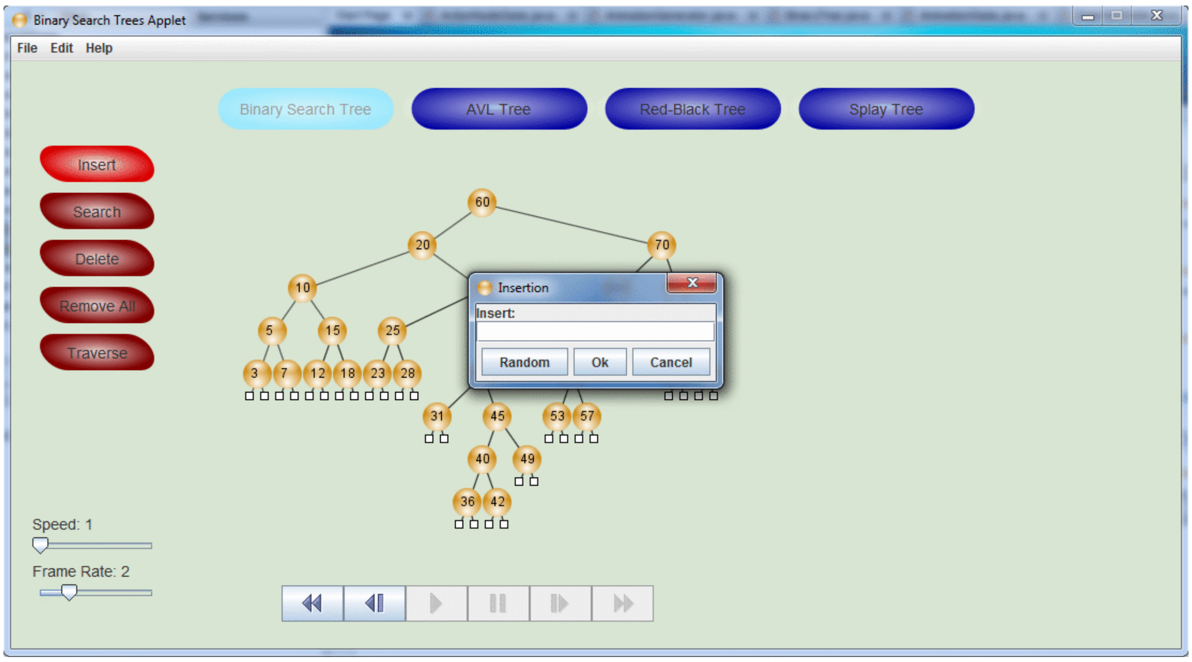
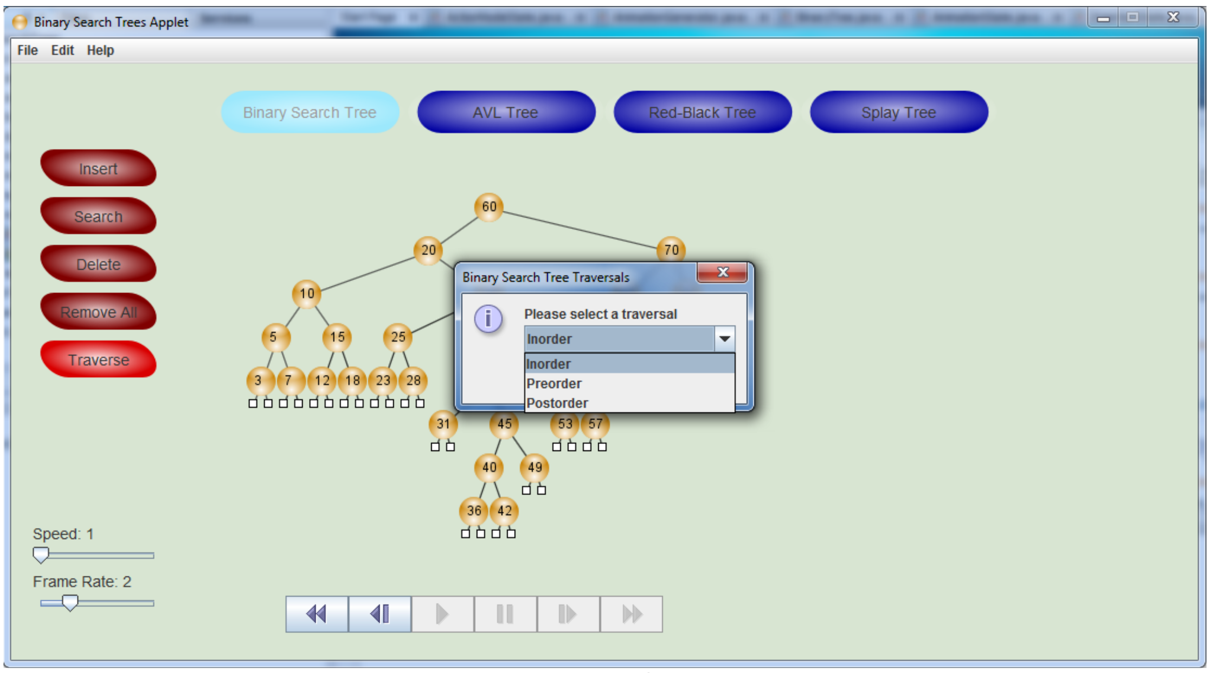


Figure 5.4.3: Input Dialog for Insertion



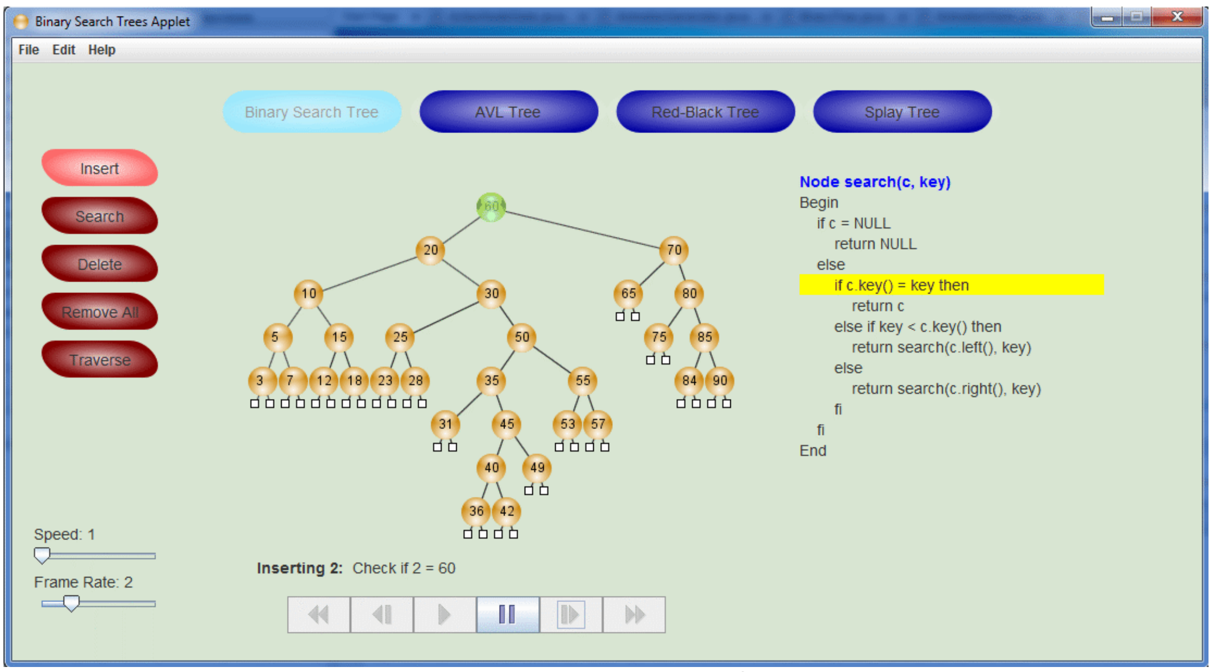
Binary Search Tree Traversals

Please select a traversal

Inorder

Inorder
Preorder
Postorder

Figure 5.4.4: Input Dialog for Tree Traversals



```
Node search(c, key)
Begin
   if c = NULL
      return NULL
   else
      if c.key() = key then
         return c
      else if key < c.key() then
         return search(c.left(), key)
      else
         return search(c.right(), key)
      fi
   fi
End
```

Inserting 2: Check if 2 = 60

Figure 5.4.5: The system during the animation of the insertion algorithm

Finally, a class diagram for the final system is shown in figure 5.4.6. The diagram shows the connectivity of the main classes discussed previously and includes three animation states and a Balanced Binary Tree (Splay).

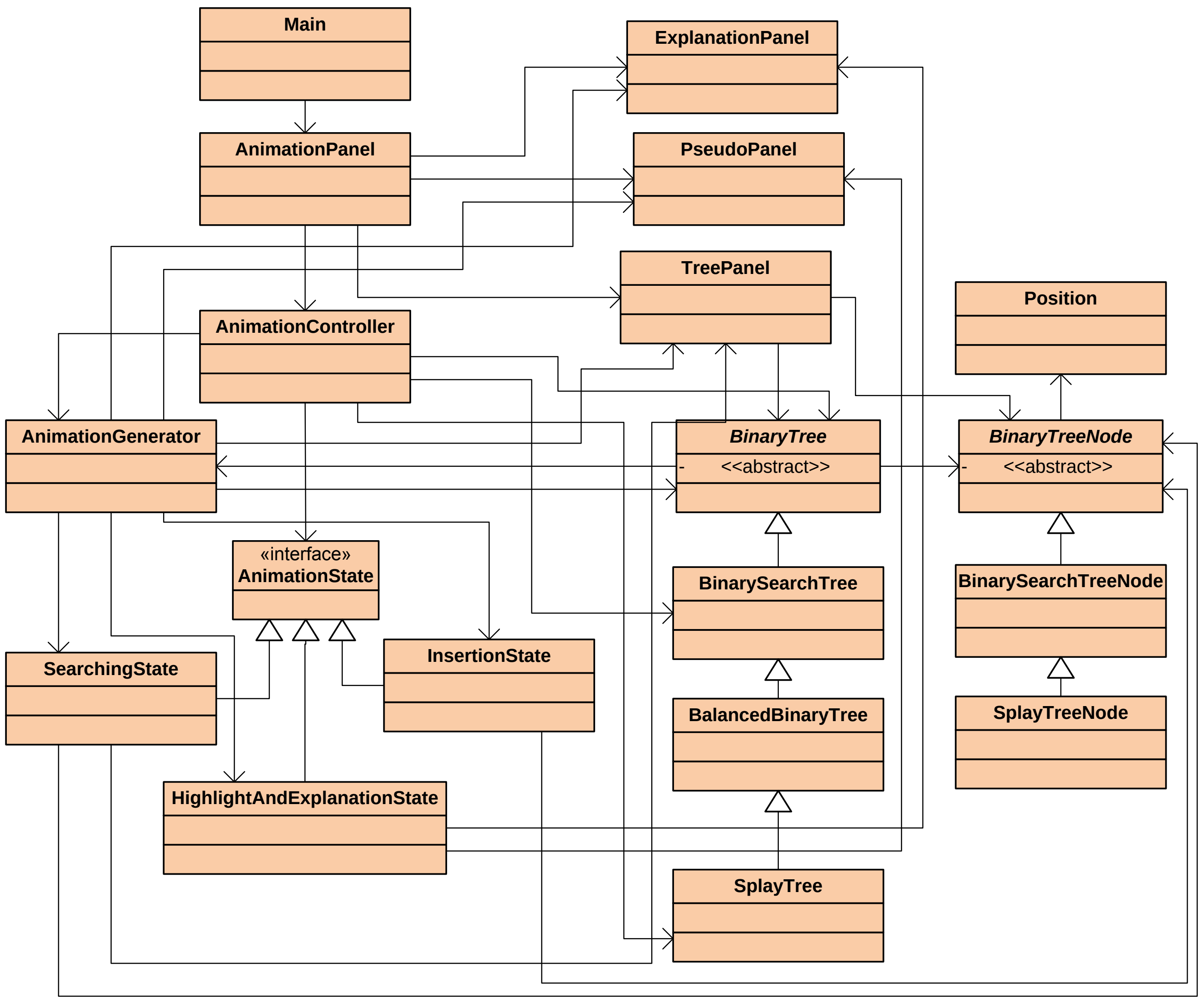


Figure 5.4.6: Class diagram for the final system including 3 animation states and a Balanced Binary Tree

6 Testing

Once the final system is implemented, we have to identify any errors (bugs) that exist in the system. An exhaustive testing will reveal errors that make the system to either fail or operate abnormally. Of course none system is bug-free, so we will try to identify as much errors as possible. The testing structure is based on [29] and consists of five different testing phases: unit testing; integration testing; system testing; acceptance testing; and regression testing. We will focus on the unit and system tests because they are the most important for this project.

6.1 Unit Testing

Unit testing is also known as white-box testing and sets out to test small units of code. In this project, such units are methods of a class implemented in java. During this phase, we try to test whether these methods operate logically correctly and produce the correct output based on a range of inputs.

For this testing we made use of the *JUnit* [24] testing framework of java which provides methods for testing the different units (methods) of a class independently. For more clear outputs, we have temporally included some print statements within the methods. Unit testing was carried out continuously during the development of the tool by testing each method in isolation.

The first method that has to be tested is the insertion method. During this test, we attempt to insert a range of keys into a Binary Search Tree. For each insertion, we check whether the current key was inserted to the right position. The data values chosen are random integers between 1 and 99. The algorithm performs comparisons between two numbers using `compareTo` method. Essentially, if the algorithm works for these values then it should work for any other number. A part that we are interested in is the case where the number inserted already exists in the tree. An example method that uses JUnit test for testing the insertion method of `BinarySearchTree` class is shown in figure 6.1.1.

The data inserted are 50, 25, 75, 25, 65, 70 and 50. As it is observed, 25 and 50 are inserted twice, so we expect this to be detected and a warning message to be printed. The output of this test is shown in figure 6.1.2. According to the insertion algorithm shown in Appendix A2 the keys are inserted to their correct positions apart from 25 and 50 where the algorithm has detected that they are duplicates and ignored them.

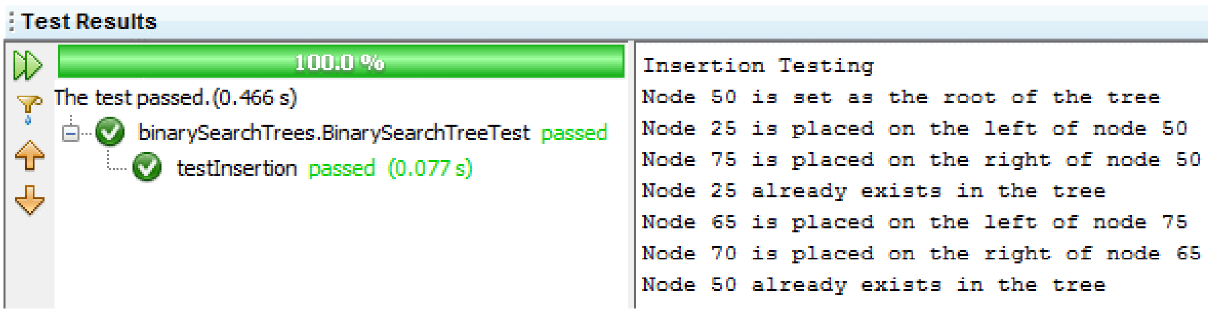
Next, we have tried to test the deletion algorithm. The data chosen for the deletion testing were based on four possible cases listed below.

1. A node that does not exists in the tree
2. A node that has no children
3. A node that has only one child
4. A node that has two children

For this test, we have constructed a random tree using the already tested insertion algorithm and created a similar JUnit test method for the deletion algorithm. Figure 6.1.3 shows the tree constructed for the deletion test.

```
@Test
public void testInsertion()
{
    System.out.println("Insertion Testing");
    BinaryTree tree = new BinarySearchTree();
    Comparable[] numbers = new Comparable[]{50, 25, 75, 25, 65, 70, 50};
    //Insert nodes
    for(int i = 0; i < numbers.length; i++)
    {
        tree.insert(tree.createNode(numbers[i]));
    }
    //Get the nodes of the tree using inorder traversal
    numbers = tree.toArray();
    boolean isSorted = true;
    for(int i = 0; i < numbers.length - 1; i++)
    {
        isSorted = isSorted & numbers[i].compareTo(numbers[i + 1]) < 0;
    }
    //Check if the tree is sorted
    assertEquals(isSorted, true);
}
```

Figure 6.1.1: Testing method for insertion



Test Results
100.0 %
The test passed.(0.466 s)
binarySearchTrees.BinarySearchTreeTest passed
testInsertion passed (0.077 s)

```
Insertion Testing
Node 50 is set as the root of the tree
Node 25 is placed on the left of node 50
Node 75 is placed on the right of node 50
Node 25 already exists in the tree
Node 65 is placed on the left of node 75
Node 70 is placed on the right of node 65
Node 50 already exists in the tree
```

Figure 6.1.2: Output produced by the testing method

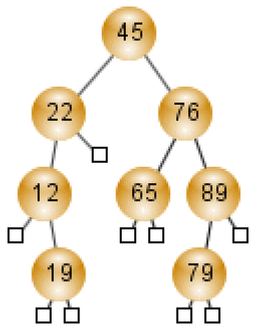


Figure 6.1.3: Binary tree constructed for testing deletion

For the first test case, we tried to delete a node with key 88. The output produced was a warning message showing that node 88 does not exist in the tree. Next, we tried to delete node 19 where the output showed that node 19 was deleted successfully. Then, node 22 was deleted, which has only one child and the results showed that the node 22 was replaced by node 12. Finally, the node 76 that has two children was deleted. As a result, 76 was replaced with its successor 79 and the right child of 79 (in this case a null node) was set as a left child of 89 (79's old parent).

A searching test followed, based on the tree in figure 6.1.3. Here we first searched for nodes that exist in the tree such as 22 and a message "22 is found" was printed. Similarly, for nodes that was not in the tree such as 99 a message "99 is not found" was printed.

Now that all the basic algorithms of the `BinarySearchTree` class are ensured to perform correctly, a unit test for the Balanced Trees has to be carried out. Below there is a list of the test cases considered during the testing of these trees.

AVL tree

- Performed an insertion or deletion that violates the balance value of a node where the difference in height between the left and right sub-trees becomes more than one and tested if the tree is rebalanced

Red-Black tree

- Performed an insertion or deletion that violates the red or black condition as discussed in section 3.1.2 and test if the tree is rebalanced

Splay tree

- Inserted a node and test if it is splayed to the root
- Deleted a node and checked if its parent is splayed to the root
- Searched for an existed node in the tree and tested if it is splayed to the root

All these tests have passed successfully, making us confident to move on to the next testing phase where we will test the integrity of the entire system.

6.2 System Testing

This is a very complex system, so the unit testing could not cover it in all possible ways. During this phase, the system will be tested as a whole. System testing is part of the black box testing where no knowledge about the inner code design and logic is required. Consequently, the system is tested with respect to the outputs generated for a range of different possible inputs. The system testing is divided into five sub-tests discussed next.

6.2.1 Algorithm Tests

The animation is explicitly depended on the states produced during an algorithm execution. Even if the algorithm works correctly, we cannot be sure that the animation states produced will be correct. The aim of this testing is to ensure that the animation states produced perform correctly with respect to the algorithm animation.

The algorithms to be tested first are searching, insertion and deletion for a standard Binary Search Tree. These algorithms need to be tested first because the algorithms of Balanced and Self-Adjusting Trees execute the operations of the Binary Search Tree before they balance their tree. An overview of the algorithm tests for each type of Binary Search Tree and their Appendix location is given in table 4.

Binary Tree	Test Numbers	Appendix Number
Binary Search Tree	BSTI - BSTI12, BSTS1 - BSTS7, BSTD1 - BSTD9, BSTR1 - BSTR15	B1 - B4
AVL tree	AVL1 - AVL8	B5
Red-Black tree	RB1 - RB8	B6
Splay tree	SPL1 - SPL10	B7

Table 4: Binary tree algorithm tests overview

For this testing, we considered all the cases mentioned during the unit testing with the difference that the output should now be a graphical representation of the tree instead of a print statement.

BSTI8 test is an insertion test where node 51 was inserted in a Binary Search Tree. The test has passed successfully and figure 6.2.1 below shows the graphical representation of the tree before and after the insertion of 51.

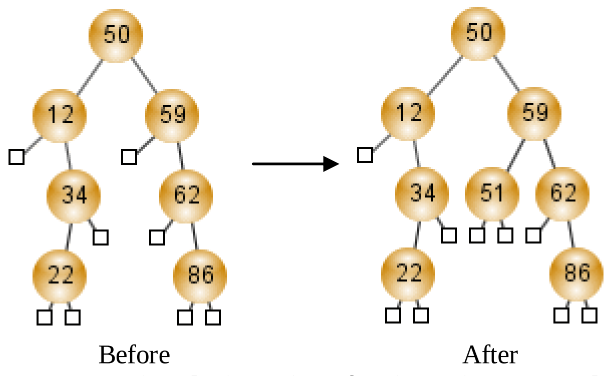


Figure 6.2.1: Testing the insertion of 51 in a Binary Search Tree

A deletion test undertaken was the BSTD3 test where node 50 was deleted from a Binary Search Tree. The figure 6.2.2 below shows the graphical representation of the tree before and after the deletion of 50. During this test, the tool has animated node 50 being swapped with its successor 51 and then falling down the tree.

The searching test was carried out as in the unit test where the output was an animation through the searching path where the searched node was either found or not.

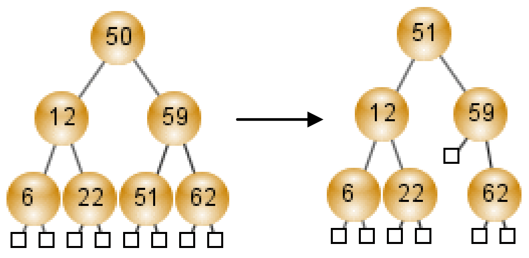


Figure 6.2.2: Testing the deletion of 50 in a Binary Search Tree

For the testing of Balanced Trees, we have considered all the cases that violate their properties (mentioned during the Unit testing) and how a violation is graphically restored.

One of such tests was AVL4 where the inserted node 40 violated the height balance of node 50. Figure below shows the animation states produced in order to restore the height balance of the AVL tree. The expected output was a left rotation of 40 over its parent 25 and then a right rotation of 40 over its new parent 50. Clearly, this test has passed successfully.

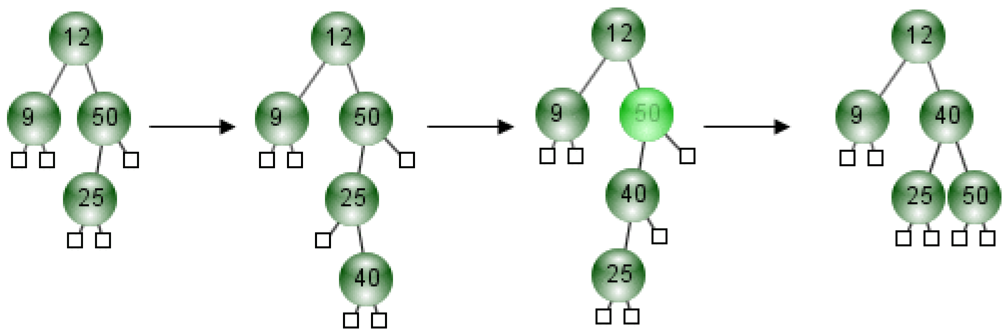


Figure 6.2.3: The rebalance of the AVL tree after the insertion of node 40

During the above tests, there was an additional testing on the currently highlighted line in the pseudo code and the explanation comments displayed during each algorithmic step. In addition, no negative numbers was allowed to enter as well as three digit numbers because this would cause the number to exceed the area of the node. These restrictions prevented by the GUI controller, which will be discussed next. All 69 tests undertaken have passed successfully which means that the algorithms produce the correct animation states.

6.2.2 Animation Menu Tests

The Graphical user interface consists of one display, the animation menu. This menu displays the buttons that control the animation, the pseudo code, the tree operation buttons, the tree visualization display and the tree selection buttons. The 35 tests performed on these components and their Appendix location is given in table 5.

GUI Component	Test Numbers	Appendix Number
Tree Operation Buttons	TOB1 – TOB7	B8
Tree Selection Buttons	TSB1 – TSB4	B9
Animation Control Buttons	ACB1 – ACB6	B10
Pseudo code	PCD1 – PCD4	B11
Tree Visualization Display	TVD1 – TVD4	B12

Table 5: GUI components tests overview

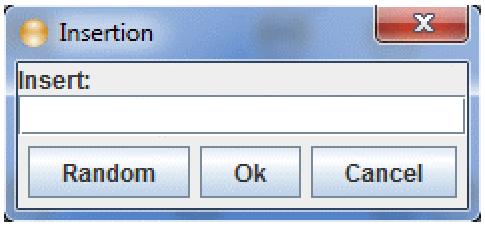
From the 35 tests, 34 passed successfully, but one of them has failed. The table 6 below shows some of the tests to be demonstrated as an example test for each GUI component.

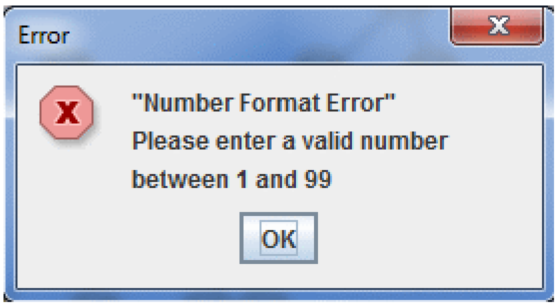
Test #	**Input**	**Expected Output**	**Actual Output**	**Decision**
TOB1.2	An invalid input is entered	A warning window saying that the input must be an integer between 1 and 99	A warning window saying that the input must be an integer between 1 and 99	Pass
TSB3	The Red-Black tree button is	The current Red-Black tree is displayed in the	The current Red-Black tree is displayed in the	Pass

	pressed	tree visualization display and the button is disabled. All the other tree buttons are enabled	tree visualization display and the button is disabled. All the other tree buttons are enabled	
ACB1	The play button is pressed	The animation starts and all the other animation buttons apart the pause are disabled	The animation starts and all the other animation buttons apart the pause are disabled	Pass
PCD4.1	The line of the pseudo code to be highlighted is located off-screen	The display is scrolled to show the highlighted line of code	The display is scrolled to show the highlighted line of code	Pass
TVD4	The area of interest for the current animation exits the size of the tree visualization display	The display is automatically scrolled to the area of interest	The display needs to be scrolled manually	Failed

Table 6: Some of the tests for each GUI component

The testing of the tree operation buttons was to ensure that inputs received from the user are validated and any incorrect inputs should raise a warning explaining the fault. For example, when the user presses the insert button a new window is displayed in the middle of the screen asking the user to either enter an input in the text field or let the program to generate a random on its behalf. Figure 6.2.4 shows the insertion window and the error message appearing during the TOB1.2 test where an invalid input such as "134" and "test" was entered.




Error

"Number Format Error"
Please enter a valid number
between 1 and 99

Figure 6.2.4: Pressing the ok button when an invalid input or no input is inserted

The tree operation buttons testing was to ensure that when the user selects a tree, then that tree is displayed and the button referring to that tree is disabled. The figure 6.2.5 shows the Red-Black Tree button being pressed during the TSB3 test.

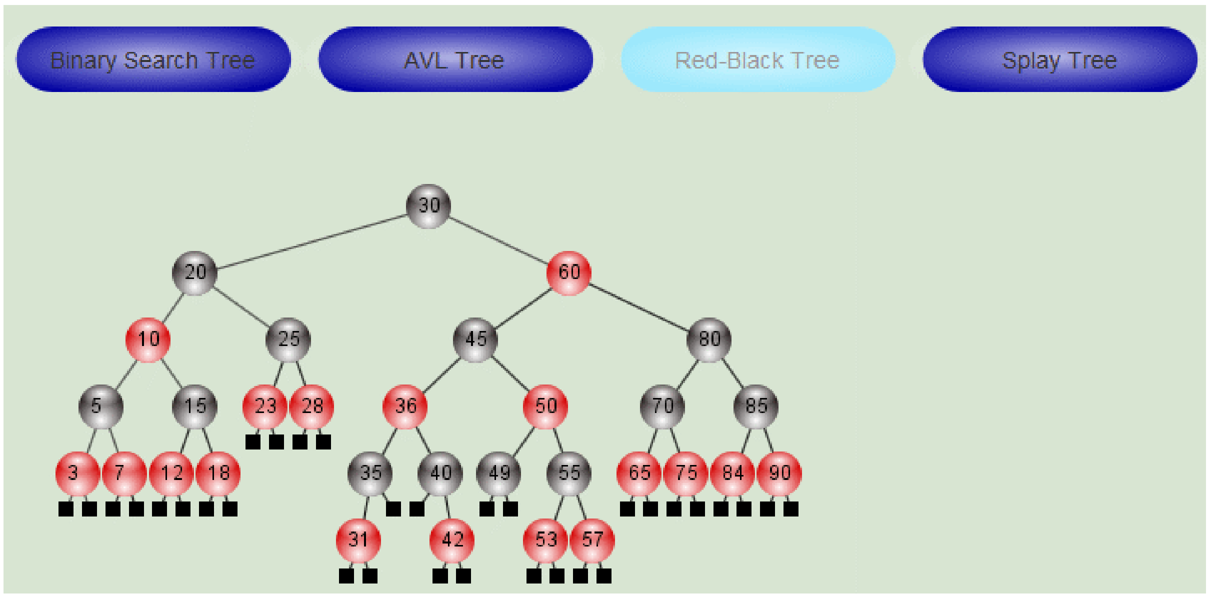


Figure 6.2.5: The Red-Black tree button is pressed

Next, all the tests on the buttons that controls the animation were succeeded. Figure 6.2.6 shows the effects on the buttons when play is pressed during ACB1 test. The output produced is as it was expected so this test passes.

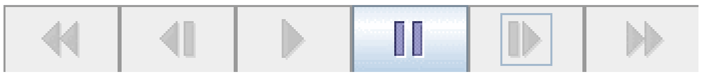
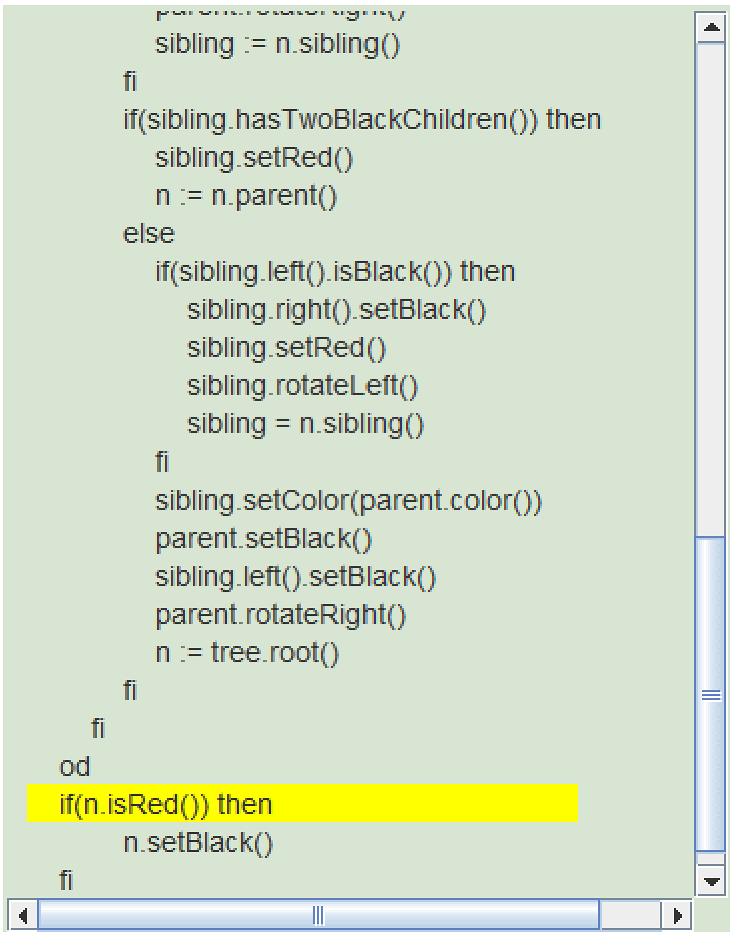


Figure 6.2.6: Buttons when play is pressed

Next, the pseudo code has been tested to ensure that it is displayed correctly and the highlighted line is visible to the user. The PCD4.1 test was performed on a Red-Black Tree during deletion rebalance. The reason was because this pseudo-code is too large and some lines of the code are located off the pseudo-code screen. The results of this test are shown in figure 6.2.7 and they show that the pseudo-code display is scrolled to show the off-screen line of code.

Finally, the tree visualization tests attempted to make sure that the tree is displayed correctly and areas of interest are visible to the user during the animation, even if the tree was becoming larger than the display (TVD4 test). This test was performed on the standard Binary Search Tree where all nodes were inserted in ascending order. This caused the tree to act as a linked list and some nodes exited the tree display. Unfortunately, this test has failed because the display was not automatically scrolled to show the insertion of a node whose parent was located off the screen.



```
            sibling := n.sibling()
        fi
        if(sibling.hasTwoBlackChildren()) then
            sibling.setRed()
            n := n.parent()
        else
            if(sibling.left().isBlack()) then
                sibling.right().setBlack()
                sibling.setRed()
                sibling.rotateLeft()
                sibling = n.sibling()
            fi
            sibling.setColor(parent.color())
            parent.setBlack()
            sibling.left().setBlack()
            parent.rotateRight()
            n := tree.root()
        fi
    fi
od
if(n.isRed()) then
    n.setBlack()
fi
```

Figure 6.2.7: The pseudo code display is scrolled to show the highlighted line

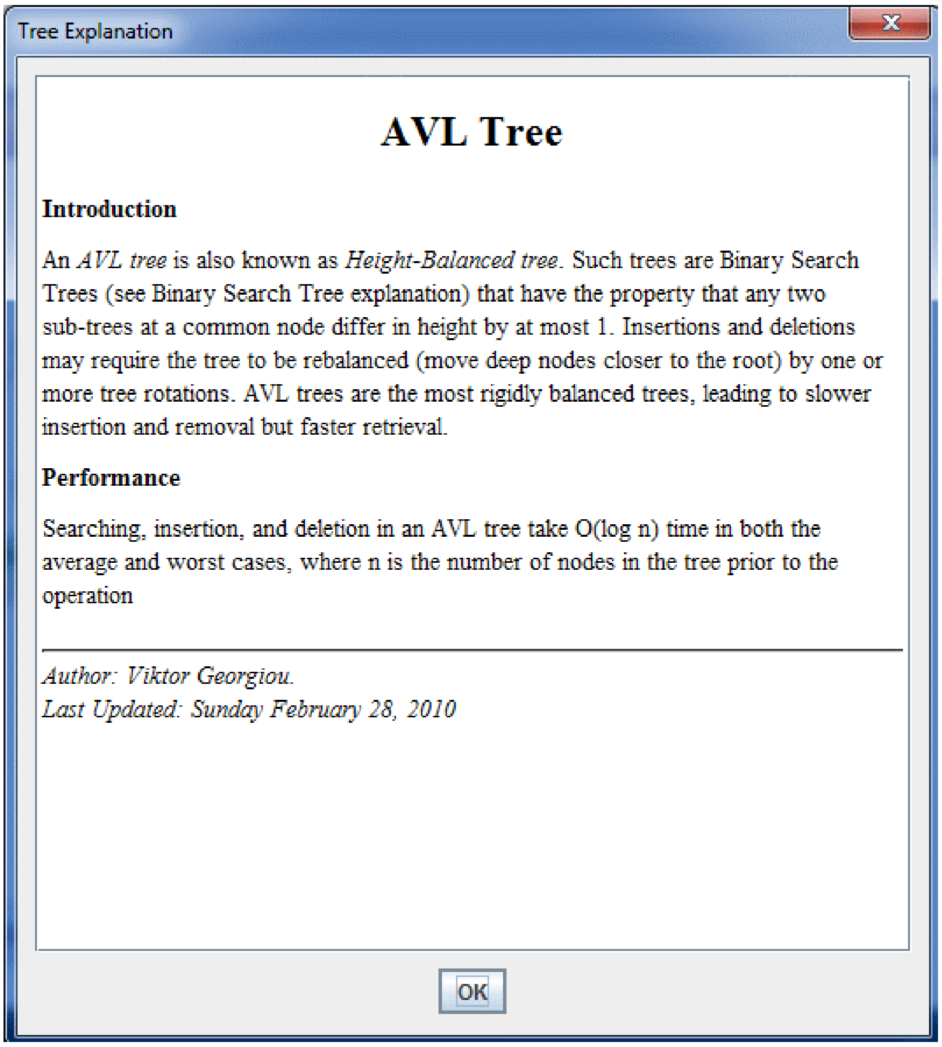
6.2.3 Menu Bar Tests

The menu bar of the tool provides additional options to the tool such as:

- Terminate the program
- Disable pseudo code and explanation comments
- Node colour change
- Change the way the tree enters the screen
- Help support

The tests MB1 – MB9 in Appendix B13 were carried out to address the correctness of these options. The output produced during the MB9 test is shown in figure 6.3.5 where an explanation window for the current tree (AVL in this case) is displayed when the user clicks on Help → Tree Explanation

Finally, the MB2.1 test shown in table 7 has failed. This happened because when the tool was extended to allow the pseudo code and explanation comments to be disabled, it resulted to some animation control buttons to malfunction. In particular, the step forwards and step backwards were not disabled when the animation was reaching the final and initial state respectively. They needed to be pressed one more time. This was not a major problem because it did not affect the internal structure of the tool.


Tree Explanation

AVL Tree

Introduction

An *AVL tree* is also known as *Height-Balanced tree*. Such trees are Binary Search Trees (see Binary Search Tree explanation) that have the property that any two sub-trees at a common node differ in height by at most 1. Insertions and deletions may require the tree to be rebalanced (move deep nodes closer to the root) by one or more tree rotations. AVL trees are the most rigidly balanced trees, leading to slower insertion and removal but faster retrieval.

Performance

Searching, insertion, and deletion in an AVL tree take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation

Author: Viktor Georgiou.
Last Updated: Sunday February 28, 2010

Figure 6.2.8: Tree explanation window

Test #	Input	Expected Output	Actual Output	Decision
MB2.1	Check that animation control buttons work well when explanations are disabled	The buttons functionality is as in ACB1 – ACB6	The step forwards and step backwards are not disabled when reach the final and initial state respectively. They need to be pressed one more time.	Fail

Table 7: Menu Bar Test

7 Evaluation

The evaluation of the system is one of the most significant phases of this project because it will assess whether the right system is built. During this section, we first discuss about the testing results obtained from the previous chapter and the feedback received from second and third year students. Finally, we evaluate each individual requirement based on the testing results and the feedback received.

7.1 Testing Results

The results obtained during testing were very positive, giving a great confidence about the effectiveness of the tool. From the 114 tests, only two have failed. The test MB2.1 has failed because the support for disabling and enabling the pseudo code and explanation comments was not part of the design. As a result, the tool was implemented without such a feature. To provide this functionality some methods had to be modified. In particular, the `stepForwards` and `stepBackwards` methods of the `AnimationController` class were overloaded with additional if statements. Consequently, the tool became more complicated and the solution for this error had an impact to the tests ACB2, ACB5 and ACB6. As these tests were crucial and due to time constraints this error remained unresolved.

Similarly, the tree visualization display test TVD4 has failed because it has not predicted during the design that the nodes might exit the screen area. The system has been totally built from scratch, without using any library source that could probably resolve this problem. An attempt was made to fix this error based on its similarity with test PCD4.1. Unfortunately, this attempt was unsuccessful because the pseudo code panel was using `JLabel` for each line of code which had a build in method for getting the exact bounds of the line (`label.getBounds()`). Even if the bounds of the tree were obtained successfully, it was not possible to get the exact area bounds where the animation was running. Even though, the problem had partially resolved by adding a scrollbar to the tree display so that the user could scroll to the area of interest manually.

All the other tests passed successfully and in particular, no errors have been found in the algorithm animations. On the other hand, the testing has been thoroughly carried out by the developer without any other additional support and under time pressure. For this reason, we cannot be certain that no other errors exist in the system because testing cannot cover all the possible cases. In particular, some algorithms are very complicated and consist of a large range of various states; some of them probably were not covered exhaustively.

Finally, from the testing results we can be confident that the tool operates correctly and effectively. This proves that the design phase was very important and has indeed formed the shape of the tool.

7.2 User feedback

A part of the system's evaluation was based on the feedback received from users. Taking into consideration some of the evaluations techniques discussed in section 3.6 during the background research we concluded that the most effective method was to conduct an online survey.

A small questionnaire has been designed using the university's online *ISS Form Builder* [27], and consisted of 11 questions shown in Appendix D. The questions were simple with five possible answers each. The first eight questions were about the rating of the tool based on users understanding, the usability of the tool, the animation effects and control, the pseudo code importance and finally the tool's recommendation to other users. The last three questions were optional, regarding the personal thoughts of the users such as what the liked less/most and any future recommendations. The reason we made these questions optional was to not force users to write something down, as many of them could probably skip the survey leaving the questionnaire unanswered.

Next, a web page [14] has been set up where we placed the tool converted as a java Applet along with a user manual (shown in Appendix C) and a link to the online survey. The web page link was then distributed to a set of students in Computing Science course from the second and third years who have been asked to evaluate the system. The students should had completed the "CSC2016: Algorithm Design and Analysis" module and have a basic knowledge in Binary Search Trees. Although, students were not taught about Balanced and Self-Adjusting Trees, as a result the evaluation was mainly focus on the standard Binary Search Trees.

The feedback received was limited because only 21 participants have answered the questionnaire. Even though, valuable information has been derived. A summary of the results is shown in the bar chart in figure 7.2.1 below where the maximum rating for each answer is 5.

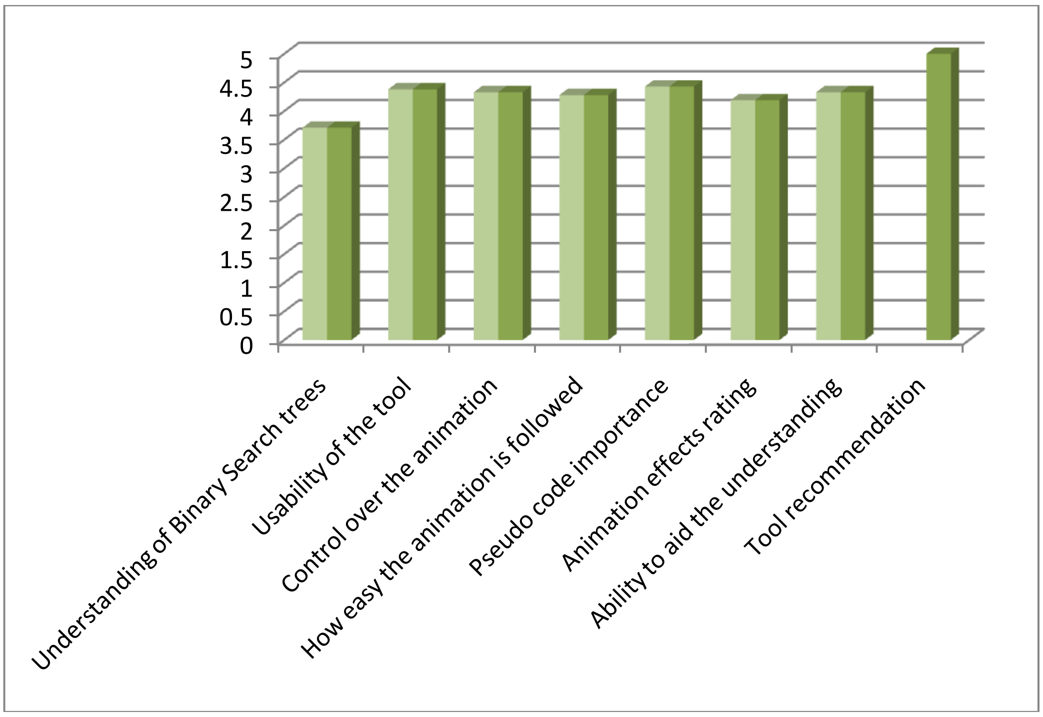


Figure 7.2.1: Results of the questionnaire's feedback

It has been observed that no answer received was below 3, the average rating. In addition, the summary of each question regarding the usability, effectiveness and understanding aid of the tool had an overall rating of above 4 which is a very positive feedback. In particular, we received a 100% rating from the 8th question asking whether they will recommend the tool to other users who want to learn about Binary Search Trees.

Unfortunately, not many users have answered the last three questions that were asking about their personal thoughts. Even though, some good ideas have been suggested. Some of them were the addition of new types of Binary Search Trees and automatic start of animation when an input is inserted instead of pressing the play button. Most users liked most the animation effects of the tool and the link to the pseudo-code. Moreover, a student identified the error on the tree visualization where the display was not scrolled automatically when the animation was running off the screen. The analytical feedback received by each individual student along is shown in Appendix D.

Finally, their ideas have been seriously taken into consideration for the tool's improvement. Based on the survey results, we can be very confident and believe that we have developed the right tool that aids the understanding of Binary Search Trees, which was the aim of this project.

7.3 Project Requirements

We now evaluate whether the requirements have been successfully met. The list of the requirements and a discussion for each one is given below.

Requirement 1: *The tool should provide animations for four types of Binary Search Trees.*

The tool provides animations for the key types of Binary Search Trees as standard Binary Search Trees, AVL, Red-Black and Splay trees so we can assess that this requirement is met.

Requirement 2: *It should animate the key algorithms including searching, insertion and deletion*

The tool provides the ability to select which algorithm to animate. As a result, all of the algorithms addressed to this requirement were successfully implemented using animation effects. In addition, we have provided some extra algorithms for tree traversals.

Requirement 3: *A pseudo-code should be displayed for each algorithm*.

When a tree algorithm starts, its pseudo-code is displayed on the screen. In particular, the tool displays each algorithm's pseudo-code that is currently executed. For example during insertion, the searching pseudo-code is displayed for finding the parent of the node to be inserted. Next, the searching pseudo-code is cleared and the insertion pseudo-code is displayed. Based on these facts we can support that the pseudo-code requirement is met.

Requirement 4: *The tool should provide clear animation based on a range of animation effects*

The tool provides a range of animation effects as highlighting the node of interest and moving smoothly from one node to another during searching. Particularly, each time a node moves from one node to another it jabs the node that it left from, presenting a clearer and more advance animation. During insertion, the node is slowly moving from its parent to its final position, noticeably showing how insertion is performed. Moreover, a deleted node starts falling from the tree in a very soft way. Finally, the frame rate of the animation can be controlled from the user interface. If animation is not clear enough, the user simple reduces the value of the frame rate and the animation runs smoother using more frames per second.

Requirement 5: *It should provide smooth transitions from one algorithm state to another.*

This requirement is similar to the previous one. A clear animation constitutes to a smooth transitions from one state to another. We are now confident that this requirement is met.

Requirement 6: *The animation should be linked to the pseudo-code.*

In addition to the pseudo-code display, during each algorithmic step, the associated line of code is highlighted in order to help the user understand the algorithm. The tool also provides the ability to enable or disable the link to the pseudo-code display. Moreover, many users have expressed their interest in this feature, mentioning that is very helpful. Apparently, this verifies that this requirement has been met successfully.

Requirement 7: *Explanation comments should be displayed for each step of the algorithm during the animation.*

In addition to the linking of the pseudo-code to the animation, the tool displays explanation comments for each step of the algorithm that is in execution. These comments explain the line of code that is executed and are linked to the animation. Consequently, this requirement is met.

Requirement 8: *The tool should provide full control over the animation including step backwards.*

Control over the animation was a recommended feature for good animations learnt during section 3.2.3. Apparently, the tool provides six animation control buttons, the play, pause, step backwards, rewind, step forwards and fast forwards buttons. All these buttons have been tested in detail and it turned out that they function very well. Therefore, the tool provides full control over the animation and in particular step backwards which was a more advance feature.

Requirement 9: *The tool should provide a friendly graphical user interface*.

In section 3.5, we discussed about the interface usability guidelines as learnability, flexibility and robustness. The learnability property is applied when the tool is loaded, which is displayed to the user allowing him to begin an effective interaction with the system. Next, the flexibility property is met as the tool's interface provides clear buttons for the animation control, tree selection and the tree operations. In particular, the tree operation buttons as insert, delete and searching display an additional window with texts explaining exactly what should be done, allowing the user to exchange information easily. Finally, the robustness property is enforced during the animation as it attracts the user's attention. All the control buttons except the pause are disabled and the appropriate lines in the pseudo-code display are highlighted, showing that the animation is in execution. When the animation is finished, the rewind and step backwards buttons become enabled and the pseudo-code display is cleared. This indicates to the user that the animation is finished, and gives him the option to restart the animation. It is supported that the usability principles have been met, thus, it can be verified that a friendly graphical user interface has been developed.

Requirement 10: *The tool should allow two or more Binary Search Trees to be compared.*

When a user operates on a tree, it can easily switch to another type of tree without having to perform all the operations performed to the previous one in order to compare it with the new one. The newly selected tree provides the ability to restart the last operation performed on the previous tree. This allows the user to observe an operation being applied to any type of tree. For example, a user might insert the node 34 into an AVL tree. After or during this operation, he can switch to another tree as Red-Black. When this is done, the Red-Black tree will contain all the nodes inserted in the previous AVL tree (excluding deleted nodes), including node 34. The user is now able to restart the animation and watch node 34 being inserted in a Red-Black tree. This allows a tree constructed from a number of insertions and deletions to be transformed to any other type of tree that is supported by the tool allowing the user to compare the different types of Binary Search Trees. A further improvement for this feature is to allow the trees to be compared in parallel and not sequentially, during an algorithms execution. For this reason, it is stated that this requirement is partially met.

Requirement 11: *The tool should be extendable, allowing new Binary Search Trees to be easily added.*

In figures 5.1.1 and 5.1.2 in chapter 5 we have shown the inheritance of the Binary Trees and nodes. This allows a new type of Binary tree to be easily added to the tool by extending the `BinarySearchTree` or `BalancedBinaryTree` classes. Similarly, for the node of the new tree, the `BinarySearchTreeNode` node is extended. Next, the unique methods of that tree and its node need to be implemented and augmented along with the `AnimationState` classes to represent its animation states. For example, during the implementation of Red-Black tree we have extended the `BinarySearchTreeNode` and `BalancedBinaryTree` classes. Next, we implemented the `insertionRebalance` and `deletionRebalance` methods of the `BalancedBinaryTree` class. The uniqueness of Red-Black trees are the colour flips so we have implemented a new class called `ColourFlipState` to represent the animation state where a node's colour is switched from red/black to black/red. Next, we added augmented code for creating the animation states making use of the already existed `RotationState` class and other classes representing relevant animation states as `HighlightAndExplanationState`.

Requirement 12: *The tool should provide a description of each Binary Search Tree.*

Finally, the menu bar of the tool provides support for displaying a description of the tree that is currently selected. The user simply goes to `Help → Tree Explanation` and a description about the tree, its properties and the best, worst and average case performances is displayed.

8 Conclusion

8.1 Overview

The aim of this project was to create a tool that would use animation effects to support a range of algorithms for different types of Binary Search Trees and assist their learning. A set of objectives and requirements has been laid out to form the foundations for this project. Next, a background research was undertaken for important information about the problem domain to be collected and studied in detail. Subsequently, the structure of the tool development was divided into three parts of the software engineering process, the design, the implementation and testing. The design part has formed the shape of the project whereas the implementation has brought the project into life. The testing phase followed the implementation and tried to uncover and rectify any errors found in the tool. Finally, the final system has been evaluated in order to assess the correctness of the tool according to the initial requirements.

8.2 Aims and Objectives

The aims and objectives defined during chapter 2 were the foundation for the development of this project. In the following discussion, we take each individual objective and assess whether it is fulfilled.

Objective 1: *To identify the key types of Binary Search Trees that should be included in the tool.*

During background research, a range of different type of Binary Search Trees has been identified. We have concentrated on the most popular trees found on books and existing systems, such as AVL, Red-Black and Splay. Finally, the identified Binary Search Trees and their associated algorithms have been successfully incorporated to the tool.

Objective 2: *To identify how animation can be used to aid the understanding of Binary Search Trees.*

Animation was another important part of the project. First, we have learned what the animation is. After an extensive research, we became familiar with a range of animation techniques that could be applied to the tool. Such techniques revealed the need of the animation to be clear with smooth transitions from one state to another. In addition, animation could enforce the understanding by highlighting areas of interest, having control over the animation and include sound effects. All of the acknowledged techniques have been carefully applied to the tool with the exception of sound due to time constraints. However, the testing results derived from the animation testing accompanied with the positive feedback received from users addressing the rating of the animation have verified our resolve to provide animations that indeed aid the understanding of Binary Search Trees.

Objective 3: *To identify the functional requirements of the animation tool*

The requirements of the project were the most important part. The design and implementation have been developed upon the identified requirements. After a careful consideration for the functionality of the tool to be developed, we have concluded to a list of significant requirements for the project. In chapter 7 it was verified that all the requirements but one have been successfully met.

Objective 4: *To develop a prototype animation tool for Binary Search Trees.*

The tool development was entirely concentrated on the first three objectives discussed above. Once these objectives had been met, the design and implementation followed in order to bring the tool to life. Obviously, the final tool verifies that this objective has been successfully met.

Objective 5: *To evaluate the effectiveness of the tool*

Finally, the objectives chain ends to the evaluation of the tool. The evaluation procedure was carried out in terms of different test cases, the feedback received from users and personal thoughts. Firstly, the testing results have proved that the tool is reliable and provides all the functionality addressed in the requirements. Moreover, an evidence for the effectiveness of the tool was obtained during the survey that was conducted. The results collected from the users have shown us that the tool is effective and helps users to understand Binary Search Trees better. Unfortunately, users had knowledge of standard Binary Search Trees only, so the evaluation of the other types of trees was solely carried out by the developer.

8.3 Summary

A set of aims and objectives has been initially set to address the problem domain of this project, which was to develop an animation tool that will help users to understand the algorithms associated with Binary Search Trees.

As a starting point, a background research was undertaken in order to study in detail a range of different types of Binary Search Trees. Many difficulties were encountered while we tried to understand the algorithms of the Balanced Trees that were unfamiliar. In addition, during the research we looked into some animation techniques that would help us to incorporate animation effects in the tool. Next, we looked into some existing tools for Binary Search Trees and evaluated them. A big observation about those tools was that only one of them had full control over the animation. The chosen programming language was another important aspect. For this reason we compared four different programming languages based on the visualization features, portability and extendibility, and we concluded that Java was the most suitable. The final subject during the background research was the evaluation techniques to be used after the tool would be finished.

After the background research, a high-level design of the system was created that divided the system into smaller pieces. A key decision made here was the separation of the algorithm logic from the animation logic in order to provide control over the animation. This decision was critical and made the implementation of the system that followed next, much easier. The idea of Tree Visualization Engine was another key decision made during the implementation because it simplified the functions of the animation states and resulted to a very smooth animation effects.

Throughout testing, we attempted to uncover errors by first testing each method in isolation and then the system as a whole. At this phase, two tests have failed and the failure of one of them was based on the design of the Tree Visualization Engine. This has highlighted the importance of the design process during the development of the tool, as it was the stage where the foundations of the tool were constructed. Unfortunately, testing was exclusively carried out by the developer so we cannot be convinced that all the possible test cases have been covered.

Finally, for the evaluation of the system we decided to conduct a survey. A questionnaire was designed and distributed to students in the Computing Science. A limitation here was that the students were unfamiliar with AVL, Red-Black and Splay trees. As a result, the evaluation from the users was mainly focused on the standard Binary Search Trees. The feedback received was very positive and made us confident about the correctness of the final tool and its ability to aid the understanding of Binary Search Trees. This was the main aim of this project and it is evaluated that this aim and the objectives defined were successfully met.

8.4 Future Work

The software engineer process states that a system should never be published and then left unattended. A range of possible improvements and new features can be incorporated to the tool. Some of such refinements have been identified during and after the development of the tool, but they have not been applied. Some of them are stated in the following discussion.

8.4.1 Automatic scrolling support for the tree visualization

The testing uncovered a problem regarding the visualization of the tree. When the tree was becoming very large, some nodes exited the size of the screen. A scrollbar for the tree display was included but when the animation was running on these nodes, the user was unable to observe them and had to scroll the display manually. Correcting this problem to provide automatic scrolling during the animation would be a crucial improvement for the tool.

8.4.2 Supporting two types of trees to be compared in parallel

This feature will be a refinement of the Requirement 10. It would be a good idea to provide the user with a display that will animate operations for any two selected Binary Search Trees in parallel. Users will be able to observe the changes performed by an operation on both two types of trees simultaneously and clearly see their final structure after insertions and deletions. For example, users will notice that in the worst-case performance, where elements are inserted in ascending or descending order, a standard Binary Search Tree acts as a linked list whereas an AVL and Red-Black tree remain balanced.

8.4.3 Supporting new types of Binary Search Trees

The Requirement 12 was to ensure that the tool is extendable, allowing new types of Binary Search Trees to be easily added. It would be a good idea to exploit this feature and add new types of Binary Search Trees such as Treaps.

8.4.4 Adding more animation effects

The tool includes a wide range of animation effects but it can be extended to include some more. For example, when a node to be deleted is swapped with its successor, instead of just swapping their key, we can add an animation effect showing both nodes to start moving simultaneously until they reach to each other's position. Another example would be to animate a Red-Black node when is changing its colour instead of a direct colour flip from red to black and vice versa. Finally, sound effects would be a good idea to be added since they would make the animation more realistic.

8.4.5 Supporting types of M-way searching trees

M-way search trees are trees that contain N sub-trees and N-1 keys, where 2<= N <= M for some fixed value of M >= 2 (A Binary Search Tree has M equal to 2). Unfortunately, the tool was designed to support the addition of Binary trees, which means that the implementation restricts the number of children for each node to two. The idea is to implement a new class for M-way search trees where it will be extended by the `BinaryTree` class. Similarly, we create another class to represent an M-way search tree node that should be extended by the `BinaryTreeNode` class. This feature requires the whole structure of the project to be modified, including the tree visualization. Fortunately, the modifications will not be too difficult because we only have to generalize the code and change some algorithms to support M children for each node instead of at most two.

8.4.6 Allowing the user to modify the data structure

TRACLA 2 (discussed in 3.4.4) provides to user the ability to modify the data structure. For example when a node is to be inserted, it is inserted by the user instead of the tool. The tool simply checks the position of the inserted node and informs the user whether the node has been inserted correctly. Having this feature will definitely improve the understanding of Binary Search Trees.

References

1. *Binary Search trees Applet*. [cited 2009 18/11/2009]; Available from: http://people.ksp.sk/~kuko/bak/index.html.
2. Adobe Systems., *Adobe Flash CS3 Professional*. Classroom in a book. 2007, Berkeley, Calif.: Adobe Press. v, 339 p.
3. Anderson, P. and G. Morgan, *Developing tests and questionnaires for a national assessment of educational achievement*. National assessments of educational achievement v. 2. 2008, Washington, DC: World Bank. xvii, 168 p.
4. Arnold, K., J. Gosling, and D. Holmes, *The Java programming language*. 4th ed. 2006, Upper Saddle River, NJ: Addison-Wesley. xxviii, 891 p.
5. Black, P.E. and National Institute of Standards and Technology (U.S.), *Dictionary of algorithms and data structures*.
6. Bucknall, J., *The Tomes of Delphi : algorithms and data structures*. Wordware Delphi developer's library. 2001, Plano, Tex.: Wordware Publishing. xviii, 525 p.
7. Cormen, T.H., *Introduction to algorithms*. 2nd ed. 2001, Cambridge, Mass.: MIT Press. xxi, 1180.
8. Diehl, S., *Software visualization : visualizing the structure, behaviour, and evolution of software*. 2007, Berlin ; New York: Springer. xii, 187 p.
9. Dittrich, J.-P. *DSN : Data Structure Navigator*. [cited 2009 19/11/2009]; Available from: http://dbs.mathematik.uni-marburg.de/research/projects/dsn/.
10. Dix, A., *Human-computer interaction*. 3rd ed. 2004, Harlow, England ; New York: Pearson/Prentice-Hall. xxv, 834 p.
11. Drozdek, A., *Data structures and algorithms in C++*. 2nd ed. 2001, Pacific Grove, CA: Brooks/Cole. xviii, 650 p.
12. Drury, D.W., *The art of computer programming*. 1st ed. 1983, Blue Ridge Summit, Pa.: Tab Books. viii, 303 p.
13. Fischer, P., *Introduction to graphical user interfaces with Java Swing*. 2005, Harlow, England ; New York: Addison-Wesley. x, 306 p.
14. Georgiou, V. *Animating Binary Search trees*. [cited 2010 20/04/2010]; Available from: http://homepages.cs.ncl.ac.uk/victor.georgiou/.
15. Gosling, J., *The Java language specification*. 3rd ed. Java series. 2005, Upper Saddle River, NJ: Addison-Wesley. xxxii, 651 p.
16. Haase, C. and R. Guy, *Filthy rich clients : developing animated and graphical effects for desktop Java applications*. The Java series. 2008, Upper Saddle River, NJ: Addison-Wesley. xxvii, 572 p.
17. Helsinki University of Technology. *TRAKLA 2*. [cited 2009 20/11/2009]; Available from: http://www.cse.hut.fi/en/research/SVG/TRAKLA2/.
18. Hershberger, M.H.B.a.J., *Color and Sound in Algorithm Animation*. 1991.
19. Java Models. *Binary Search trees Applet*. [cited 2009 18/11/2009]; Available from: http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm.
20. Kernighan, B.W. and D.M. Ritchie, *The C programming language*. 2nd ed. 1988, Englewood Cliffs, N.J.: Prentice Hall. xii, 272 p.
21. Kernighan, B.W. and D.M. Ritchie, *The C programming language*. Prentice-Hall software series. 1978, Englewood Cliffs, N.J.: Prentice-Hall. x, 228 p.
22. Knuth, D.E., *The art of computer programming*. 2005, Upper Saddle River, NJ: Addison-Wesley. v. <v.1, fasc. 1, v. 4, fasc. 0-4>.
23. Lafore, R., *Data structures & algorithms in Java*. 2nd ed. 2003, Indianapolis, Ind.: Sams. xix, 776 p.

24. Link, J. and P. Fröhlich, *Unit testing in Java : how tests drive the code*. 2003, San Francisco, Calif.: Morgan Kaufmann. xvii, 376 p.
25. Macromedia Inc., *Macromedia Flash MX 2004*. 2003, Macromedia: San Francisco, CA.
26. Morris, J., *Algorithm Animation: Using algorithm code to drive an animation*, The University of Auckland.
27. Newcastle University. *ISS Form Builder*. [cited 2010 10/04/2010]; Available from: http://forms.ncl.ac.uk/.
28. Parent, R., *Computer animation : algorithms and techniques*. 2nd ed. 2008, Amsterdam ; Boston: Elsevier/Morgan Kaufmann. xviii, 593 p.
29. Pressman, R.S., *Software engineering : a practitioner's approach*. 6th ed. 2005, McGraw-Hill Higher Education: Boston. xxxii, 880 p.
30. Rudolf Fleischer, L.K., *Algorithm Animation for Teaching*, Charles University: Prague.
31. Schildt, H., *C++, the complete reference*. 2nd ed. 1995, Berkeley: Osborne McGraw-Hill. xxi, 671 p.
32. Sears, A. and J.A. Jacko, *Human-computer interaction. Fundamentals*. Human factors and ergonomics. 2009, Boca Raton: CRC Press. xv, 331 p.
33. Watson, K., *Beginning Microsoft Visual C# 2008*. 2008, Indianapolis, IN: Wiley Pub.
34. Wiener, R. and L.J. Pinson, *Fundamentals of OOP and data structures in Java*. 2000, Cambridge, [England] ; New York: Cambridge University Press. xv, 463 p.

Appendices

Appendix A: Binary Search Tree algorithms pseudo-code

A1. Searching Algorithm

```
Algorithm: search
Inputs: node: Pointer; key: Data
Returns: node: Pointer

search(node, key)
Begin
    if (node = NULL)
        return NULL
    else
        if (node.key = key) then
            return node
        else if (key < node.key) then
            return search(node.leftChild, key)
        else
            return search(node.rightChild, key)
        fi
    fi
End
```

A2. Insertion Algorithm

```
Algorithm: insert
Inputs: node: Pointer; key: Data
Returns: isInserted: Boolean

insert(key, node)
Begin
    if(node = NULL)
        node = new binaryNode(key)
        return true
    else
        if (node.key = key)
            return false
        else if (key < node.key)
            return insert(node.leftChild, key)
        else
            return insert(node.rightChild, key)
        fi
    fi
End
```

A3. Deletion Algorithm

```
Algorithm: delete
Inputs: tree: BinaryTree; node: Pointer;

delete(tree, node)
Begin
    toBeDeleted = NULL
    newChild = NULL
    --If the node has at most one child
    if (node.leftChild = NULL or node.rightChild = NULL) then
        toBeDeleted = node
    else
        --replace node.key with the minimum key from right subtree
        toBeDeleted = findSuccessor(node)
        swapKey(toBeDeleted, node)
    fi
    if (toBeDeleleted.leftChild != NULL) then
        newChild = toBeDeleted.leftChild
    else
        newChild = toBeDeleted.rightChild
    fi
    if (isRoot(toBeDeleted) then
        tree.root = newChild
    else if isLeftChild(toBeDeleted) then
        toBeDeleted.parent.leftChild = newChild
    else
        toBeDeleted.parent.rightChild = newChild
    fi
END
```

A4. AVL Rebalance Algorithm

```
Algorithm: AVLRebalance
Inputs: node: Pointer;

AVLRebalance(node)
BEGIN
    n = node
    while(n != NULL)
        n.height = max(n.leftChild.height, n.rightChild.height) + 1
        balance = n.balance
        if(abs(balance) > 1) then
            --If left sub-tree is deeper than the right sub-tree
            if(balance < 0) then
                if(n.leftChild.balance > 0)
                    leftRotation(n.leftChild)
                    rightRotation(n)
                else
                    rightRotation(n)
                fi
            --right sub-tree is deeper than the left sub-tree
            else
                if(n.rightChild.balance < 0) then
                    rightRotation(n.rightChild)
                    leftRotation(n)
                else
                    leftRotation(n)
                fi
            fi
        fi
        n = n.parent
    od
END
```

A5. Red-Black Insertion Rebalance Algorithm

```
 Algorithm: Red-BlackInsertionRebalance
 Inputs: tree: Red-Black tree; node: Pointer;

 Red-BlackInsertionRebalance(tree, node)
 BEGIN
     redSon = node
     n = node.parent
     while(isRed(n))
         sibling = n.sibling
         if(isRed(sibling)) then
             switchColor(sibling)
             switchColor(n)
             switchColor(n.parent)
             redSon = n.parent
             n = redSon.parent;
         else
             if(isLeftChild(n) and isLeftChild(redSon)) then
                 switchColor(n.parent)
                 switchColor(n)
                 rightRotation(n.parent)
             else if(isRightChild(n) and isRightChild(redSon)) then
                 switchColor(n.parent)
                 switchColor(n)
                 leftRotation(n.parent)
             else if(isLeftChild(n) and isRightChild(redSon)) then
                 leftRotation(n)
                 temp = redSon;
                 redSon = n;
                 n = temp;
            else
                rightRotation(n)
                temp = redSon;
                redSon = n;
                n = temp;
            fi
        fi
    od
    if(isRed(tree.root))
        switchColor(tree.root)
    fi
END
```

A6. Red-Black Deletion Rebalance Algorithm

```
Algorithm: Red-BlackDeletionRebalance
Inputs: tree: Red-Black tree; node: Pointer;

Red-BlackDeletionRebalance(tree, node)
BEGIN
    n = node
    while(!isRoot(n) and isBlack(n))
        parent = n.parent
        if(isLeftChild(n)) then
            sibling = n.sibling
            if(isRed(sibling)) then
                setBlack(sibling)
                setRed(parent)
                leftRotation(parent)
                sibling = n.sibling
            fi
            if(hasTwoBlackChildren(sibling)) then
                setRed(sibling)
                n = n.parent
            else
                if(isBlack(sibling.rightChild)) then
                    setBlack(sibling.leftChild)
                    setRed(sibling)
                    rightRotation(sibling)
                    sibling = n.sibling
                fi
                sibling.setColor(parent.color)
                setBlack(parent)
                setBlack(sibling.rightChild)
                leftRotation(parent)
                n = tree.root
            fi
        else
            sibling = n.sibling
            if(isRed(sibling)) then
                setBlack(sibling)
                setRed(parent)
                rightRotation(parent)
                sibling = n.sibling
            fi
            if(hasTwoBlackChildren(sibling)) then
                setRed(sibling)
                n = n.parent
            else
                if(isBlack(sibling.leftChild)) then
                    setBlack(sibling.rightChild)
                    setRed(sibling)
                    leftRotation(sibling)
                sibling = n.sibling
            fi
            if(hasTwoBlackChildren(sibling)) then
                setRed(sibling)
                n = n.parent
            else
                if(isBlack(sibling.leftChild)) then
                    setBlack(sibling.rightChild)
                    setRed(sibling)
                    leftRotation(sibling)
                    sibling = n.sibling
                fi
                sibling.setColor(parent.color)
                setBlack(parent)
                setBlack(sibling.leftChild)
                rightRotation(parent)
                n = tree.root
            fi
        fi
    od
    if(isRed(n)) then
        setBlack(n)
    fi
END
```

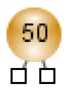
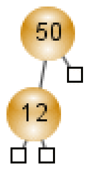
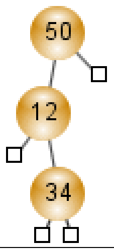
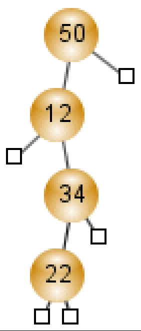
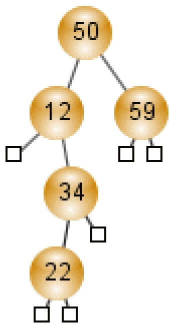
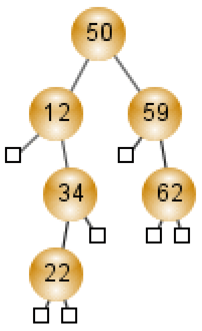
A7. Splay Algorithm

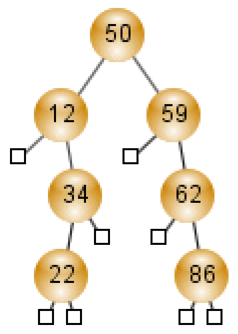
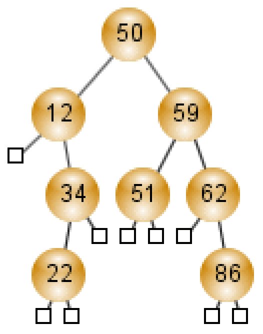
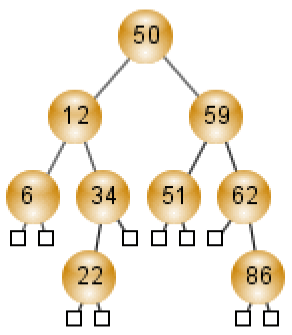
```
Algorithm: splay
Inputs: node: Pointer;

Splay(node)
BEGIN
    parent = node.parent
    grandparent = NULL
    while(parent != NULL)
        grandparent = parent.parent
        if(grandparent = NULL) then
            if(isLeftChild(node)) then
                rightRotation(parent)
            else
                leftRotation(parent)
            fi
        else
            if(isLeftChild(parent)) then
                if(isLeftChild(node)) then
                    rightRotation(grandparent)
                    rightRotation(parent)
                else
                    leftRotation(parent)
                    rightRotation(grandparent)
                fi
            else
                if(isRightChild(node)) then
                    leftRotation(grandparent)
                    leftRotation(parent)
                else
                    rightRotation(parent)
                    leftRotation(grandParent)
                fi
            fi
        fi
        parent = node.parent
    od
END
```

Appendix B: Complete Testing Document

B1. Testing the Insertion Algorithm

Test number	Input	Expected Output	Actual Output	Decision
BSTI1	50	50 is set as the root of the tree		Pass
BSTI2	12	12 is placed to the left of 50		Pass
BSTI3	34	34 is placed to the right of 12		Pass
BSTI4	22	22 is placed to the left of 34		Pass
BSTI5	59	59 is placed to the right of 50		Pass
BSTI6	62	62 is placed to the right of 59		Pass

BSTI7	86	86 is placed to the right of 62		Pass
BSTI8	51	51 is placed to the left of 59		Pass
BSTI9	6	6 is placed to the left of 12		Pass
BSTI10	22	22 cannot be inserted. Already exists	22 already exists in the tree	Pass
BSTI11	34	34 cannot be inserted. Already exists	34 already exists in the tree	Pass
BSTI12	59	59 cannot be inserted. Already exists	59 already exists in the tree	Pass

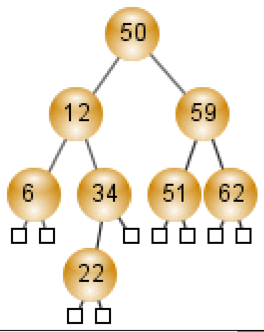
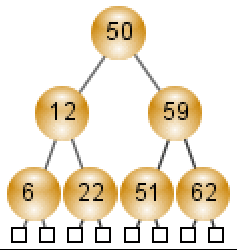
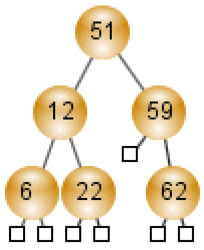
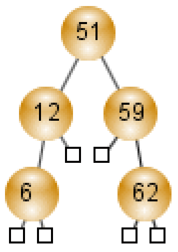
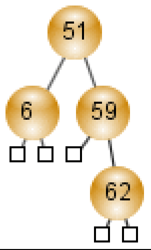
B2. Testing the Searching Algorithm

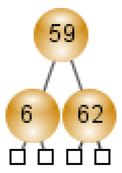
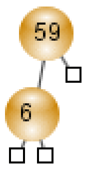
(based on the tree produced during the insertions)

Test number	Input	Expected Output	Actual Output	Decision
BSTS1	50	Found	Found	Pass
BSTS2	12	Found	Found	Pass
BSTS3	33	Not Found	Not Found	Pass
BSTS4	22	Found	Found	Pass
BSTS5	99	Not Found	Not Found	Pass
BSTS6	3	Not Found	Not Found	Pass

BSTS7	86	Found	Found	Pass

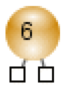
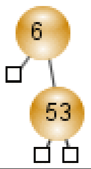
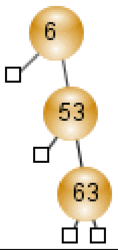
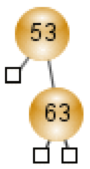
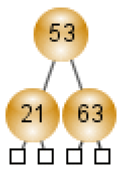
B3. Testing the Deletion Algorithm (based on the tree produced during the insertions)

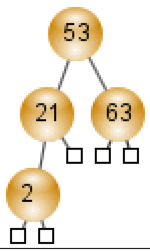
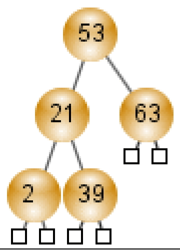
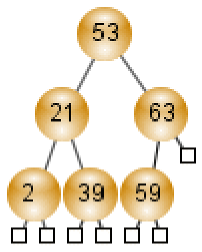
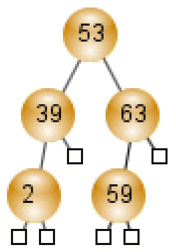
Test number	Input	Expected Output	Actual Output	Decision
BSTD1	86	Deleted		Pass
BSTD2	34	Replaced by its right child 22		Pass
BSTD3	50	Replaced by its successor 51		Pass
BSTD4	22	Deleted		Pass
BSTD5	12	Replaced by its left child 6		Pass
BSTD6	50	Not Found	Not Found	Pass
BSTD7	34	Not Found	Not Found	Pass

BSTD8	51	Replaced by its right child 59		Pass
BSTD9	98	Not Found	Not Found	Pass
BSTD9	62	Deleted		Pass

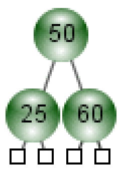
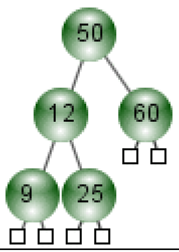
B4. Testing Searching, Insertion and Deletion Algorithms

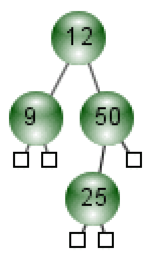
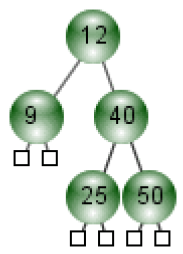
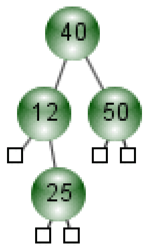
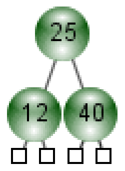
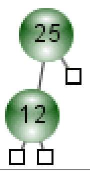
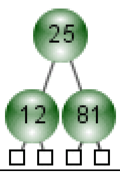
(Based on the tree resulted by the deletions)

Test number	Input	Expected Output	Actual Output	Decision
BSTR1	Searching 59	Found	Found	Pass
BSTR2	Deleting 59	Replaced by 6		Pass
BSTR3	Searching 59	Not Found	Not Found	Pass
BSTR4	Deleting 23	Not Found	Not Found	Pass
BSTR5	Insert 6	Already exists	Already exists	Pass
BSTR6	Insert 53	Inserted to the right of 6		Pass
BSTR7	Insert 63	Inserted to the right of 53		Pass
BSTR8	Delete 6	Replaced by its right child 53		Pass
BSTR9	Search 53	Found	Found	Pass
BSTR10	Insert 21	Inserted to the left of 53		Pass

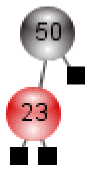
BSTR11	Insert 2	Inserted to the left of 21		Pass
BSTR12	Insert 39	Inserted to the right of 21		Pass
BSTR13	Insert 59	Inserted to the left of 63		Pass
BSTR14	Delete 21	Replaced by its right child 39		Pass
BSTR15	Insert 39	Already exists	Already exists	Pass

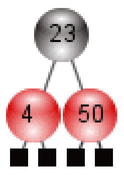
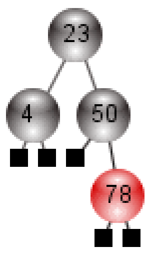
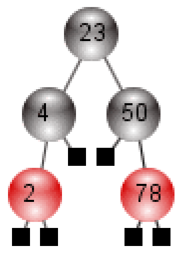
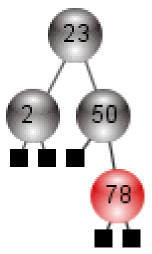
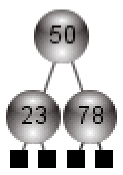
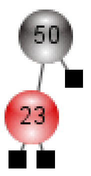
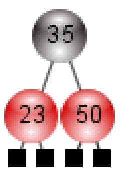
B5. Testing AVL Rebalance Algorithm

Test number	Input	Expected Output	Actual Output	Decision
AVL1	Insert 25, 50 and 60	Left Rotation of 50 over its parent 25		Pass
AVL2	Insert 12 and 9	Right rotation of 12 over its parent 25		Pass

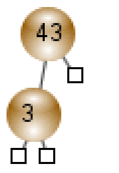
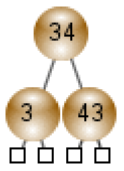
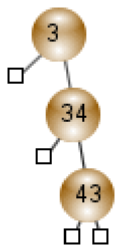
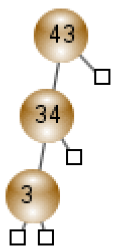
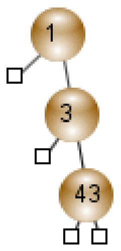
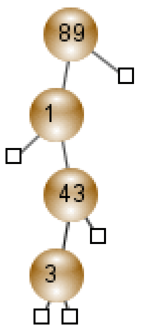
AVL3	Delete 60	Right Rotation of 12 over its parent 50		Pass
AVL4	Insert 40	Rotate 40 left and then right over its new parent 50		Pass
AVL5	Delete 9	Rotate 40 left over its parent 12		Pass
AVL6	Delete 50	Rotate 25 left and then right over its new parent 40		Pass
AVL7	Delete 40	No rotation needed. The tree is already balanced		Pass
AVL8	Insert 81	No rotation needed. The tree is already balanced		Pass

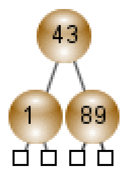
B6. Testing Red-Black Insertion and Deletion Rebalance Algorithms

Test number	Input	Expected Output	Actual Output	Decision
RB1	Insert 50 and 23	No Red-Black violation		Pass

RB2	Insert 4	Violation of red condition is restored by: a. Switching the colour of 50 to red b. Switching the colour of 23 to black c. Performing a left rotation of 23 over its parent 50		Pass
RB3	Insert 78	Violation of red condition is restored by: a. Switching the colour of nodes 4 and 50 to black		Pass
RB4	Insert 2	No violation		Pass
RB5	Delete 4	Violation of black condition is restored by a. Switching the colour of node 2 to black		Pass
RB6	Delete 2	Violation of black condition is restored by: a. Switching the colour of sibling's right child 78 to black b. Performing a left rotation of the sibling 50 over its parent 23		Pass
RB7	Delete 78	Violation of black condition is restored by: a. Switching the colour of the sibling 23 to red		Pass
RB8	Insert 35	Violation of red condition is restored by: a. Performing a left rotation of 35 over its parent 23 b. Switching the colour of 35's new parent 50 to red c. Switching the colour of 35 to black d. Performing a right rotation of 35 over its new parent 50		Pass

B7. Testing Splay Algorithm

Test #	Input	Expected Output	Actual Output	Decision
SPL1	Insert 3	No splaying	No splaying	Pass
SPL2	Insert 43	Node 43 is splayed to the root by: a. Performing a left rotation of 43 over its parent 3		Pass
SPL3	Insert 34	Node 34 is splayed to the root by: a. Performing a left rotation 34 over its parent 3 b. Performing a right rotation 34 over its new parent 43		Pass
SPL4	Search 3	Node 3 is splayed to the root by: Performing a right rotation over its parent 34		Pass
SPL5	Search 67	Node 43 is splayed to the root by: a. Performing a double left rotation of 34 over its parent 3 and the 43 over its parent 34		Pass
SPL6	Insert 34	No splaying, 34 already exists	No splaying	Pass
SPL7	Delete 34	No splaying, 3's parent is already the root	No splaying	Pass
SPL8	Insert 1	Node 1 is splayed to the root		Pass
SPL9	Insert 89	Node 89 is splayed to the root		Pass

SPL10	Delete 3	Node 43 is splayed to the root		Pass

B8. Testing the tree operation buttons

Test #	Input	Expected Output	Actual Output	Decision
TOB1	Insert button pressed	A new window is displayed in the middle of the screen asking for input either by text or randomly	A new window asking for input by text or randomly	Pass
TOB1.1	No input is entered in the text field of the new window	A warning window saying that the input must be an integer between 1 and 99	A warning window saying that the input must be an integer between 1 and 99	Pass
TOB1.2	An invalid input is entered	A warning window saying that the input must be an integer between 1 and 99	A warning window saying that the input must be an integer between 1 and 99	Pass
TOB1.3	The random button is pressed	A randomly generated number is displayed in the text field	A randomly generated number is displayed in the text field	Pass
TOB1.4	A valid input is entered either using the text field or the random button	Returns to the main display for the animation of the insertion operation	Returns to the main display for the animation of the insertion operation	Pass
TOB1.5	The cancel button is pressed	The operation is cancelled and the window is disappeared	The operation is cancelled and the window is disappeared	
TOB2	Delete button pressed	A new window is displayed in the middle of the screen asking for input either by text or randomly	A new window asking for input by text or randomly	Pass
TOB2.1	As in TOB1.1 – TOB1.5	As in TOB1.1 –TOB1.5 with the difference that inputs are addressed for the deletion operation	As in TOB1.1 –TOB1.5 where inputs are addressed for the deletion operation	Pass
TOB3	Search button pressed	A new window is displayed in the middle of the screen asking for input either by text or randomly	A new window asking for input by text or randomly	Pass
TOB3.1	As in TOB1.1 –	As in TOB1.1 –TOB1.5	As in TOB1.1 –TOB1.5	Pass

	TOB1.5	with the difference that inputs are addressed for the searching operation	where inputs are addressed for the searching operation	
TOB4	Remove All button pressed	A new window is displayed on the middle of the screen asking to validate this operation	A new window asking for operation validation	Pass
TOB4.2	The Yes button is pressed in the new window	Animation of all the nodes of the tree being removed	Animation of all the nodes of the tree being removed	Pass
TOB4.1	The No button is pressed in the new window	The window is disappeared and the nodes are not removed	The window is disappeared and the nodes are not removed	Pass
TOB5	Traverse button pressed	A new window is displayed on the middle of the screen providing a selection of traversal algorithm	A new window asking to select a traversal algorithm	Pass
TOB5.1	The yes button is pressed	Returns to the main display, ready to animate the selected traversal	Returns to the main display, ready to animate the selected traversal	Pass
TOB5.1	The No button is pressed	Cancels the operation and the window disappears	The operation is cancelled and the window disappears	Pass
TOB6	The animation speed value of the scroll bar is increased/ decreased	The animation speed is increased/ decreased	The animation speed is increased/ decreased	Pass
TOB7	The frame rate value of the scroll bar is increased/ decreased	The animation runs using more/ less frames per second	The animation runs using more/ less frames per second	Pass

B9. Testing the tree selection buttons

Test #	Input	Expected Output	Actual Output	Decision
TSB1	The Binary Search Tree button is pressed	The current Binary Search Tree is displayed in the tree visualization display and the button is disabled. All the other tree buttons are enabled	The current Binary Search Tree is displayed in the tree visualization display and the button is disabled. All the other tree buttons are enabled	Pass
TSB2	The AVL tree button is pressed	The current AVL tree is displayed in the tree visualization display and	The current AVL tree is displayed in the tree visualization display and	Pass

		the button is disabled. All the other tree buttons are enabled	the button is disabled. All the other tree buttons are enabled	
TSB3	The Red-Black tree button is pressed	The current Red-Black tree is displayed in the tree visualization display and the button is disabled. All the other tree buttons are enabled	The current Red-Black tree is displayed in the tree visualization display and the button is disabled. All the other tree buttons are enabled	Pass
TSB4	The Splay tree button is pressed	The current Splay tree is displayed in the tree visualization display and the button is disabled. All the other tree buttons are enabled	The current Splay tree is displayed in the tree visualization display and the button is disabled. All the other tree buttons are enabled	Pass

B10. Testing the animation control buttons

Test #	Input	Expected Output	Actual Output	Decision
ACB1	The play button is pressed	The animation starts and all the other animation buttons apart the pause button are disabled	The animation starts and all the other animation buttons apart the pause button are disabled	Pass
ACB2	The animation finishes	All buttons are disabled apart the rewind and step backwards buttons	All buttons are disabled apart the rewind and step backwards buttons	Pass
ACB3	The pause button is pressed during the animation	The animation pauses and all the other buttons apart the pause button are enabled	The animation pauses and all the other buttons apart the pause button are enabled	Pass
ACB4	The play button is pressed after the animation is paused	The animation resumes from the current state	The animation resumes from the current state enabled	Pass
ACB5	The step forwards button is pressed when the animation is paused	The animation moves a step forwards. If reach the end of the animation, the step forwards and the fast forwards buttons are disabled	The animation moves a step forwards. The step forwards and the fast forwards buttons are disabled when the next state is the last one	Pass
ACB6	The step backwards button is pressed when the animation is paused	The animation moves a step backwards. If reach the beginning of the animation, the step backwards and rewind buttons are disabled	The animation moves a step forwards. The step backwards and rewind buttons are disabled when the previous state is the initial one	Pass
ACB7	The rewind button is	The animation restarts and moves to the initial	The animation restarts and moves to the initial	Pass

	pressed when the animation is paused	state. Step backwards and rewind buttons are disabled	state. Step backwards and rewind buttons are disabled	
ACB6	The fast forwards button is pressed when the animation is paused	The animation restarts and moves to the final state. The step forwards and fast forwards buttons are disabled.	The animation moves a step forwards. The step forwards and fast forwards buttons are disabled.	Pass

B11. Testing the pseudo code and explanations

Test #	Description	Expected Output	Actual Output	Decision
PCD1	The animation moves to the next state	The line of code for the next state is highlighted and explanation is displayed for that line.	The line of code for the next state is highlighted and explanation is displayed for that line.	Pass
PCD2	The animation moves to the previous state	The line of code for the previous state is highlighted and explanation is displayed for that line.	The line of code for the previous state is highlighted and explanation is displayed for that line.	Pass
PCD3	The algorithm for the current animation changes	The new pseudo code is loaded and displayed on the screen and the explanation is cleared	The new pseudo code is loaded and displayed on the screen and the explanation is cleared	Pass
PCD4	The pseudo code for the current algorithm is larger than the pseudo code display	A scroll bar is displayed	A scroll bar is displayed	Pass
PCD4.1	The line of the pseudo code to be highlighted is located off-screen	The display is scrolled to show the highlighted line of code	The display is scrolled to show the highlighted line of code	Pass

B12. Testing the tree visualization display

Test #	Description	Expected Output	Actual Output	Decision
TVD1	A tree is selected	The tree is displayed without overlapped nodes.	The tree is displayed without overlapped nodes.	Pass
TVD2	The tree is modified as a result of an insertion	The tree is displayed without overlapped	The tree is displayed without nodes to be	Pass

	or deletion	nodes.	overlapped	
TVD3	The size of the tree exits the tree visualization display	A scroll bar to be displayed	A scroll bar is displayed	Pass
TVD4	The area of interest for the current animation exits the size of the tree visualization display	The display is automatically scrolled to the area of interest	The display needs to be scrolled manually	Fail

B13. Testing the Menu Bar

Test #	Description	Expected Output	Actual Output	Decision
MB1	File → Exit	The system exits	The system is exited	Pass
MB2	Edit → Disable Explanation(checked)	Pseudo code and explanations are disappeared from the animation	The pseudo code and explanations are cleared	Pass
MB2.1	Check that animation control buttons work well when explanations are disabled	The buttons functionality is as in ACB1 – ACB6	The step forwards and step backwards are not disabled when reach the final and initial state respectively. They need to be pressed one more time.	Fail
MB3	Edit → Disable Explanation(unchecked)	Pseudo code and explanations are displayed for the current state	The pseudo code and explanations are displayed addressing to the current state	Pass
MB4	Edit → Case Performance → Best	The current tree is loaded and redisplayed, showing its best case performance	The best case performance is displayed for the current tree	Pass
MB5	Edit → Case Performance → Worst	The current tree is loaded and redisplayed, showing its worst case performance	The worst case performance is displayed for the current tree	Pass
MB6	Edit → Case Performance → Average	The current tree is loaded and redisplayed, showing its average case performance	The average case performance is displayed for the current tree	Pass
MB7	Edit → Change Node Colour → Colour	The colour of the nodes of the current tree is changed to the selected colour. This	The nodes are coloured the selected colour. This option is disabled for the Red-	Pass

		option is disabled for Red-Black trees	Black trees	
MB8	Edit → Tree Entrance→ Angle	When a tree is selected it enters the display from the selected angle	The tree enters the display from the selected angle	Pass
MB9	Help → Tree Explanation	A new window is displayed, showing a description of the current tree	Description for the current tree is displayed	Pass

Appendix C: User Manual

1. Tool Introduction

Binary Search Trees applet is a tool developed to assist the understanding of Binary Search Trees and their associated algorithms for basic operations as searching, insertion, deletion and traversals. The tool uses animation effects to visualize the algorithms associated with Binary Search Trees while they operate. It also provides full control over the animation and in particular, links the animation to the pseudo code in order to help the user understand the algorithm. A range of different types of Binary Search Trees as Red-Black, AVL and Splay trees were selected and included in the tool.

Note: The tool is programmed in Java programming language, and requires a Java Virtual Machine (JVM) installed on your machine.

2. Getting started

Once the applet is loaded, a screen will appear as in figure 1 below.

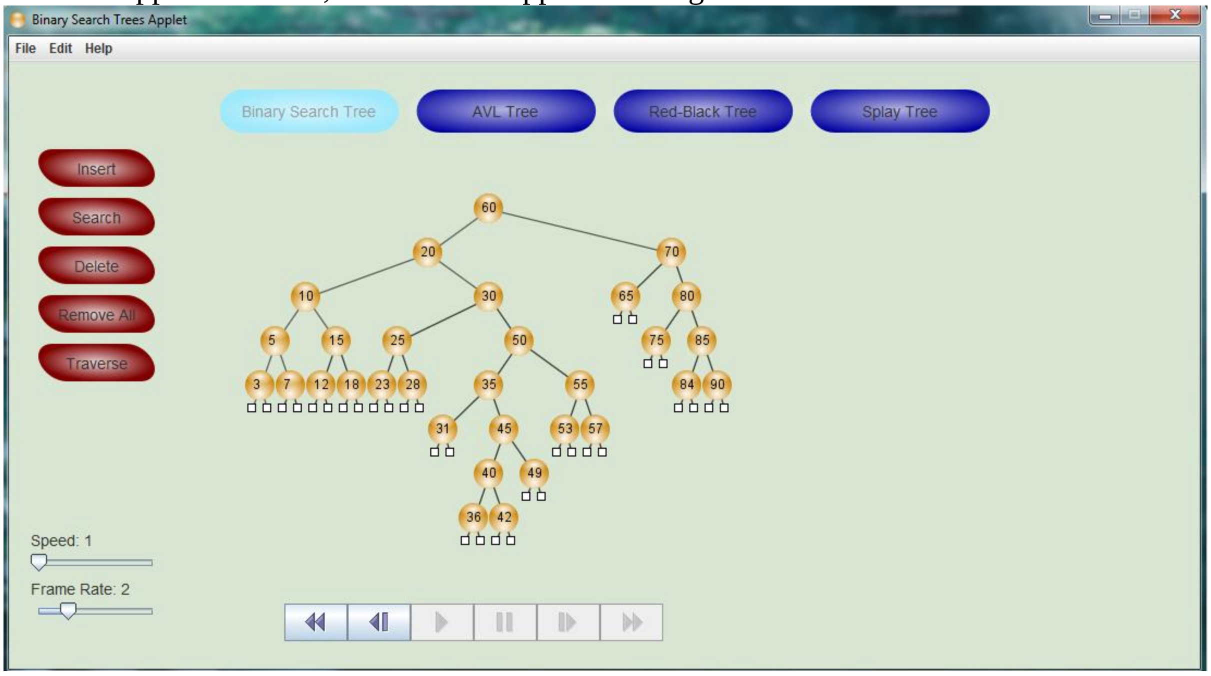


Figure 1: The applet when it loads

Tree selection buttons: The top blue buttons that allow you to select a Binary Search Tree. A light blue button indicates the selected tree.
Operation Buttons: The left most buttons that allow you to select an operation to be performed.
Animation Control buttons the buttons on the bottom of the tool that are used for controlling the animation.
Sliders: The speed and frame rate sliders that allow you to set the animation speed and the frames per second respectively.

Menu: The menu provides functions for loading case performance data, changing nodes colour, selecting an entrance for the tree and explanation about the currently running Binary Search Tree.

2.1 Operations Buttons

Insert Button: Inserts a node to the tree
When the insert button is pressed, a window appears on the screen asking for an input from the user as in figure 2.

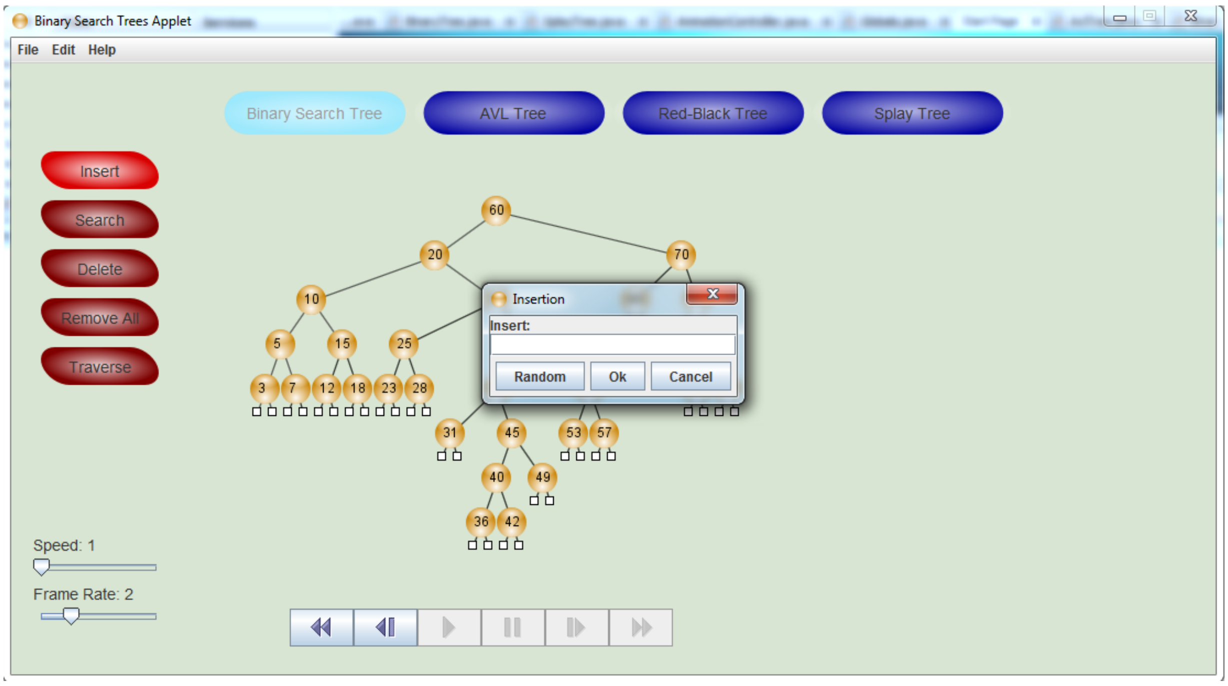


Figure 2: The insert button is pressed

Insert window choices:
Random button: Generates a random number to be inserted.
Text Field: Allows you to specify your own number to be inserted.
Cancel button: Cancels the operation.
Delete and Search Button: A similar window appears on the screen when pressing the Delete and Search buttons asking for a number to be deleted and searched respectively.
RemoveAll button: Removes all the nodes of the tree. When all nodes are removed, you can either start creating your own tree using the operations available or load a case performance such that it will load a new pre-generated tree according to the case selected.
Traverse Button: Traverses the tree using 3 different traversing algorithms as in-order, pre-order and post-order.

The window that appears when traverse button is pressed is shown in figure 3

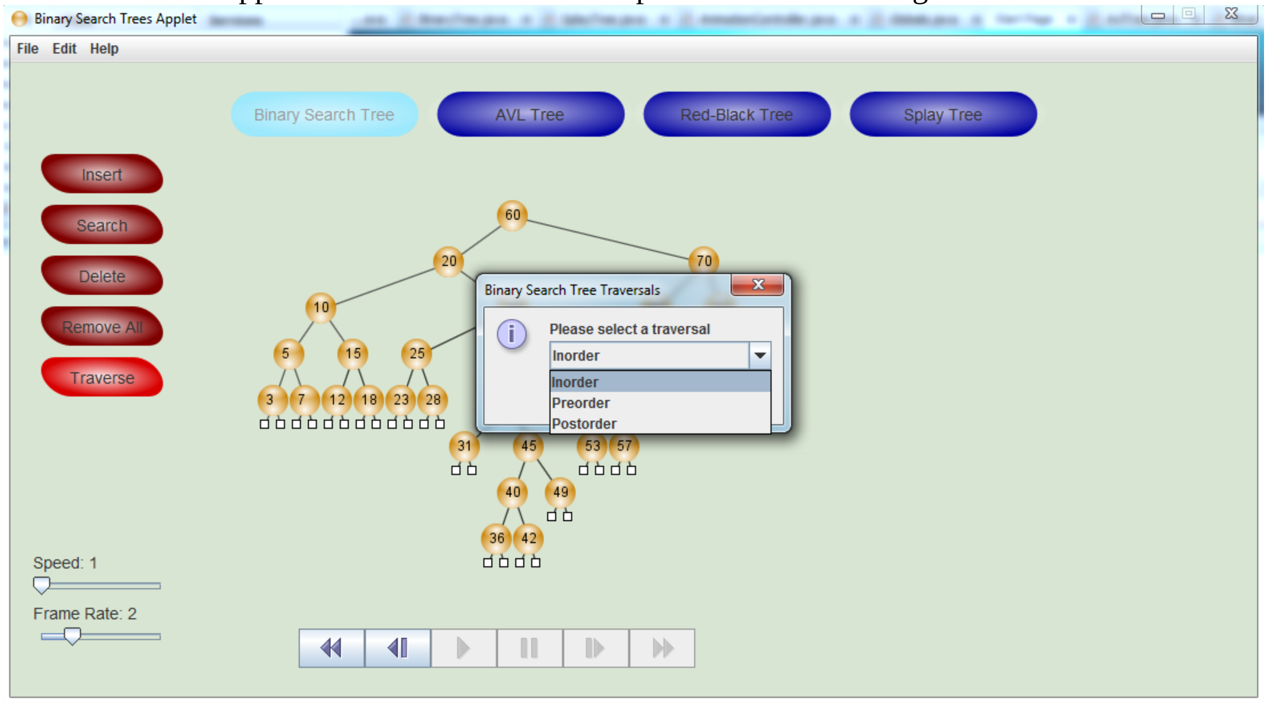


Figure 3: The traverse button is pressed

2.2 Animation Control buttons

Rewind Button: Rewinds the animation to the initial position since an operation button was pressed.
Note: When a user selects another binary tree (AVL, Red-Black or Splay tree), rewind button is enabled in order to allow him to watch the last operation performed on that newly selected tree.

Step backwards button: Steps the algorithm backwards

Play button: Starts the algorithm animation

Pause button: Pauses the algorithm animation

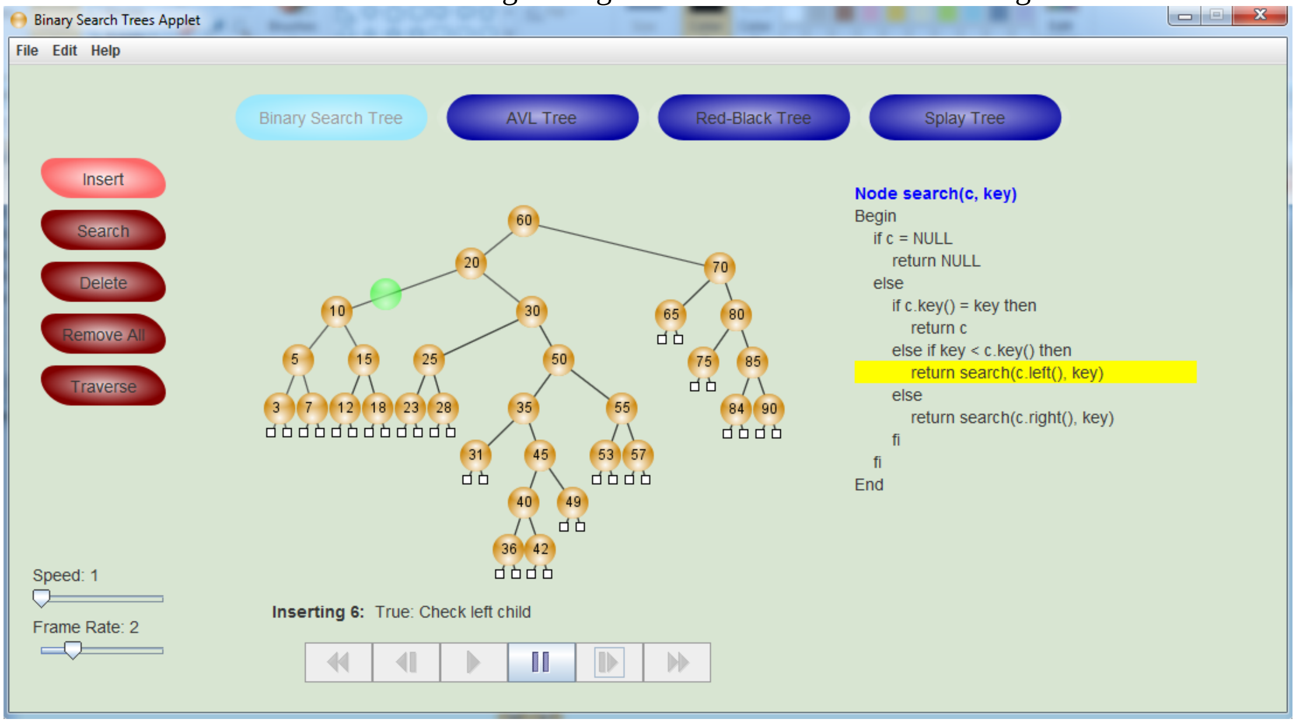
Step forwards: Steps the algorithm forwards.

Fast-forwards: Moves the animation to its final state. For example when inserting a node and then this button is pressed, it will automatically place the node to its final position (if it does not already exist).
Note: If a user decides to insert or delete a node while there is pending insertion or deletion operation then the pending operation is automatically performed and moves the associated node to its final position before the new operation takes place. For example, a user chooses to insert a new node, let us say 12 but before he/she watches the animation of 12 being inserted to the tree, it chooses to insert or delete a new node, let us say 45. In such case 12 is automatically placed to its final position before the animation of the new node 45 starts playing.

2.3 Pseudo-Code and explanation Comments

Pseudo-code: When the animation starts, a pseudo-code appears on the right hand side of the tool highlighting the code currently executed.
Pseudo-code explanation comments: In addition to the pseudo-code, there are some comments just above the animation control buttons, describing the currently executed piece of code.

A screenshot of the tool executed during the algorithm animation is shown in figure 4 below.



```
Node search(c, key)
Begin
    if c = NULL
        return NULL
    else
        if c.key() = key then
            return c
        else if key < c.key() then
            return search(c.left(), key)
        else
            return search(c.right(), key)
        fi
    fi
End
```

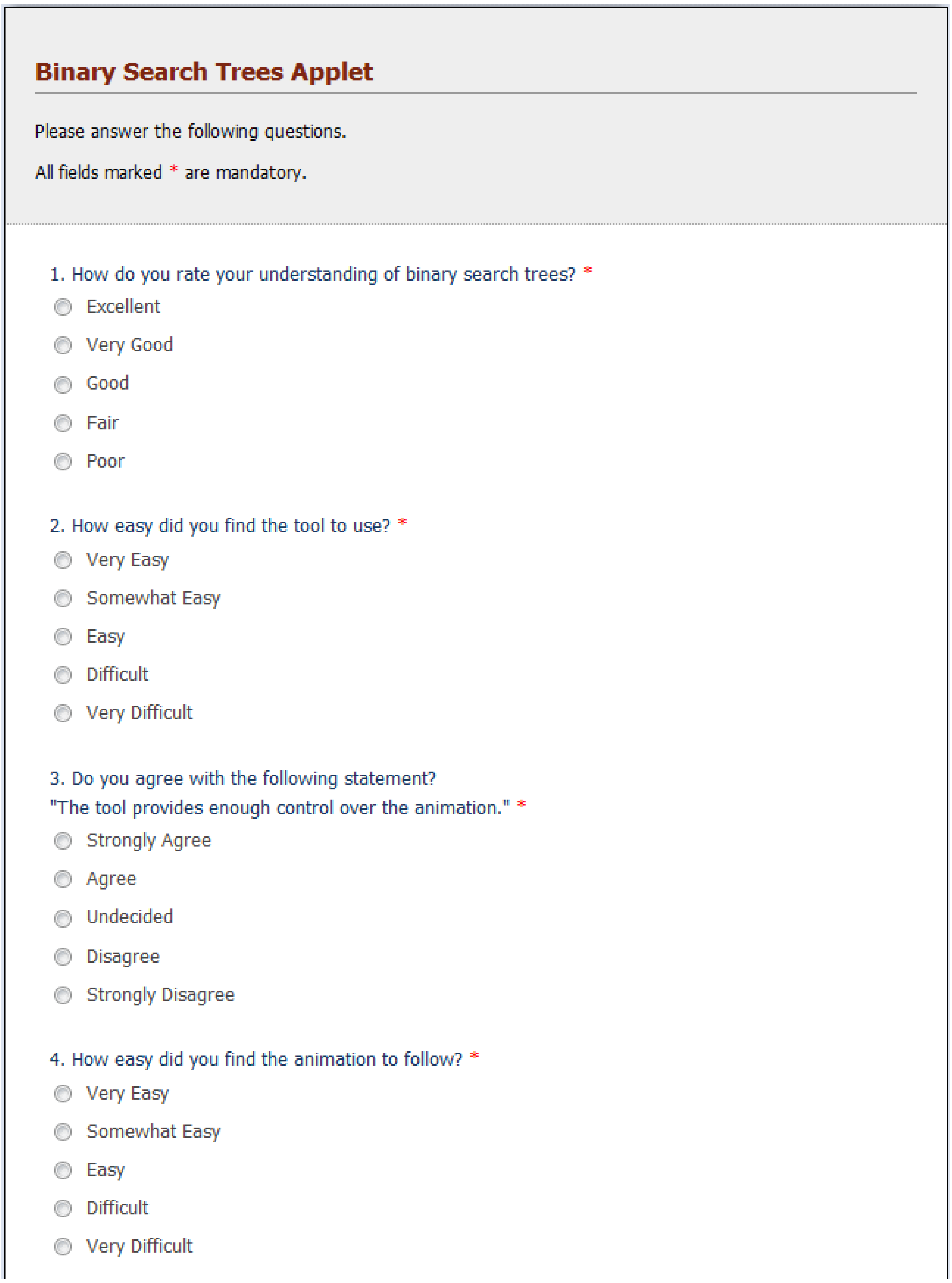
Inserting 6: True: Check left child

Figure 4: The tool while executing an insertion

For any questions, please contact me at victor.georgiou@ncl.ac.uk

Appendix D: Questionnaire Results

D1. The Questionnaire



Binary Search Trees Applet

Please answer the following questions.

All fields marked * are mandatory.

1. How do you rate your understanding of binary search trees? *

- ○ Excellent
- ○ Very Good
- ○ Good
- ○ Fair
- ○ Poor

2. How easy did you find the tool to use? *

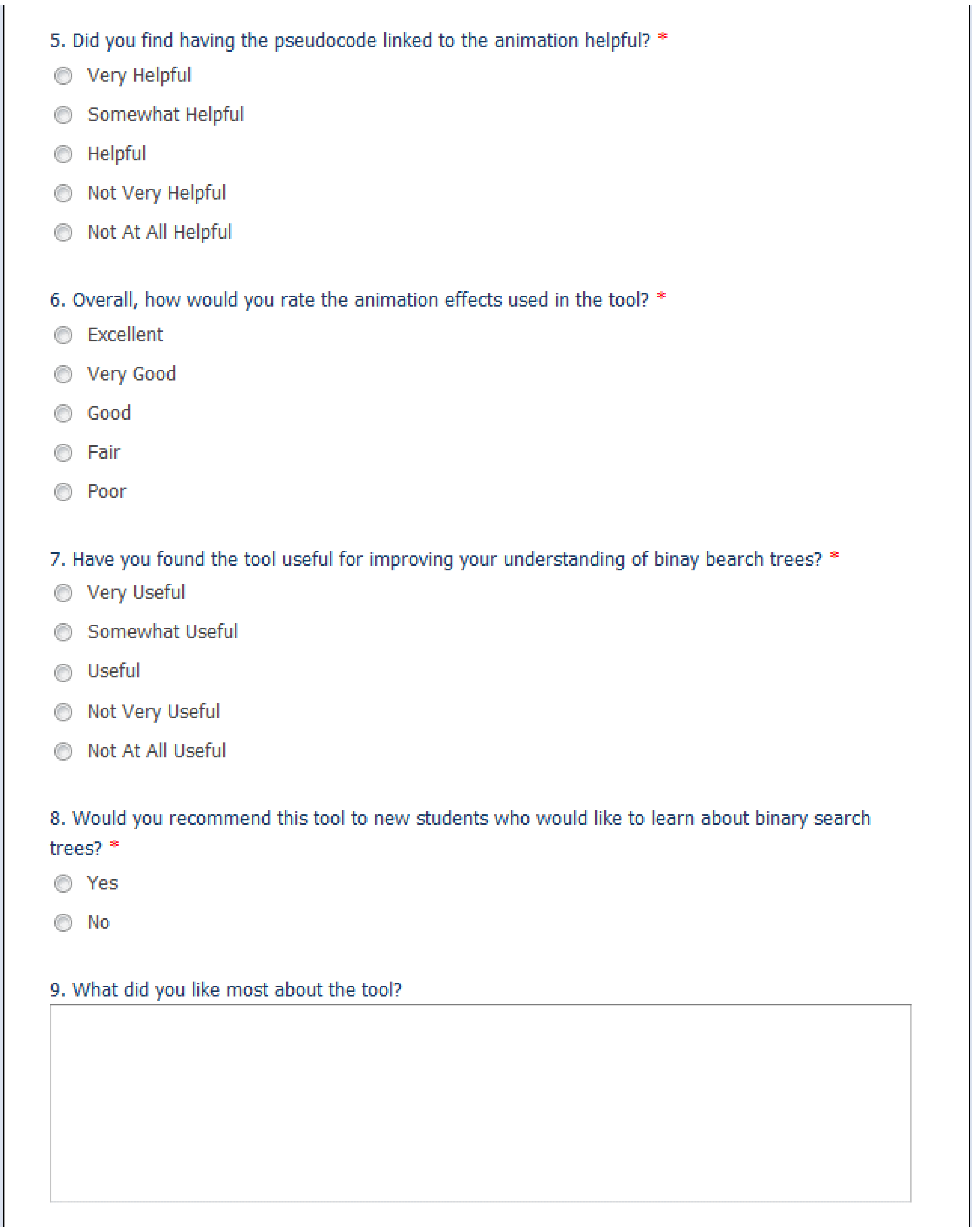
- ○ Very Easy
- ○ Somewhat Easy
- ○ Easy
- ○ Difficult
- ○ Very Difficult

3. Do you agree with the following statement?
"The tool provides enough control over the animation." *

- ○ Strongly Agree
- ○ Agree
- ○ Undecided
- ○ Disagree
- ○ Strongly Disagree

4. How easy did you find the animation to follow? *

- ○ Very Easy
- ○ Somewhat Easy
- ○ Easy
- ○ Difficult
- ○ Very Difficult



5. Did you find having the pseudocode linked to the animation helpful? *

- ○ Very Helpful
- ○ Somewhat Helpful
- ○ Helpful
- ○ Not Very Helpful
- ○ Not At All Helpful

6. Overall, how would you rate the animation effects used in the tool? *

- ○ Excellent
- ○ Very Good
- ○ Good
- ○ Fair
- ○ Poor

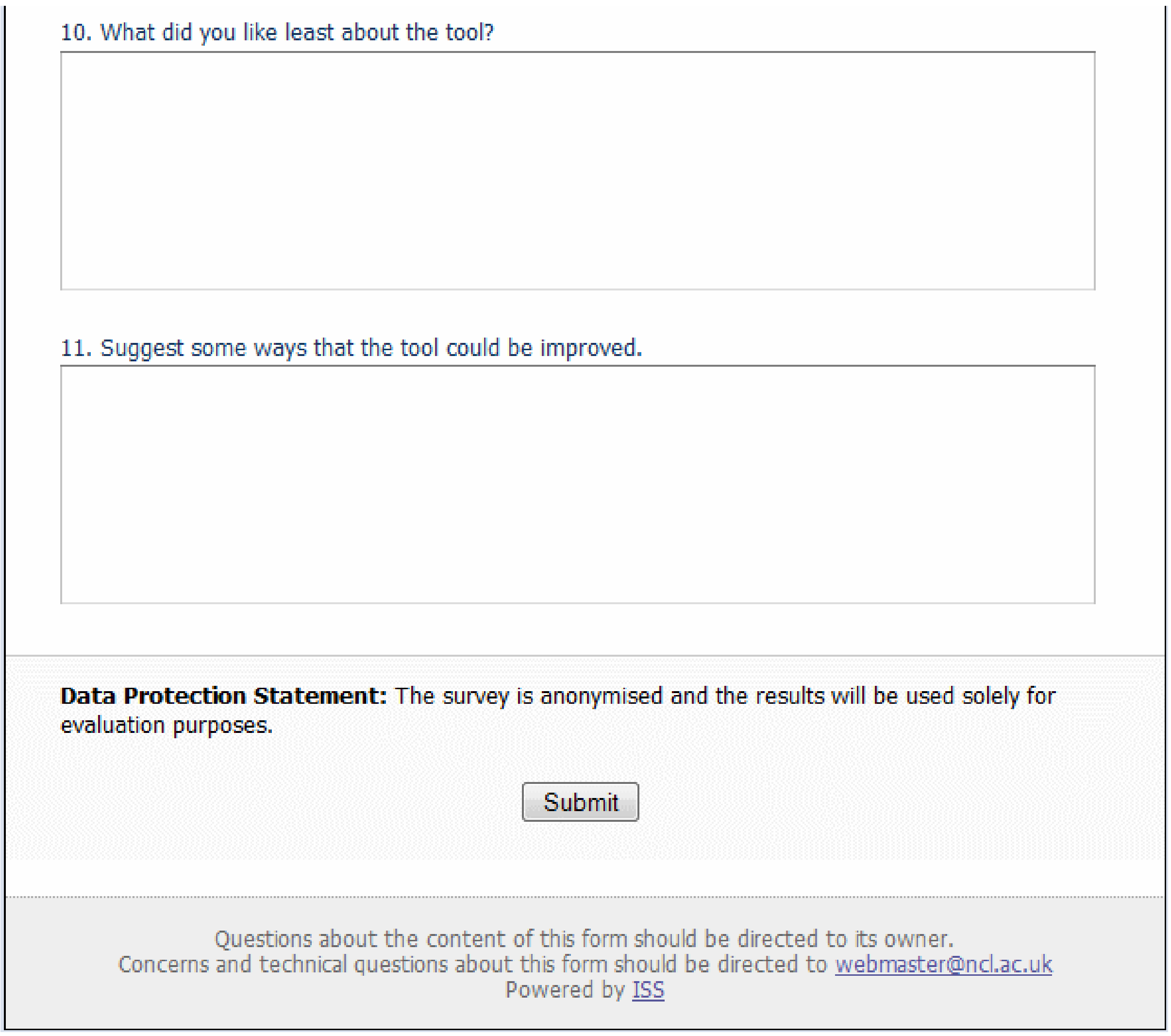
7. Have you found the tool useful for improving your understanding of binay bearch trees? *

- ○ Very Useful
- ○ Somewhat Useful
- ○ Useful
- ○ Not Very Useful
- ○ Not At All Useful

8. Would you recommend this tool to new students who would like to learn about binary search trees? *

- ○ Yes
- ○ No

9. What did you like most about the tool?


10. What did you like least about the tool?

11. Suggest some ways that the tool could be improved.

Data Protection Statement: The survey is anonymised and the results will be used solely for evaluation purposes.

Submit

Questions about the content of this form should be directed to its owner.
Concerns and technical questions about this form should be directed to webmaster@ncl.ac.uk
Powered by ISS

D2. Survey Results

	Answers for Question Numbers			
Questionnaire Number	1	2	3	4
1	Excellent	Very Easy	Agree	Somewhat Easy
2	Very Good	Somewhat Easy	Agree	Easy
3	Good	Very Easy	Strongly Agree	Very Easy
4	Very Good	Very Easy	Agree	Easy
5	Good	Very Easy	Strongly Agree	Somewhat Easy
6	Excellent	Somewhat Easy	Agree	Very Easy
7	Good	Somewhat Easy	Agree	Somewhat Easy
8	Good	Very Easy	Agree	Very Easy
9	Good	Very Easy	Strongly Agree	Very Easy
10	Very Good	Easy	Agree	Very Easy
11	Excellent	Very Easy	Strongly Agree	Very Easy
12	Very Good	Very Easy	Strongly Agree	Very Easy
13	Good	Very Easy	Agree	Very Easy

14	Good	Easy	Agree	Very Easy
15	Good	Easy	Agree	Easy
16	Very Good	Very Easy	Agree	Somewhat Easy
17	Good	Somewhat Easy	Strongly Agree	Very Easy
18	Good	Very Easy	Agree	Easy
19	Very Good	Very Easy	Strongly Agree	Somewhat Easy
20	Excellent	Easy	Agree	Very Easy
21	Very Good	Somewhat Easy	Agree	Easy

	Answers for Question Numbers			
Questionnaire Number	5	6	7	8
1	Very Helpful	Very Good	Very Useful	Yes
2	Very Helpful	Very Good	Useful	Yes
3	Very Helpful	Excellent	Very Useful	Yes
4	Helpful	Very Good	Very Useful	Yes
5	Helpful	Very Good	Somewhat Useful	Yes
6	Very Helpful	Very Good	Somewhat Useful	Yes
7	Helpful	Very Good	Somewhat Useful	Yes
8	Very Helpful	Very Good	Somewhat Useful	Yes
9	Very Helpful	Excellent	Very Useful	Yes
10	Somewhat Helpful	Excellent	Very Useful	Yes
11	Very Helpful	Excellent	Very Useful	Yes
12	Very Helpful	Excellent	Very Useful	Yes
13	Very Helpful	Good	Very Useful	Yes
14	Very Helpful	Good	Very Useful	Yes
15	Very Helpful	Good	Useful	Yes
16	Very Helpful	Excellent	Very Useful	Yes
17	Somewhat Helpful	Very Good	Somewhat Useful	Yes
18	Somewhat Helpful	Very Good	Somewhat Useful	Yes
19	Somewhat Helpful	Excellent	Somewhat Useful	Yes
20	Very Helpful	Excellent	Useful	Yes
21	Helpful	Good	Somewhat Useful	Yes

	Answers for Question Numbers
Questionnaire Number	9
1	The pseudo code showing the steps involved in each operation
7	The way, in which the applet demonstrates each individual in the algorithm, allowing students to see exactly what happens
8	Effects are nice, animates through the algorithm making it easy to understand
9	How it showed each line of code being executed
10	The animation and the pseudo code

11	What I liked most about the tool is the animation. It was easier to understand binary trees this way especially when removing and adding new elements.
21	The linking with pseudo code: This type of visual aid really helps in the understanding of the relevant lines of code.

	Answers for Question Numbers
Questionnaire Number	10
7	Controls slightly confusing at first: Unsure how to get the applet to run the demonstration
21	The scale and fit of the window: Once over a certain amount of nodes are added it becomes difficult to see and navigate.

	Answers for Question Numbers
Questionnaire Number	11
7	When a value is entered for insertion/deletion the program executes automatically
9	Add in more algorithms
21	Give the user the choice to skip directly to a specified line of code instead of having to either iterate through one line at a time or skip the whole lot

Appendix E: Class Implementations

E1. The `BinaryTree` abstract class

```
package binarySearchTrees;
import animationEngine.TreePanel;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.util.ArrayList;
import java.util.List;
import animationGenerator.AnimationGenerator;
/**
 * An abstract class representing a Binary tree
 * It contains methods for searhcing, insertion, deletion
 * and traversals.
 *
 * @author Victor
 */
public abstract class BinaryTree
{
    protected BinaryTreeNode root;
    public TreePanel treePanel;
    protected AnimationGenerator animationGenerator;
    protected BinaryTreeNode rootParent;
    private final String SEARCHING_PSEUDOCODE = "/data/search.txt";
    private final String INSERTION_PSEUDOCODE = "/data/insert.txt";
    private final String DELETION_PSEUDOCODE = "/data/deletion.txt";
    private List<Comparable> arrayTree;
    private boolean isUpdated;
    private BinaryTreeNode searchNode;
    protected int lastPseudoFile;
    protected boolean searching;

    /**
     * Constructor of the BinaryTree class which
     * constructs an empty Binary Search Tree
     */
    public BinaryTree()
    {
        rootParent = createNullNode();
        rootParent.setPosition(new Position(0, 50));
        root = createNullNode();
        root.setParent(rootParent);
        arrayTree = new ArrayList<Comparable>();
        searchNode = createNode(null);
    }
    /**
     * Returns true if the tree is empty
     */
    public boolean isEmpty()
    {
        return root.isNull();
    }
   /**
    * Returns the root of the tree
    */
    public BinaryTreeNode getRoot()
    {
        return root;
    }
   /**
    * Sets the node given as parameter to the root of the tree
    */
    public void setRoot(BinaryTreeNode node)
    {
        if(node != null)//.isNull())
        {
            root = (BinaryTreeNode)node;
            root.setParent(rootParent);
        }
    }
    /**
     * Searches for the newNode in the the current's subtree
     * @param current The root of the current subtree
     * @param newNode The node to search
     * @return The last visited node
     */
    private BinaryTreeNode find(BinaryTreeNode current, BinaryTreeNode newNode)
    {
        if(current.isRoot())
        {
            animationGenerator.addSearchNodeState(searchNode,root, true);
        }
        else if(!current.isNull())
        {
            animationGenerator.addSearchingState(newNode, current.parent(), current);
        }
        animationGenerator.addHighlightAndExplanationState(2, "Check if reach the end of the
tree");
        if(current.isNull())
        {
            animationGenerator.addHighlightAndExplanationState(3,"True: Key not found!");
            return current.parent();
        }
        else
        {
            animationGenerator.addHighlightAndExplanationState(4, "False: Comparing " +
newNode  + " with " + current);
            int d = newNode.key.compareTo(current.key);
            animationGenerator.addHighlightAndExplanationState(5, "Check if " + newNode  + " =
" + current);
            if (d == 0)
            {
                       animationGenerator.addHighlightAndExplanationState(6, "True: Key
       found!!");
               return current;
            }
            else if(d < 0)
            {
                animationGenerator.addHighlightAndExplanationState(7, "False: Check if " +
newNode  + " < " + current);
                animationGenerator.addHighlightAndExplanationState(8, "True: Check left
child");
                return find(current.leftChild(), newNode);
            }
            else
            {
                animationGenerator.addHighlightAndExplanationState(7, "False: Check if " +
newNode  + " < " + current);
                animationGenerator.addHighlightAndExplanationState(10, "False: Check right
child");
                return find(current.rightChild(), newNode);
            }
        }
    }
    /**
     * Search the tree for the node having the key given as parameter
     * @param key The key of the node to be searched for
     * @return
     */
    public  BinaryTreeNode find(Comparable key)
    {

        searchNode = createNode(key);
        animationGenerator.getAnimationStates().clear();
        animationGenerator.addChangePseudoCodeState(null, getPseudoCode(1));
        lastPseudoFile = 1;
        animationGenerator.addHighlightAndExplanationState(1, "Begin searching");
        BinaryTreeNode n =  find(root, searchNode);
        if(searching)
        {
             animationGenerator.addChangePseudoCodeState(getPseudoCode(lastPseudoFile), null);
        }
        return n;
    }
    /**
     * Creates a node with the key given as parameters and
     * inserts it in the tree (If it does not already exists)
     * @param key The key to be search for
     * @return
     */
    public BinaryTreeNode insert(Comparable key)
    {
         animationGenerator.getAnimationStates().clear();
         searching = false;
         int index=0;
         BinaryTreeNode parent = null;
         BinaryTreeNode newNode = createNode(key);
         if(isEmpty())
         {
            setRoot(newNode);
            index = 0;
         }
         else
         {
            parent = find(newNode.key);
            int d =  newNode.compareTo(parent);
            if(d == 0)
            {
                animationGenerator.addHighlightAndExplanationState(3, "Cannot insert duplicate
data");
                animationGenerator.addSearchNodeState(createNode(null),parent, false);
                searching = true;
                return null;
            }
            else
            {
                animationGenerator.addChangePseudoCodeState(getPseudoCode(1),
getPseudoCode(2));
                lastPseudoFile = 2;
                animationGenerator.addHighlightAndExplanationState(1, "Begin insertion");
                animationGenerator.addHighlightAndExplanationState(2, "Check if " + newNode +
" < " + parent);
                if(d < 0)
                {
                    animationGenerator.addHighlightAndExplanationState(3, "True: Set " +
newNode+ " as left child");
                    parent.setLeftChild(newNode);
                    index = 1;
                }
                else if(d > 0)
                {
                    animationGenerator.addHighlightAndExplanationState(5, "False: Set " +
newNode + " as right child");
                    parent.setRightChild(newNode);
                    index = 2;
                }
            }
        }
         searching = true;
         animationGenerator.addInsertionState(parent, newNode, index);
        return newNode;
    }
    /**
     * Deletes the node with the given key if it exists in the tree
     * @param key The key of the node to be deleted
     * @return
     */
    public BinaryTreeNode delete(Comparable key)
    {
        searching = false;
        BinaryTreeNode toBeDeleted = find(key);
        if(isEmpty() || toBeDeleted.key.compareTo(key) != 0)
        {
            animationGenerator.addSearchNodeState(createNode(null),toBeDeleted, false);
            searching = true;
            return toBeDeleted;
        }
        animationGenerator.addChangePseudoCodeState(getPseudoCode(1), getPseudoCode(3));
        lastPseudoFile = 3;
        animationGenerator.addHighlightAndExplanationState(1, "Begin deletion");
        BinaryTreeNode current = null;
        animationGenerator.addHighlightAndExplanationState(4, "Check if node " + toBeDeleted +
" has at most one child" );
        if(toBeDeleted.leftChild().isNull() || toBeDeleted.rightChild().isNull())
        {
            animationGenerator.addHighlightAndExplanationState(5, "True:  Node to be removed: 
" + toBeDeleted );
            current = toBeDeleted;
        }
        else
        {
            animationGenerator.addHighlightAndExplanationState(6, "False: Node " + toBeDeleted 
+ " has 2 childs" );
            animationGenerator.addHighlightAndExplanationState(7, "Find " + toBeDeleted + " 
successor");
            animationGenerator.addActionNodeState(toBeDeleted, true);
            current = getSuccessor(toBeDeleted);
            animationGenerator.addHighlightAndExplanationState(8, "Swap " + toBeDeleted + " 
with its successor " + current);
            animationGenerator.addSwapState(current, toBeDeleted);
            swap(current, toBeDeleted);
            animationGenerator.addActionNodeState(toBeDeleted, false);
        }
        spliceOut(current);
        searching = true;
        return current.parent();
    }
    /**
     * Splices out the node given as parameter
     * @param node The node to be removed
     * @return
     */
    protected BinaryTreeNode spliceOut(BinaryTreeNode node)
    {
        BinaryTreeNode child;
        int index;
        int childIndex;
        animationGenerator.addHighlightAndExplanationState(10, "Check if node " + node.key + " 
has no left child" );
        if(node.leftChild().isNull())// == null)
        {
            animationGenerator.addHighlightAndExplanationState(11, "True: The right child will 
replace node " + node);
            child = node.rightChild();
            childIndex = 2;
        }
        else
        {
            animationGenerator.addHighlightAndExplanationState(12, node + " has no right 
child" );
            animationGenerator.addHighlightAndExplanationState(11, "The left child will 
replace node " + node);
            child = node.leftChild();
            childIndex = 1;
        }
        animationGenerator.addHighlightAndExplanationState(18, "Check if " + node + " is the 
root of the tree" );
        if(node.isRoot())
        {
            animationGenerator.addHighlightAndExplanationState(19, "True: Set " + child + " as 
the root of the tree");
            setRoot(child);
            index = 0;
        }
        else if(node.isLeft())
        {
            animationGenerator.addHighlightAndExplanationState(20, "False: Check if " + node + 
" is a left child" );
            animationGenerator.addHighlightAndExplanationState(21, "True: Replace " + node + " 
with " + child);
            node.parent().setLeftChild(child);
            index = 1;
        }
        else
        {
            animationGenerator.addHighlightAndExplanationState(20, "False: Check if " + node + 
" is a left child" );
            animationGenerator.addHighlightAndExplanationState(22, "False: Check if " + node + 
" is a right child");
            animationGenerator.addHighlightAndExplanationState(23, "True: Replace " + node + " 
with " + child);
            node.parent().setRightChild(child);
```

```
                index = 2;
            }
            animationGenerator.addDeletionState(node.parent(), node, child, index, childIndex);
            return child;
        }
        /**
         * Swaps the keys of the nodes given as parameters
         * @param node1
         * @param node2
         */
        public void swap(BinaryTreeNode node1, BinaryTreeNode node2)
        {
            Comparable d = node1.key;
            node1.key = node2.key;
            node2.key = d;
        }
        /**
         * Returns the successor node of the node given as parameter
         * @param node
         * @return
         */
        private BinaryTreeNode getSuccessor(BinaryTreeNode node)
        {

            BinaryTreeNode successor = node.rightChild();
            searchNode = createNode(node.key);
             animationGenerator.addSearchingState(searchNode, node, successor);
            while(!successor.leftChild().isNull())// != null)
            {
                animationGenerator.addSearchingState(searchNode, successor,
    successor.leftChild());
                successor = successor.leftChild();
            }
            return successor;
        }
        /**
         * Traverses the tree in using preorder traversal
         * @param n The node currently accessed
         * @param l List for adding the visited nodes
         */
        private void preorder(BinaryTreeNode n, List<BinaryTreeNode> l)
        {
            l.add(n);
            if(!n.leftChild().isNull())
            {

                preorder(n.leftChild(), l);
            }
            if(!n.rightChild().isNull())
            {
                preorder(n.rightChild(), l);
            }
        }
        /**
         * Traverses the tree using inorder traversal
         * @param n The node currently accessed
         * @param l List for adding the visited nodes
         */
        public void inorder(BinaryTreeNode n, List<BinaryTreeNode> l)
        {
            if(!n.leftChild().isNull())
            {
                inorder(n.leftChild(), l);
            }
            l.add(n);
            if(!n.rightChild().isNull())
            {
                inorder(n.rightChild(), l);
            }
        }
        /**
         * Traverses the tree using preorder traversal
         * @param n The node currently accessed
         * @param l List for adding the visited nodes
         */
         private void postorder(BinaryTreeNode n, List<BinaryTreeNode> l)
         {
```

```
    if(!n.leftChild().isNull())
    {
        postorder(n.leftChild(), l);
    }
    if(!n.rightChild().isNull())
    {
        postorder(n.rightChild(), l);
    }
    l.add(n);
}
 /**
 * Returns the nodes of the tree in order
 * @param n The node currently accessed
 * @param l List for adding the visited nodes
 */
public List<BinaryTreeNode> getTreeNodes(BinaryTreeNode n, List<BinaryTreeNode> l)
{
   n.calculatePosition();
   n.setPosition(n.getTargetPosition());
    if(!n.isNull())
    {
        l.add(n);
        getTreeNodes(n.leftChild(), l);
        getTreeNodes(n.rightChild(), l);
    }
    return l;

}
/**
 * Traverse the tree using the traversal type given as parameter
 * @param i The traversal type
 */
public void traversal(int i)
{
    if(isEmpty())
    {
        return;
    }
    animationGenerator.getAnimationStates().clear();
    List<BinaryTreeNode> l = new ArrayList<BinaryTreeNode>();
    switch(i)
    {
        case 1: inorder(root, l);
                break;
        case 2: preorder(root, l);
                break;
        case 3: postorder(root, l);
                break;
    }
    animationGenerator.addSearchNodeState(searchNode,l.get(0), true);
    for(int j = 0; j + 1 < l.size(); j++)
    {
        animationGenerator.addSearchingState(searchNode,l.get(j), l.get(j + 1));
    }
    animationGenerator.addSearchNodeState(searchNode,l.get(l.size() -1), false);
    searchNode = createNode(null);
}
/**
 * Removes all the nodes of the tree
 */
public void removeAll()
{
    animationGenerator.getAnimationStates().clear();
    calculateTotalWidths(root);
    List<BinaryTreeNode> l = new ArrayList<BinaryTreeNode>();
    l = getTreeNodes(root, l);
    animationGenerator.addRemoveAllState(root, l);
}
/**
 * Returns the height of the tree
 * @return
 */
public int getHeight()
{
    calculateHeight(root);
    return root.getHeight();
}
```

```
    /**
     * Calculates the height of the tree
     * @param node
     */
    private void calculateHeight(BinaryTreeNode node)
    {
        if(node.isNull())
        {
            node.setHeight(0);
        }
        else
        {
            calculateHeight(node.leftChild());
            calculateHeight(node.rightChild());
            node.setHeight(Math.max(node.leftChild().getHeight(),
node.rightChild().getHeight()) + 1);
        }
    }
    /**
     * Calculates the total width of each node based in the total widths
     * of its left and right subtrees
     * @param node
     */
    public void calculateTotalWidths(BinaryTreeNode node)
    {
        node.setTotalWidth(0);
        node.setHeight(0);
        if(!node.isNull())
        {
            calculateTotalWidths(node.leftChild());
            calculateTotalWidths(node.rightChild());
            node.setTotalWidth(node.leftChild().getTotalWidth() +
node.rightChild().getTotalWidth());//totalWidth += left.getTotalWidth();
        }
        if(node.getCanvasWidth() > node.getTotalWidth())
        {
            node.setTotalWidth(node.getCanvasWidth());
        }

    }
    /**
     * Calculates the position of each node in the tree
     * @param tnode
     */
    public void calculateSubTree(BinaryTreeNode tnode)
    {
        if(isEmpty())
        {
            return;
        }
        tnode.calculatePosition();
        if(!tnode.isNull())
        {
            calculateSubTree(tnode.leftChild());
            calculateSubTree(tnode.rightChild());
        }
    }
    /**
     * Updates the position for each node of the tree
     * @param node
     * @param rate Indicates how far a node should be moved from
     * its current position to its target position
     */
    public void updateTreePosition(BinaryTreeNode node, double rate)
    {
         if(isEmpty())
        {
            return;
        }
        node.calculatePosition();
        node.updatePosition(rate);
        if(!node.isNull())
        {
            updateTreePosition(node.leftChild(), rate);
            updateTreePosition(node.rightChild(), rate);
        }
    }
```

```
/**
 * Returns true if all the nodes are on their target positions
 * @return
 */
public boolean isUpdated()
{
    isUpdated = true;
    checkUpdated(root);
    return isUpdated;
}
public void checkUpdated(BinaryTreeNode node)
{
    if(node != null)
    {
        isUpdated = isUpdated & !node.isTarget();
        checkUpdated(node.leftChild());
        checkUpdated(node.rightChild());
    }
}
/**
 * Search the tree for the node at position x, y based on the half width of the
 * Tree Panel
 * @param tnode The current node to be checked
 * @param x The x position
 * @param y The y position
 * @param i The half width of the tree panel
 * @return
 */
public BinaryTreeNode searchSubTree(BinaryTreeNode tnode, double x, double y, int i)
{
    if(tnode == null)
    {
        return null;
    }
    if(tnode.isSelected(x, y, i ))
    {
        return tnode;
    }
    BinaryTreeNode node = searchSubTree(tnode.leftChild(),x, y, i) ;
    if(node != null)
    {
        return node;
    }
    node = searchSubTree(tnode.rightChild(),x, y, i) ;
    if(node != null)
    {
        return node;
    }
    return null;
}
/**
 * Inner method used to draws the connection lines between the nodes
 * Used by the TreePanel to drw the tree onto the screen
 * @param node1
 * @param node2
 * @param g
 * @param i
 */
private void connectNodes(BinaryTreeNode node1, BinaryTreeNode node2, Graphics2D g, int i)
{
    int x1 = (int)node1.position.x + node1.getCanvasWidth() / 2;
    int y1 = (int)node1.position.y + node1.getCanvasWidth() / 2;
    int x2 = (int)node2.position.x + node2.getCanvasWidth() / 2;
    int y2 = (int)node2.position.y + node2.getCanvasWidth() / 2;
    if(node2.isNull())
    {
        x2 += +6;//12 for large;
        y2 -=2; //6 for large;
    }
    g.drawLine(x1 + i, y1, x2 + i, y2);
}
/**
 * Inner method for drawing the tree
 * @param node
 * @param g
 * @param i
 */
```

```
    private void paintSubTree(BinaryTreeNode node, Graphics2D g, int i)
    {
        if(!node.isNull())
        {
            connectNodes(node, node.leftChild(),g, i);
            connectNodes(node, node.rightChild(), g, i);
            paintSubTree(node.leftChild(), g, i);
            paintSubTree(node.rightChild(), g, i);
        }
        node.draw(g, i);
    }
    /**
     * Used by the TreePanel class to draw the tree onto the screen
     * @param g
     * @param i The half width of the TreePanel
     */
    public void draw(Graphics2D g, int i)
    {
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
        calculateSubTree(root);
        paintSubTree(root, g, i);
    }
    /**
     * Sets the Animation Generator for creating the Animation States
     * @param sC
     */
    public void setAnimationGenerator(AnimationGenerator sC)
    {
        animationGenerator = sC;
    }
    public AnimationGenerator getAnimationGenerator()
    {
        return animationGenerator;
    }
    /**
     * Returns the file name for the given pseudo code algorithm
     * @param i
     * @return
     */
    public String getPseudoCode(int i)
    {
        switch(i)
        {
            case 1: return SEARCHING_PSEUDOCODE;
            case 2: return INSERTION_PSEUDOCODE;
            case 3: return DELETION_PSEUDOCODE;
            default: return null;
        }
    }
    public void addToArray(List<Comparable> list, BinaryTreeNode node)
    {
        if(!node.isNull())
        {
            list.add(node.key);
            addToArray(list, node.leftChild());
            addToArray(list, node.rightChild());
        }
    }
    /**
     * Returns an array containing all the nodes of the tree
     */
    public List<Comparable> toArray()
    {
        arrayTree.clear();
        addToArray(arrayTree, root);
        return arrayTree;
    }
    @Override
    public BinaryTreeNode clone()
    {
        return root.clone();
    }
    public abstract String getExplanationFile();
    public abstract BinaryTreeNode createNode(Comparable data);
    public abstract BinaryTreeNode createNullNode();
}
```

E2. The `BinaryTreeNode` abstract class

```
package binarySearchTrees;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Rectangle2D;
/**
 * Class representing a Binary tree node
 * @author Victor
 */
public abstract class BinaryTreeNode  implements Comparable<BinaryTreeNode>
{
    protected BinaryTreeNode parent;
    protected BinaryTreeNode left;
    protected BinaryTreeNode right;
    protected Comparable key;
    protected Position position;
    protected Position targetPosition;
    private int totalWidth;
    public static final int canvasHeight = 40;//70
    protected final int HEIGHT_DIFFERENCE = 40;
    private int height;
    protected final int canvasWidth = 28;//56
    private boolean calculateTarget = true;
    private boolean isAction = false;
    public Image actionNode = NodeProperties.getActionNode();
    /**
     * Constructor for creating a Binary tree node
     * @param d
     */
    public BinaryTreeNode(Comparable d)
    {
        key = d;
        position = new Position(NodeProperties.treeEntrance);//0, 50);
        targetPosition = new Position(position);//0, 50);
        if(key !=  null)
        {
            left = createNullNode();
            left.parent = this;
            right = createNullNode();
            right.parent = this;
        }
    }
    /**
     * Constructor for creating a null node
     */
    public BinaryTreeNode()
    {
        this(null);
    }
    /**
     * Returns true if this node is the root of the tree
     * @return
     */
    public boolean isRoot()
    {
        return parent.isNull();
    }
    /**
     * Returns true if this node is a null node
     * @return
     */
    public boolean isNull()
    {
        return key == null;
    }
    /**
     * Returns the key of this node
     * @return
     */
    public Comparable getKey()
    {
        return key;
    }
```

```
/**
 * Sets the key for this node
 */
public void setKey(Comparable key)
{
    this.key = key;
}
/**
 * Returns the parent of this node
 * @return
 */
public BinaryTreeNode parent()
{
    return parent;
}
/**
 * Sets the parent for this node
 * @param p
 */
public void setParent(BinaryTreeNode p)
{
    parent = p;
}
/**
 * Returns the left child of this node
 * @return
 */
public BinaryTreeNode leftChild()
{
    return left;
}
/**
 * Returns the right child of this node
 * @return
 */
public BinaryTreeNode rightChild()
{
    return right;
}
public void setLeftChild(BinaryTreeNode node)
{
    left = node;
    node.parent = this;
}
public void setRightChild(BinaryTreeNode node)
{
    right = node;
    node.parent = this;
}
/**
 * Returns true if this node is a left child
 * @return
 */
public boolean isLeft()
{
    return !isRoot() && parent.leftChild() == this;
}
/**
 * Returns true if this node is a right child
 * @return
 */
public boolean isRight()
{
    return !isRoot() && parent.rightChild() == this;
}
/**
 * Returns true if this node is a leaf
 * @return
 */
public boolean isLeaf()
{
    return left.isNull() && right.isNull();
}
/**
 * Sets the position of this node
 * @param p
 */
```

```
public void setPosition(Position p)
{
    position.setLocation(p);
}
/**
 * Returns the position of this node
 * @return
 */
public Position getPosition()
{
    return position;
}
/**
 * Sets the target position of this node
 * @param p The position to be moved towards
 */
public void setTargetPosition(Position p)
{
    targetPosition = p;//setLocation(p);
}
/**
 * Returns the target position of this node
 * @return
 */
public Position getTargetPosition()
{
    return targetPosition;
}
/**
 * Moves the node to the position of the node given as parameter
 * @param node
 */
public void move(BinaryTreeNode node)
{
    move(node.position.x - position.x, node.position.y - position.y);
}
/**
 * Moves the node to its target position
 */
public void moveToTargetPosition()
{
    calculatePosition();
    translate(targetPosition.x - position.x, targetPosition.y - position.y);
    if(!isNull())
    {
        left.moveToTargetPosition();
        right.moveToTargetPosition();
    }
}
/**
 * Moves the node to the position (x, y)
 */
public void move(double x, double y)
{

    position.translate(x, y);
    if(!isNull())
    {
        left.move(x, y);
        right.move(x, y);
    }
}
/**
 * Moves the node by x and y distance
 */
public void translate(double x, double y)
{
    position.translate(x, y);
}
/**
 * Moves the node closer to its target position
 * @param rate The frame rate value
 */
public void updatePosition(double rate)
{
    double dx = targetPosition.x - position.x;
    double dy = targetPosition.y - position.y;
    if(targetPosition.distance(position) <= 0.1)
    {
        translate(dx, dy);
    }
    else
    {
        translate(rate * dx, rate * dy);
   }
}

/**
 * Updates the position of a deleted node while
 * is falling from the tree
 */
public void updateFallingPosition()
{
    double dx = targetPosition.x - position.x;
    double dy = targetPosition.y - position.y;
    if(targetPosition.distance(position) <= 2)
    {
        translate(dx, dy);
    }
    else
    {
        translate(0, 2);
    }
}
/**
 * Returns the total width of this node
 */
public int getTotalWidth()
{
    return totalWidth;
}
 /**
 * Sets the total width for this node
 */
public void setTotalWidth(int width)
{
    totalWidth = width;
}
/**
 * Sets the node's height
 * @param h The height of the node
 */
public void setHeight(int h)
{
    height = h;
}
/**
 * Returns the height of this node
 */
public int getHeight()
{
    return height;
}
/**
 * Sets whether the node should be updated and moved to its
 * target position (Used when it is dragged by the user)
 * @param flag
 */
public void setCalculateTarget(boolean flag)
{
    calculateTarget = flag;
}
/**
 * Highlights this node as an area of interest
 * @param flag
 */
public void setActionNode(boolean flag)
{
    isAction = flag;
}
/**
 * Returns true if this node is highlighted
 */
public boolean isActionNode()
```

```
    {
        return isAction;
    }
    /**
     * Returns true if this node is not on itstarget position
     * (it is still moving towards its final position)
     */
    public boolean isTarget()
    {
        double dx = targetPosition.x - position.x;
        double dy = targetPosition.y - position.y;
        return Math.abs(dx) >= 2.0 || Math.abs(dy) >= 2.0;//!position.equals(targetPosition);
    }
    /**
     * Calculates the position of this node based on its total width
     * and the total width and position of its parent
     */
    public void calculatePosition()
    {
        if(!calculateTarget)
        {
            return;
        }
        if(!isRoot())
        {
            double x = parent.position.x - (parent.getTotalWidth() - getTotalWidth()) / 2;
            double y = parent.position.y + getHeightDifference();
            if(isRight())
            {
                x += parent.left.getTotalWidth();
            }
            setTargetPosition(new Position( x, y));
        }
        else
        {
            setTargetPosition(new Position(0, 50));
        }
    }
    /**
     * Returns the difference in height between a parent
     * and its children
     * (A null node is closer to its parent than a normal one)
     */
    public int getHeightDifference()
    {
        if(!isNull())
        {
            return HEIGHT_DIFFERENCE;
        }
        else
        {
            return HEIGHT_DIFFERENCE -  10;
        }
    }
    /**
     * Returns true if this node is selected
     * (for drag and drop)
     */
    public boolean isSelected(double x, double y, int i)
    {
        if(!isNull())
        {
            return new Ellipse2D.Double(position.x + i, position.y, getCanvasWidth(),
getCanvasWidth()).contains(x, y);
        }
        else
        {
            return new Rectangle2D.Double(position.x + i + 9, position.y, 7, 7).contains(x,
y);
        }
    }
    /**
     * Returns the diameter of this node
     * @return
     */
    public int getCanvasWidth()
    {
        if(!isNull())
        {
            return canvasWidth;
        }
        else
        {
            return canvasWidth / 2;
        }
    }
    @Override
    public String toString()
    {
        if(key == null)
        {
            return "null";
        }
        return key.toString();
    }
    /**
     * Compares this node with node given as parameter
     * @param node
     */
    public int compareTo(BinaryTreeNode node)
    {
        if(isNull() || node.isNull())
        {
            return -2;
        }
        else
        {
            return key.compareTo(node.key);
        }
    }
    /**
     * Draws this node as an action node
     * (It is drawn on a different image)
     */
    public void drawActionNode(Graphics2D g, int i)
    {
         g.drawImage(NodeProperties.getActionNode(), (int)position.x + i, (int)position.y,
null);
         g.drawString(key.toString(), (int)position.x + 7 + i, (int)position.y + 17);
    }
    /**
     * Draws this node as an empty rectangle if
     * it is a null node
     */
    public void drawNullNode(Graphics2D g, int i)
    {
        g.setColor(Color.white);
        g.fillRect((int)position.x   + i + 9 , (int)position.y , 7, 7);//14,14
        g.setColor(Color.black);
        g.drawRect((int)position.x   + i + 9, (int)position.y , 7, 7);
    }
    @Override
    public abstract BinaryTreeNode clone();
    public abstract void draw(Graphics2D g, int i);
    public abstract BinaryTreeNode createNullNode();
    public abstract void drawSearchNode(Graphics2D g, int i);
    public abstract void drawFallingNode(Graphics2D g, int i);

}
```

E3. The `TreePanel` class

```
package animationEngine;
import binarySearchTrees.BinaryTree;
import binarySearchTrees.BinaryTreeNode;
import binarySearchTrees.Position;
import data.Globals;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JPanel;


/**
 * Class representing the tree visualization engine
 * TreePanel is a Thread class that runs in background
 * and periodically draws the current Binary tree on the screen
 * @author Victor
 */
public class TreePanel extends JPanel implements Runnable,  MouseListener, MouseMotionListener
{
    private BinaryTree tree;
    private BinaryTreeNode selectedNode;
    private Thread animator;
    private BinaryTreeNode searchNode;
    private BinaryTreeNode fallingNode;
    private int speed;
    private Dimension panelSize;
    private double frameRate;
    private  List<BinaryTreeNode> fallingNodes;
    private boolean terminated;
    private Position clickedPosition;

    public TreePanel(BinaryTree tree)
    {
        this.tree = tree;
        terminated = false;
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
        animator = new Thread(this);
        speed = Globals.ANIMATION_DELAY;
        setOpaque(false);
        panelSize = new Dimension(getWidth(), getHeight());
        frameRate = 0.05;
        fallingNodes = new ArrayList<BinaryTreeNode>();
        animator.start();
    }
    @Override
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        if(tree != null && !tree.isEmpty())
        {
            setPreferredSize(new Dimension(tree.getRoot().getTotalWidth(), (tree.getHeight() +
1) * 40 + 10));
            tree.draw(g2, getWidth() / 2);
        }
        if(searchNode != null)
        {
            //animates the search node during searching
            searchNode.drawSearchNode(g2, getWidth()/ 2);
        }
        if(fallingNode != null)
        {
            //animates the deleted node while is falling from the tree
            fallingNode.draw(g2, getWidth()/ 2);
        }
        if(fallingNodes != null)
        {
            //animate the nodes of the tree while they are falling
            for(int i=0; i< fallingNodes.size(); i++)
            {
                BinaryTreeNode n = fallingNodes.get(i);
                n.updateFallingPosition();
                n.draw(g2, getWidth()/ 2);
                if(!n.isTarget())
                {
                    fallingNodes.remove(i);
                }
            }
        }
    }
    public void setFrameRare(double r)
    {
        frameRate = r;
    }
    public void setTree(BinaryTree t)
    {
        tree = t;
    }
    public void terminate()
    {
        terminated = true;
    }
    public void run()
    {
        while(!terminated)
        {
            tree.calculateTotalWidths(tree.getRoot());
            Dimension treeSize = new Dimension(tree.getRoot().getTotalWidth(),
(tree.getHeight() + 1) * 40 + 10);
            //If the size of the tree exists the size of the panel
            if(!panelSize.equals(treeSize))
            {
                panelSize = treeSize;
                setPreferredSize(treeSize);
                //A scroll bar becomes visible
                revalidate();
            }
            //update the positions of each node
            tree.updateTreePosition(tree.getRoot(), frameRate);
            if(searchNode != null)
            {
                searchNode.updatePosition(frameRate + 0.02);
            }
            if(fallingNode != null)
            {
                fallingNode.updateFallingPosition();
                if(!fallingNode.isTarget())
                {
                    fallingNode = null;
                }
            }
            repaint();
            try
            {
                Thread.sleep(speed);
            }
            catch (InterruptedException ex){}
        }
    }
    public void setFallingNodes(List<BinaryTreeNode> l)
    {
        if(l != null)
        {
            fallingNodes = new ArrayList<BinaryTreeNode>();
            for(int i=0; i< l.size(); i++)
            {
                fallingNodes.add(l.get(i).clone());
            }
        }
        else
        {
            fallingNodes = l;
        }
    }
    public void mouseClicked(MouseEvent e){}
    public void mousePressed(MouseEvent e)
    {
        //find the selected node
        selectedNode = tree.searchSubTree(tree.getRoot(), e.getX(), e.getY(), getWidth() / 2);
        if(selectedNode != null)
        {
            //change the colour of the selected node
            selectedNode.setActionNode(true);
            //Allows the user to move this node where its children positions
            //will be calculated relative to this node instead of the root
            selectedNode.setCalculateTarget(false);
            clickedPosition = new Position(e.getX() - selectedNode.getPosition().x -
getWidth()/2, e.getY() - selectedNode.getPosition().y );
```

```
        }
    }
    public void mouseReleased(MouseEvent e)
    {
        if(selectedNode != null)
        {
            selectedNode.setActionNode(false);
            selectedNode.setCalculateTarget(true);
        }
    }
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
    public void mouseDragged(MouseEvent e)
    {
        if(selectedNode != null)
        {
            selectedNode.setPosition(new Position(e.getX() - clickedPosition.x - getWidth() /
2, e.getY() - clickedPosition.y));
            selectedNode.setTargetPosition(new Position(e.getX() - clickedPosition.x -
getWidth() / 2, e.getY() - clickedPosition.y));
        }
    }
    public void mouseMoved(MouseEvent e){}
    public void setSearchNode(BinaryTreeNode searchNode)
    {
        this.searchNode = searchNode;
    }
    public void setFallingNode(BinaryTreeNode node)
    {
        fallingNode = node;
    }
    public void setSpeed(int i)
    {
        speed = i;
    }
    public Dimension getPreferredScrollableViewportSize()
    {
       return new Dimension(getWidth() + 100, getHeight() + 500);
    }
    public int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int
direction)
    {
        return 10;
    }
    public int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int
direction)
    {
        return 10;
    }
    public boolean getScrollableTracksViewportWidth()
    {
        return false;
    }
    public boolean getScrollableTracksViewportHeight()
    {
        return false;
    }
}
```

E4. The `AnimationController` class

```
package animationEngine;

import animationGenerator.History;
import binarySearchTrees.AvlTree;
import binarySearchTrees.BinarySearchTree;
import binarySearchTrees.BinaryTree;
import binarySearchTrees.NodeProperties;
import binarySearchTrees.Position;
import binarySearchTrees.RedBlackTree;
import binarySearchTrees.SplayTree;
import data.Globals;
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;
import javax.swing.JButton;
import javax.swing.JProgressBar;
import javax.swing.JTextField;
import javax.swing.Timer;
import animationGenerator.AnimationState;
import animationGenerator.ChangePseudoCodeState;
import animationGenerator.AnimationGenerator;
/**
 * The class for controlling the animation and
 * all the buttons of the GUI
 * @author Victor
 */
public class AnimationController
{
     private Stack<AnimationState> nextStack, backStack;
    private Timer timer;
    private BinaryTree tree;
    private  TreePanel treePanel;
    private PseudoPanel pseudoPanel;
    private ExplanationPanel explanationPanel;
    public List<History> history;
    private JButton next;
    private JButton back;
    private JButton play;
    private JButton pause;
    private JButton reset;
    private JButton fastForward;
    private int treeType;
    private boolean explanationsDisabled;

    public AnimationController(BinaryTree tree, TreePanel treePanel, PseudoPanel p,
ExplanationPanel expl)
    {
        this.tree = tree;
        this.treePanel = treePanel;
        pseudoPanel = p;
        explanationPanel = expl;
        explanationsDisabled = false;
        history = new ArrayList<History>();
        setTimer();
    }
    public void init()
    {
        treeType = 1;
        loadCasePerformanceData(Globals.AVERAGE_CASE_PERFORMANCE);
    }
    public void setTimer()
    {
        timer = new Timer(Globals.ALGORITHM_STEP_DELAY, new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                stepForwards();
            }
        });
    }
    public void resetStates()
    {
        AnimationState anOp;
        while(!backStack.empty())
        {
            anOp = backStack.pop();
            if(anOp.isModi]'[fying())
            anOp.undo();
            nextStack.push(anOp);
        }
        pseudoPanel.clear();
        explanationPanel.clear();
        treePanel.setSearchNode(null);
        updateButtons();
    }
```

```
public void redoStates()
{
    AnimationState anOp;
    if(timer.isRunning())
    {
        timer.stop();
    }
    while(!nextStack.empty())
    {
        anOp = nextStack.pop();
        if(anOp.isModifying())
        anOp.redo();
        backStack.push(anOp);
    }
    treePanel.setSearchNode(null);
    updateButtons();
    pseudoPanel.clear();
    explanationPanel.clear();
}
public void stepForwards()
{
    AnimationState anOp;
    if(!nextStack.empty())
    {
        if(explanationsDisabled)
        {
            anOp = nextStack.pop();
            try
            {
                while(anOp.canBeDisabled())
                {
                    backStack.push(anOp);
                    anOp = nextStack.pop();
                }
                anOp.redo();
                backStack.push(anOp);
            }
            catch(Exception e){}
        }
        else
        {
            anOp = nextStack.pop();
            anOp.redo();
            backStack.push(anOp);
        }
    }
    if(timer.isRunning())
    {
        if(nextStack.isEmpty())
        {
            timer.stop();
            updateButtons();
            play.setEnabled(false);
            pause.setEnabled(false);
            reset.setEnabled(true);
        }
    }
    else
    {
        updateButtons();
    }
}
public void stepBackwards()
{
    AnimationState anOp;
    if(!backStack.empty())
    {
        if(explanationsDisabled)
        {
            anOp = backStack.pop();
            try
            {
                while(anOp.canBeDisabled())
                {
                    nextStack.push(anOp);
                    anOp = backStack.pop();
                }
                anOp.undo();
                nextStack.push(anOp);
            }
            catch(Exception e){}
        }
        else
        {
            anOp = backStack.pop();
            anOp.undo();
            nextStack.push(anOp);
        }
    }
    updateButtons();
}
private void updateButtons()
{
    if(nextStack.isEmpty())
    {
        next.setEnabled(false);
        fastForward.setEnabled(false);
        play.setEnabled(false);
        pause.setEnabled(false);
    }
    else
    {
        play.setEnabled(true);
        next.setEnabled(true);
        fastForward.setEnabled(true);

    }
    if(backStack.isEmpty())
    {
        back.setEnabled(false);
        reset.setEnabled(false);
    }
    else
    {
        back.setEnabled(true);
        reset.setEnabled(true);
    }
}
public void playAnimation()
{
     play.setEnabled(false);
     pause.setEnabled(true);
     next.setEnabled(false);
     back.setEnabled(false);
     reset.setEnabled(false);
     fastForward.setEnabled(false);
     timer.setInitialDelay(0);
     timer.start();
}
public void stopAnimation()
{
    notifyFinishedAnimation();
}
public void pauseAnimation()
{
     play.setEnabled(true);
     pause.setEnabled(false);
     next.setEnabled(true);
     fastForward.setEnabled(true);
     back.setEnabled(true);
     reset.setEnabled(true);
     timer.stop();
}
public void resetAnimation()
{
     play.setEnabled(true);
     pause.setEnabled(false);
     next.setEnabled(true);
     back.setEnabled(true);
     reset.setEnabled(false);
     fastForward.setEnabled(true);
     timer.stop();
    resetStates();
}
```

```
private void notifyFinishedAnimation()
{
    if(timer.isRunning())
    {
        timer.stop();
    }
}
public void insert(Comparable key)
{
    redoStates();
    tree.insert(key);
    history.add(new History(key, 1));
    resetStates();
    explanationPanel.addOperationExplanation("Inserting " + key + ": ");
    next.setEnabled(true);
    fastForward.setEnabled(true);
    back.setEnabled(false);
    play.setEnabled(true);
    pause.setEnabled(false);
    reset.setEnabled(false);
}
public void delete(Comparable key)
{
    redoStates();
    tree.delete(key);
    history.add(new History(key, 2));
    resetStates();
    explanationPanel.addOperationExplanation("Deleting " + key + ": ");
    next.setEnabled(true);
    fastForward.setEnabled(true);
    back.setEnabled(false);
    play.setEnabled(true);
    pause.setEnabled(false);
    reset.setEnabled(false);
}
public void find(Comparable key)
{
    redoStates();
    tree.find(key);
    history.add(new History(key, 0));
    resetStates();
    explanationPanel.addOperationExplanation("Searching for " + key + ": ");
    next.setEnabled(true);
    fastForward.setEnabled(true);
    back.setEnabled(false);
    play.setEnabled(true);
    pause.setEnabled(false);
    reset.setEnabled(false);
}
public void traverse(int i)
{
    String explanation;
    if(i == 1)
    {
        explanation = "Inorder Traversal";
    }
    else if(i == 2)
    {
        explanation = "Preorder Traversal";
    }
    else
    {
        explanation = "Postorder Traversal";
    }
    redoStates();
    tree.traversal(i);
    history.add(new History(null, 4, i));
    resetStates();
    explanationPanel.addOperationExplanation(explanation);
    fastForward.setEnabled(true);
    back.setEnabled(false);
    play.setEnabled(true);
    pause.setEnabled(false);
    reset.setEnabled(false);
}
public void removeAll()
{
        redoStates();
        tree.removeAll();
        history.add(new History(null, 3));
        changeTree(treeType);
        explanationPanel.addOperationExplanation("Removing all the nodes");
    }
    public void changeTree(int i)
    {
        switch(i)
        {
            case 1: tree = new BinarySearchTree();
                    break;
            case 2: tree = new AvlTree();
                    break;
            case 3: tree = new RedBlackTree();
                    break;
            case 4: tree = new SplayTree();
                    break;
        }
        treeType = i;
        timer.stop();
        AnimationGenerator opC = new AnimationGenerator(tree, treePanel, pseudoPanel,
explanationPanel);
        tree.setAnimationGenerator(opC);
        for(int j = 0; j < history.size(); j++)
        {
            History h = history.get(j);
            Comparable key = h.getKey();
            int op = h.getOperation();
            switch(op)
            {
                case History.SEARCHING: tree.find(key);
                                        break;
                case History.INSERTION: tree.insert(key);
                                        break;
                case History.DELETION: tree.delete(key);
                                        break;
                case History.REMOVE_ALL: tree.removeAll();
                                        for(int k = j; k >= 0; k--)
                                        {
                                            history.remove(k);
                                        }
                                        break;
                case History.TRAVERSAL: tree.traversal(h.getTraversal());
                                        break;
            }
        }
         pseudoPanel.clear();
         explanationPanel.clear();
         treePanel.setTree(tree);
         treePanel.setSearchNode(null);
         play.setEnabled(false);
         pause.setEnabled(false);
         next.setEnabled(false);
         back.setEnabled(true);
         reset.setEnabled(true);
         fastForward.setEnabled(false);
         nextStack = new Stack<AnimationState>();
         backStack = opC.getAnimationStates();
    }
    public boolean treeIsUpdated()
    {
        return tree.isUpdated();
    }
    public void loadCasePerformanceData(String file)
    {
        InputStream is = this.getClass().getResourceAsStream(file); //Try get the file as a
stream
        if(is != null) //Check if input stream was created (if not, then filename didnt
exist).
        {
            Scanner in = new Scanner(is);
            history.clear();
            while(in.hasNextLine())
            {
                history.add(new History(in.nextInt(), 1));
            }
            changeTree(treeType);
        }
    }
    public void setNodeColor(int color)
    {
        NodeProperties.setNodeColor(treeType, color);
    }
    public void setSpeed(int value)
    {
        timer.setDelay(Globals.ALGORITHM_STEP_DELAY / value);
    }
    public void setAnimationButtons(JButton n, JButton b, JButton pl, JButton paus, JButton r,
JButton fastF)
    {
        next = n;
        back =b;
        play =pl;
        pause = paus;
        reset = r;
        fastForward = fastF;
    }
    public void setFrameRate(int value)
    {
        treePanel.setFrameRare((double)value / 20.0);
    }
    public void setTreeEntrance(Position p)
    {
        NodeProperties.setTreeEntrance(p);
        changeTree(treeType);
    }
    public void showTreeExplanation(Component frame)
    {
        new TreeExplanation(frame, tree.getExplanationFile());
    }
    public void disablePseudoCodeandExplanations(boolean b)
    {
        explanationsDisabled = b;
        if(!b)
        {
            AnimationState anOp = null;
            int i;
            //Find the and display the pseudocode for the current animation state
            for(i = nextStack.size()-1; i >= 0 && !((anOp = nextStack.get(i)) instanceof
ChangePseudoCodeState); i--);
            if(i >= 0)
            {
                anOp.undo();
            }
        }
        else
        {
            pseudoPanel.clear();
            explanationPanel.clear();
        }
    }
}
```