Teaching Interactive Computer Science

Group 7

Alexander Kjellén
Björn Delin
Erik Skogby
Jan-Erik Bredahl
Joakim Israelsson

Design Document

Introduction

The purpose of the Design Document is to give a detailed overview of the design of Teaching Interactive Computer Science (TICS). The document is intended to be read by the developers of TICS and by people who are interested in making a similar system or changes to TICS. To get a better understanding of the Design Document, the reader should have read the Requirements Document before proceeding with the document. The Design Document includes all necessary information so that it can be used as a tool to build the system.

The Design Document contains information about the system overview that also includes an overall architecture and detailed architecture of the system. It also includes assumptions and dependencies and general constrains of the system and information on how the user interface will look like. The document also has all classes and methods that are to be implemented in the system and how they are going to be tested.

Glossary

- **A&D** – Algorithms and data structures
- **Algorithms** – A definite list of well defined instructions for completing a task. In computer science there are lots of known algorithms that that is proved to meet the goal in an efficient way.
- **Animation** – A rapid display of sequence of images in order to create an illusion of movement. In this program the animation will consist of objects moving from one place to another.
- **Computing Science -** The study of the theoretical foundations of information and computation and their implementation and application in computer systems.
- **Data structures** – A way of storing data in a computer so that it can be used efficiently.
- **Function –** A portion of code within a larger program which performs a specific task, in our case it runs a specific algorithm.
- **Singly linked list –** A data structure consisting of a sequence of nodes each containing arbitrary data fields and one reference pointing to the next node.
- **Sorted binary tree –** A data structure consisting of nodes where every node has at most two children. The “left” pointer points to a node with a lower value than itself and the “right” pointer points at a node with at least as high value.
- **Vector** – A one dimensional data structure consisting of a group of elements that are accessed by indexing.
- **Visualization** – A technique for creating animations to communicate a message. In this case it means creating an animation that will describe an algorithm or a data structure.
- **Index –** An integer which identifies an element or data structure which enables fast lookup.
- **Parameter** – A variable which takes on the meaning of a corresponding argument passed in a call to a function.
- **TICS** – Teaching interactive computer science, the name of our system.

System Overview

General Description

TICS is a computer software aimed at students studying computer science. This is a system that the students will be able to run on their computers. This means that TICS is independent program which does not need a server or an internet connection to run. The program shall be written in Java and run as a java application consisting of various classes divided into modules.

The user shall be able to choose between different data structures and algorithms and see them in action in a logical and user friendly way using arrows and boxes. This is to help them better understand the data structures.

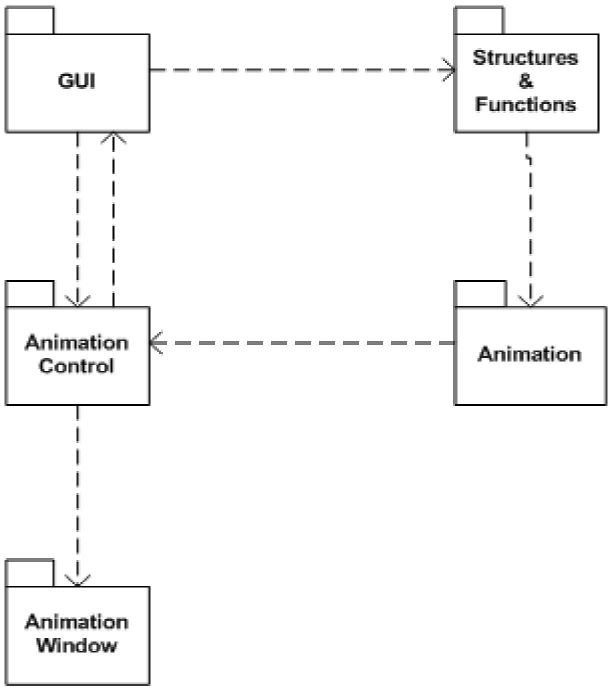
The base for our system will be built with Java 2D and Java Swing but we will need to include other packages as well. Java 2D will be used to draw the data structures and Java Swing will be used for the layout and menus of the program. We will also need own data structures to keep track of what to draw and how to draw it.

Our design approach

We found this approach on the layout of the system to be the best since it is easy to understand and we wanted a clean design. What we have done is to divide the system into five modules as shown in next section. Each module is responsible for different parts of the system, and each module consists of several classes.

Overall Architecture Description

TICS consist of five modules with different responsibilities.



The arrows represent the data flow of the program

Animation Window

This is the part of the GUI that draws the animation. The Animation Window modules responsibility is to draw the animations.

Animation Control

Animation Control keeps track of the animation and is the module that enables us to control it.

Animation

The Animation module receives all its information from the Structures & Functions module which it uses to create an animation list.

Structures & Functions

Structures & Functions contains all the structures and functions that this program can visualize.

GUI

The GUI module handles the graphical interface except for the drawing area where the animations are drawn.

Intermodal communication

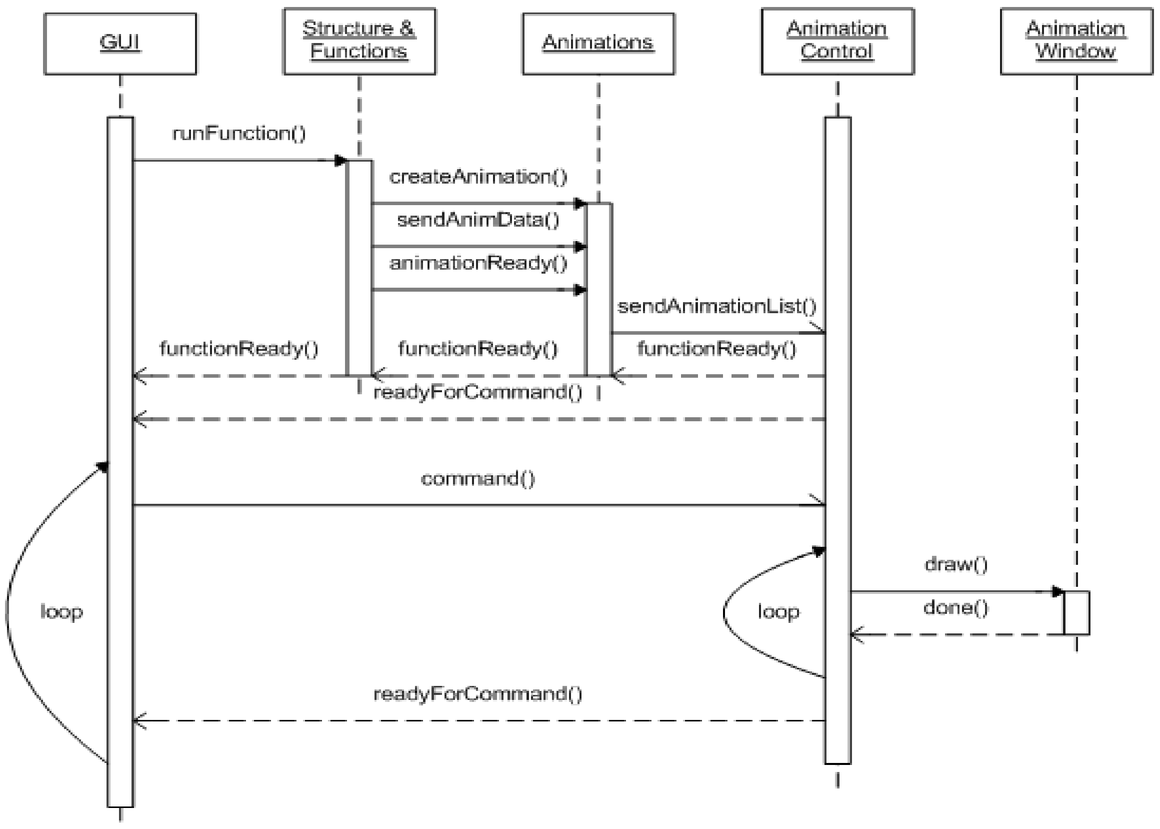
The GUI modules will most of the time wait for a command from the user. When it receives a command, such as a mouse click on a button, one of three things can happen:

- It can satisfy the request itself, for example if the user requests a help menu.
- If it is a request to load a data structure or run an algorithm, the GUI will transfer the request to the Structures & Functions module, which will run the algorithm and send information to the Animation module. This module will then create an animation list visualizing the algorithm. This list is sent to the Animation Control that animates it and sends images to the Animation Window.
- Finally, the request from the user may be to affect the current animation in some way, i.e. play. In this case, the request is sent to the Animation Control module which takes the appropriate action. This can result in the Animation Control module sending commands to the GUI (enable or disable buttons) and the Animation Window module (to redraw the animation).

Detailed Architecture

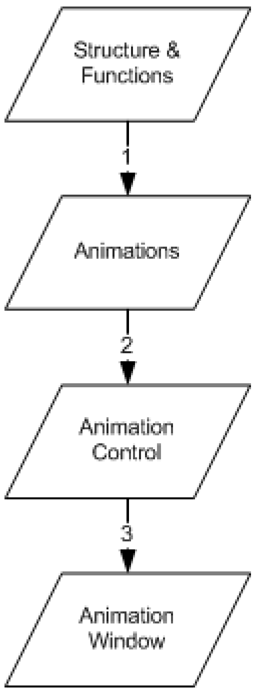
Sequence diagram

The following diagram shows what happens when the user tells TICS to run an algorithm, which is the main use of the system.



Data flow

Each number describes an arrow in the picture to the right.

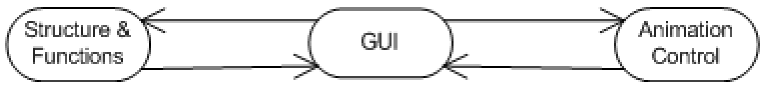


1. As a function is executed in the structure & function module data is sent to the animation module. The data includes information on what happens when a function is executed.

2. The animation module has now created an animation list from the data that the structure & functions module sent. The data sent to Animation Control is the animation list which consists of the following: a start state with information on how the structure looks like before the animation begins. This includes sizes, positions, relations etc. The animation list also contains a list of events, or changes to the structure. This could include a object creation, deletion or moving.

3. The Animation Control interprets the animation list received from the Animation module. It creates a model of the structure according to the animation list start state. This model is then modified according to the events in the event list. It then creates frames or snapshot of this animation and sends to the Animation Window module. These frames contain the current removed state of the model, with information such as sizes, positions, relations, etc.

Control



The GUI needs to collect information from two other modules besides itself. The GUI needs to collect data from the Structure & Function module when a user tries to load a data structure or an algorithm. The data consists of available data structure or algorithms that the user can choose between. The GUI also has to collect data from the Animation Control module so that the GUI knows which Animation Control buttons that should be clickable.

Design Considerations

Assumptions and Dependencies

The main assumptions we have made is that the user is a fairly advanced computer user. This comes from the fact that our software is an aid in teaching computer science, and anyone who is learning computer science is probably quite proficient with a computer. We have also assumed that our end-user speaks English.

To make the implementation easier for us we have restricted ourselves to making sure it works on a 0.8GHz PC computer running Windows XP and Java runtime environment version 1.5.

The main changes in functionality are an increase or decrease of the number of data structures and algorithms visualized. It is easy to add or remove a method without affecting the rest of the code base, so our software is very scalable in that regard.

General Constraints

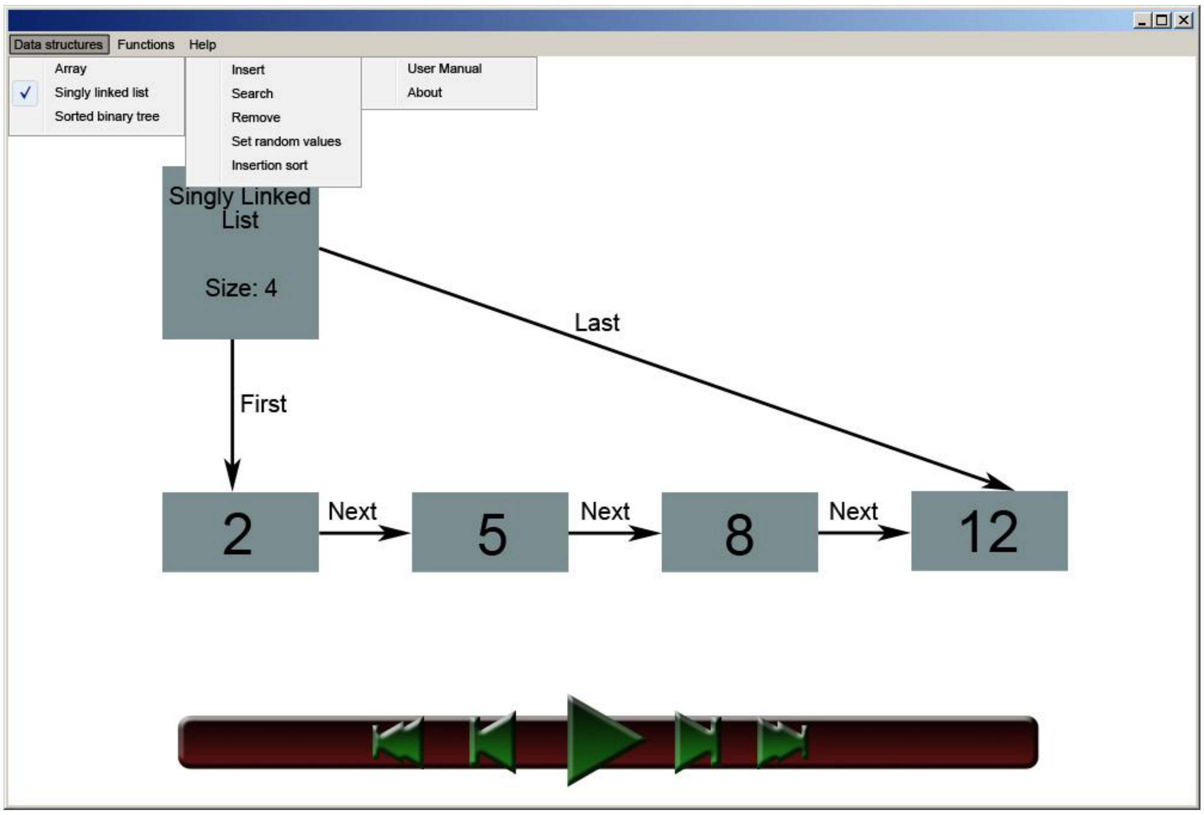
Since our program is a desktop application with that does not use any network, has no sensitive or crucial information or does any heavy computation, we have plenty of room inside our constraints. We have no need for security, we do not have to communicate with anyone else, and we do not store any permanent data.

We have some standards to comply with, in that our data structures should look like they generally do in computer science literature. Our performance demands are simply that the system should animate smoothly. With smoothly, we mean at least 20 drawn frames per second. Our testing includes a range of tests that include performance tests, since we test with the biggest data structures possible. The hardest goal we have is that of making sure people actually feel that this software is helpful and useful. To test this we will let people that has never been exposed to TICS use our software at regular intervals.

Graphical User Interface

General GUI information

This is how our main Graphical User Interface (GUI) will look like:



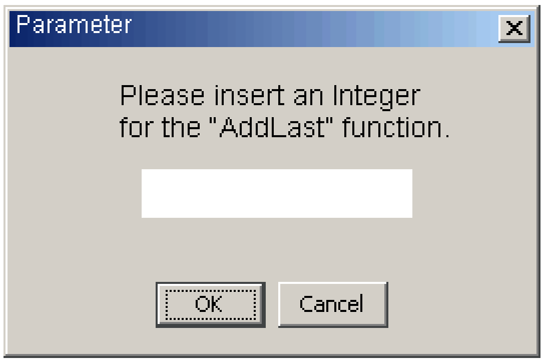
It consists of a menu bar, an animation window and an animation control area. The animation window is just a big white space used for drawing the animations, which will be described later on. The animation control area contains five buttons named StepFirst, StepBackwards, Play/Pause, StepForward and StepFinish. These correspond to the animation control requirements in the requirements document. StepFirst corresponds to “Stop”, StepBackwards to “Step Backwards”, Play/Pause to “Play” and “Pause”. The Play/Pause button will display Pause when the animation is running and Play when the animation is paused. StepForward corresponds to “Step Forward” and finally StepFinish to “Finish”.

The menu bar will have three drop down menus named: Data Structures, Functions and Help. The menu button Data Structures will have the three available data structures: Array, Linked list and Binary tree described in the requirements documents. When you click on one of these the data structure will appear in the animation window. The Functions menu, which previously was empty, now contains all the available functions for the currently selected data structure. If no data structure is currently selected, the Functions menu is empty. The help menu contains: User Manual and About which describes current version and information about the developer.

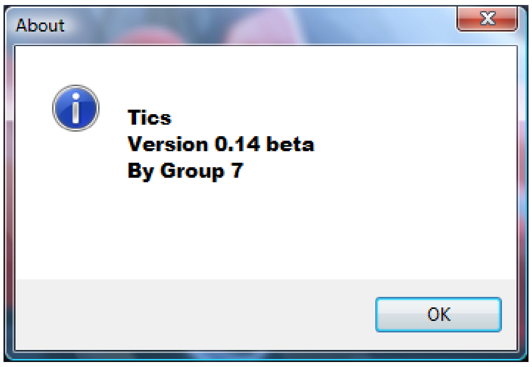
Specified GUI information

Popup windows

When a user clicks on dropdown button Functions and selects a function which takes a parameter, a popup window that looks similar to the left window below appears. It is called Parameter and waits for the user to specify a parameter and click OK. If the parameter is invalid the same window will appear again but this time the text “invalid parameter” will show in red after the rest of the text.



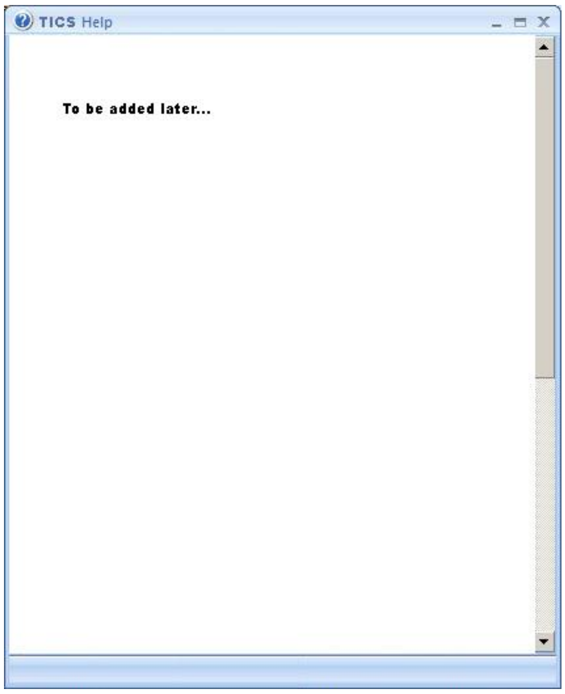
Please insert an Integer for the "AddLast" function.



Tics
Version 0.14 beta
By Group 7

Note: The pictures here are just examples of how the popup windows might look like.

The help menu has two choices as said before; User Manual and About. The About popup will look similar to the picture shown above to the right. The User Manual will look something like the picture below, but will contain text that explains how the program works.



To be added later...

Animation Window

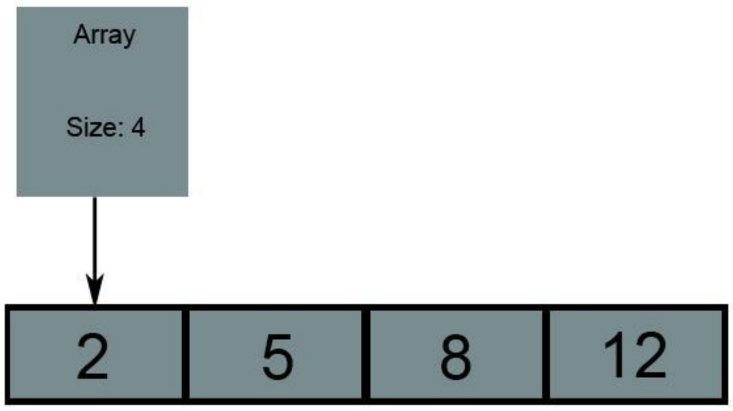
The animation window is where all the animations will take place. It is the main window of the program and also the largest. It will be blank when the program starts and stay that way until the user chooses to load a data structure from the Data Structure menu. When a data structure has been loaded, it will appear in the animation window. When the user activates a function and uses the animation control buttons, the animation window will show an animation representing the function being executed on the current data structure.

Linked List

The Linked List structure is shown in the first GUI picture.

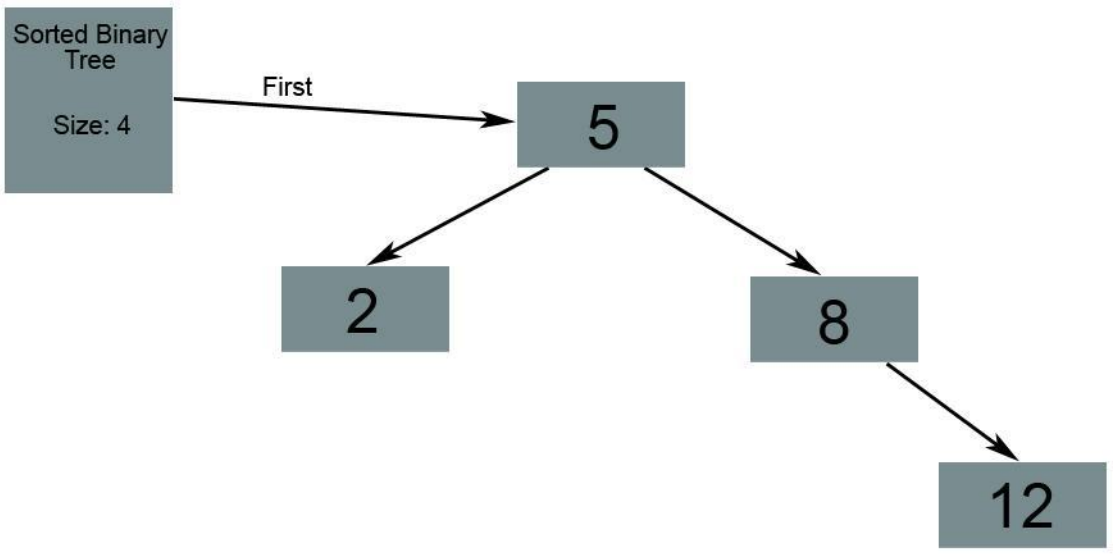
Array

The Array structure will consist of several boxes lying next to each other like shown in the figure below.



Sorted Binary Tree

The Sorted Binary Tree structure will consist of several boxes lying separated with arrows connecting them like shown in the figure below.



Detailed Design

CRC Cards

This is a template what the collaborator cards include:

Name of the Class	
Responsibilities	**Collaborators**

MainWindow	
Keep track of menus and available menu options. Sends on commands given by the user.	DataStructures AnimationController

DataStructures	
Send a structure. Keep track of available structures and functions. Check CurrentState if a structure is loaded. Create a Structure (with info from CurrentState). Current structure.	Structure AnimationCreator CurrentState MainWindow

Animation	
Keep track of each step of an algorithm. Keep track of every updated state of the structure. Keep a Structure.	Structure

AnimationWindow	
Needs to update a certain number of frames per second. Draw each object given by the FrameList and on its right place.	Frame MainWindow

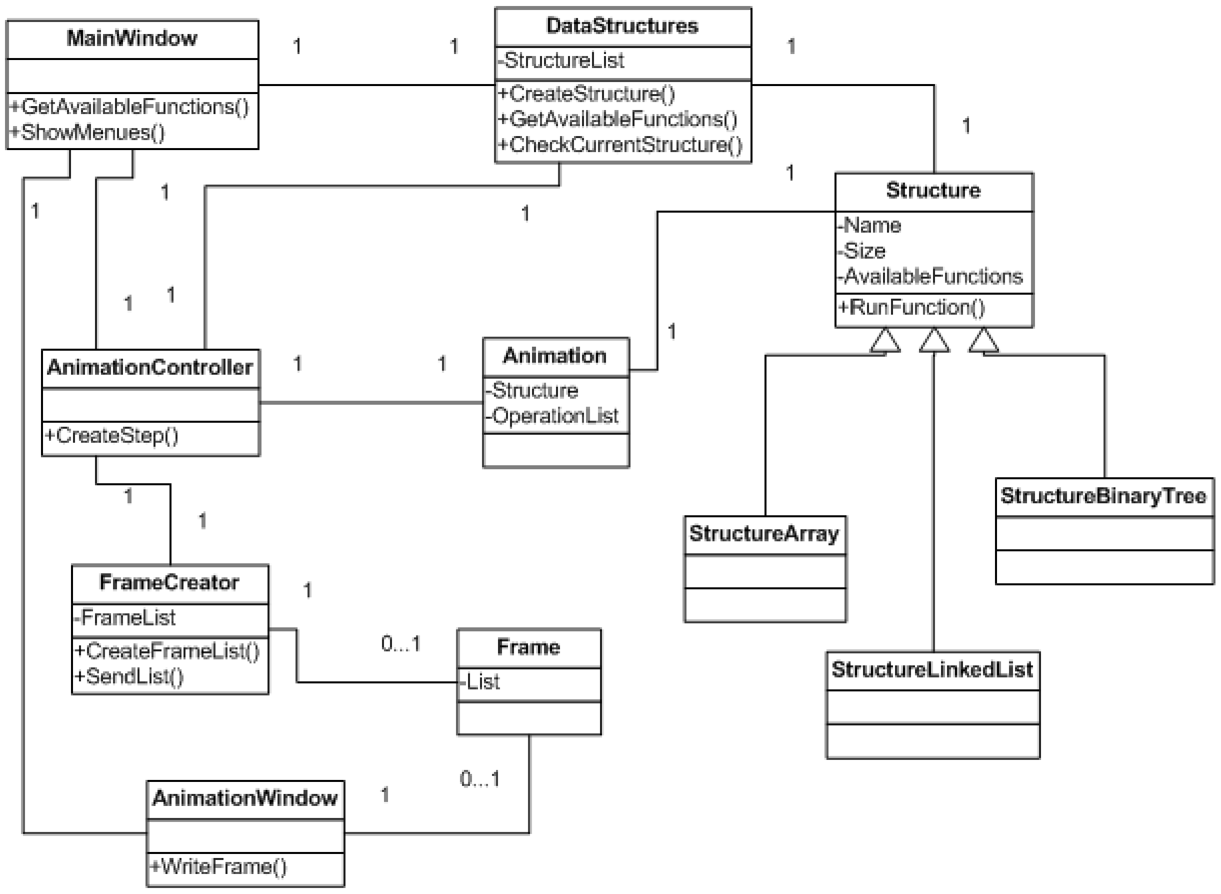
AnimationController	
Keep CurrentState up to date.	CurrentState
Sends each animation step to the FrameCreator.	MainWindow
Keep track of what buttons are clickable and sends that information to the main Window.	FrameCreator
	Animation

Structure	
Knows name	Animation
Knows size	
Knows available functions.	
Knows how the structure should be modeled.	
Knows what function to run.	
Current state	
Create an AnimationList	

Frame	
Keeps a list of each “frame” that is needed for a operation on a datastructure.	None
Each “frame” consists of coordinates and object types.	

FrameCreator	
Given an operation, create an animaion from it by creating a FrameList.	Frame
Keep track of the structure and how objects move.	

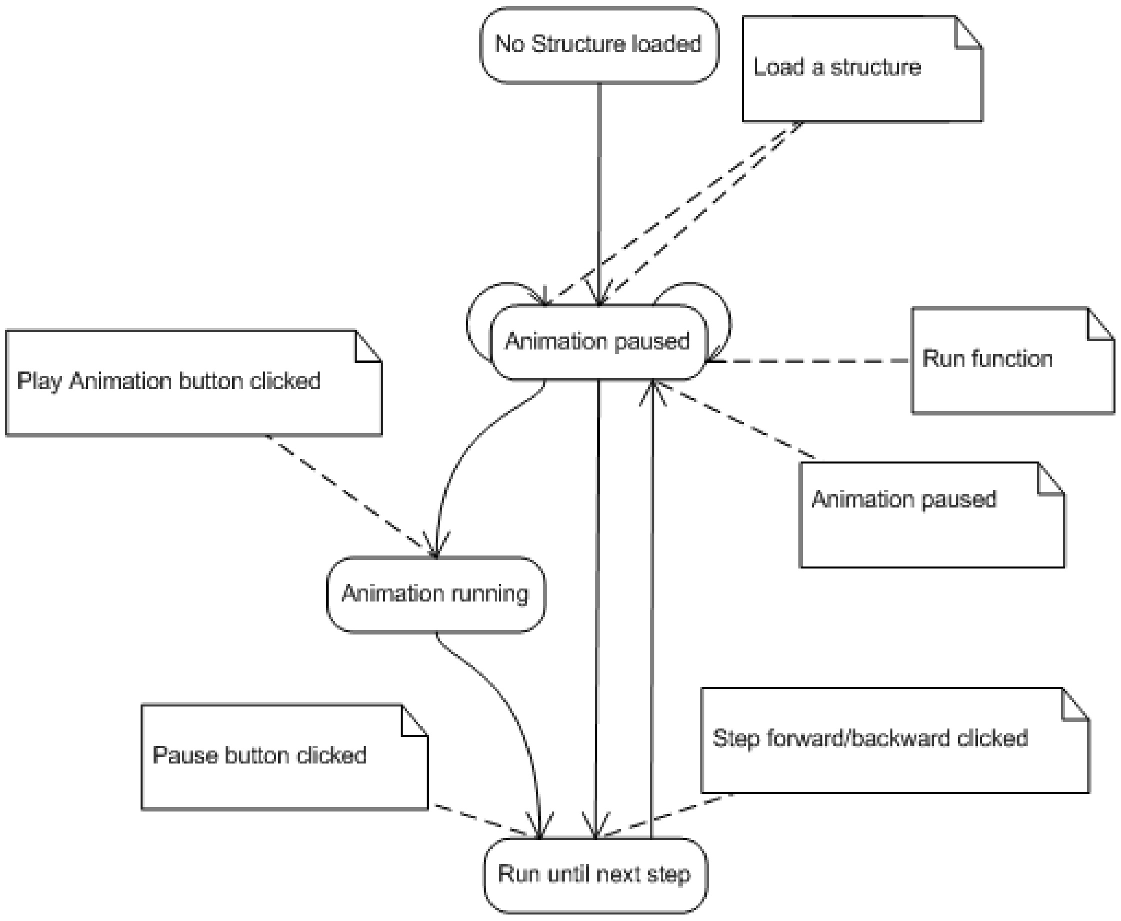
Class diagram



This is what the numbers in the diagram means:

0..1	No instances, or one instance (optional, may)
1	Exactly one instance
0..* or *	Zero or more instances
1..*	One or more instances (at least one)

State diagram



This state diagram describes how the animation system works.

The ”No Structure loaded” state is the start state. It is the only state in which no animation or visualization is visible. As soon as a visualization is visible, the application will always show one until it is restarted.

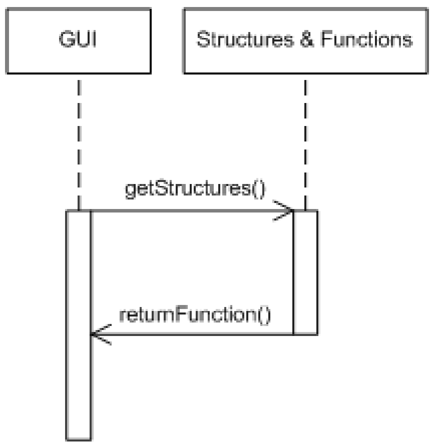
If you run a function, then a new animation visualizing it will be created which is then displayed. This animation will be paused though, so we will remain in the same ”Animation Paused” state. From the animation state machine's point of view, creating a structure is no different from running a function. It will receive an animation to play, which is paused.

Note that the animation will only pause between steps. With steps we mean that an object has appeared/disappeared completely, an arrow has reached is new destination, etc. It is not possible to pause the animation when it is half-through an action or step. Therefore, the animation state machine has a ”Run until next step” which is run automatically until a complete step has been finished. After that, it goes to the paused state.

Interaction diagram

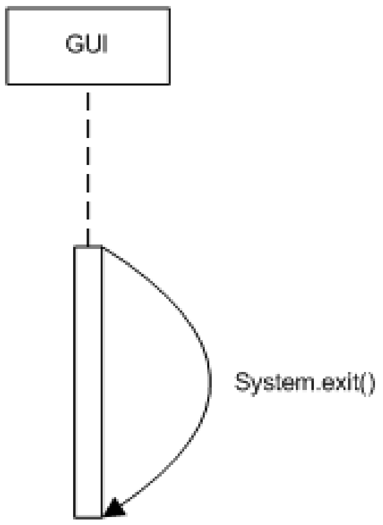
Starting the program

This diagram represents how the modules communicate during the start of the program.



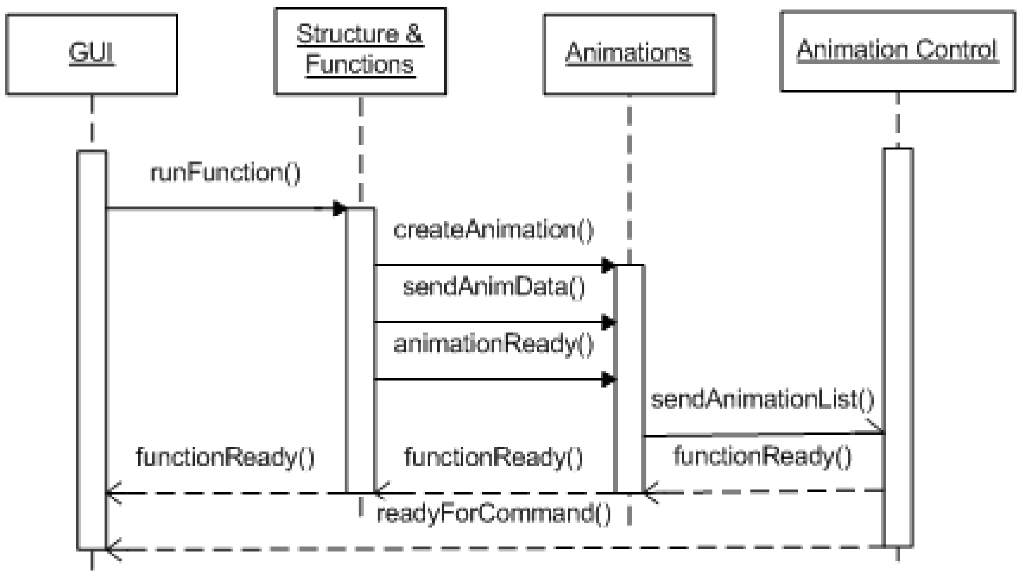
Quitting the program

This diagram represents how the class closes the program.



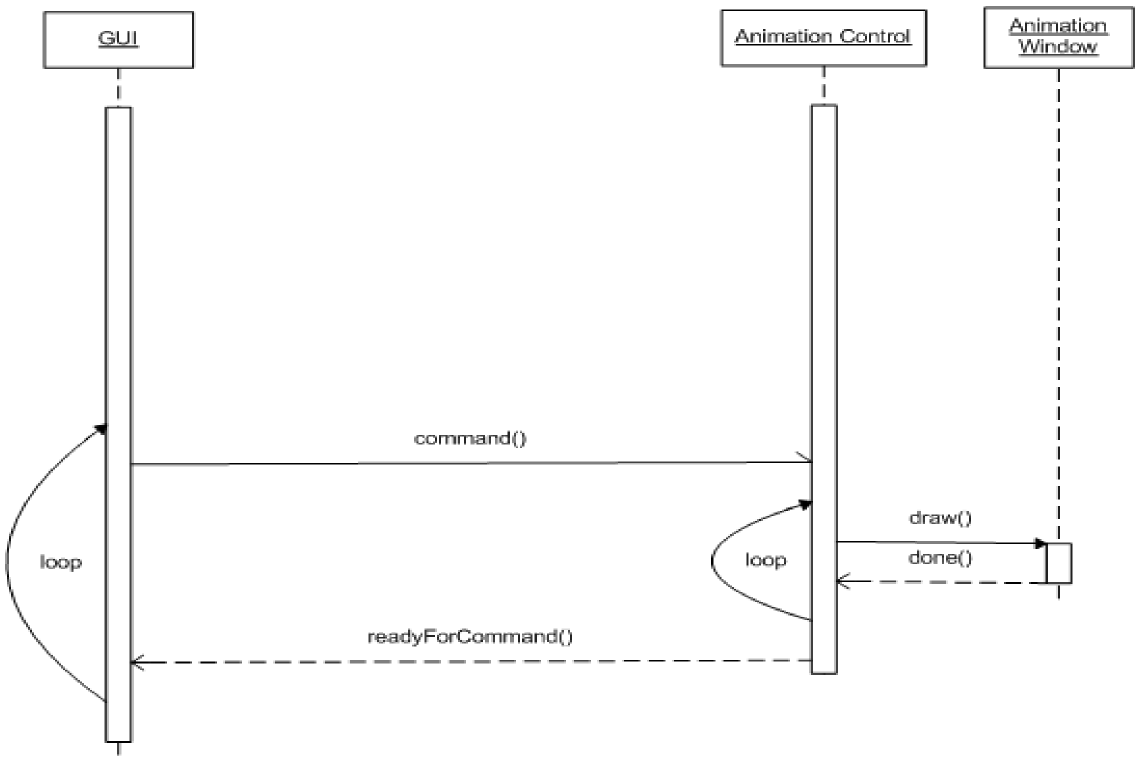
Run a function

This is how the modules communicate when the user runs a function.



Run an animation

This is how the animation control module communicates with the GUI and Animation Window when showing an animation.



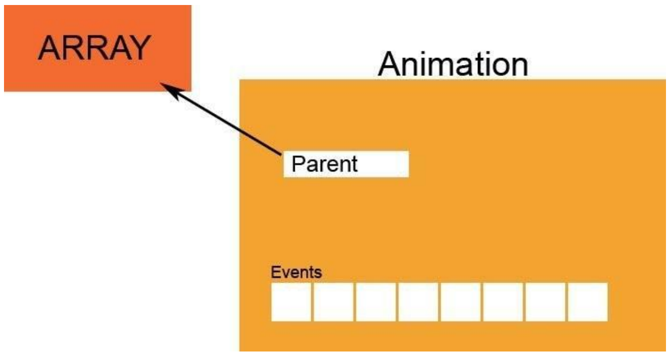
Detailed Design

Data structures

We have two internal data structures in our system, they are named: Animation, and Frame.

Animation

This data structure holds a list of animation steps as well as what type of data structure it represents. This structure also has a pointer pointing back at the structure it was created from. The available animations steps are:



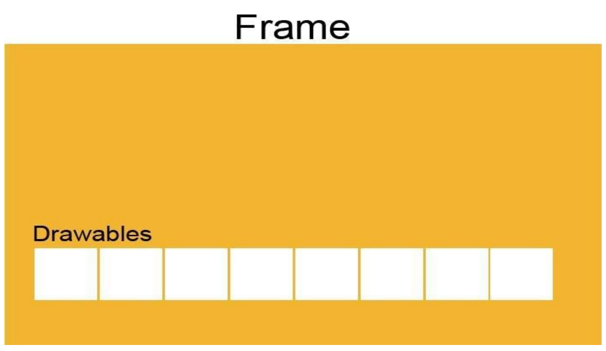
- CreateNode
- DeleteNode
- SwapNode
- CreatePointer
- DeletePointer
- SwapPointer
- MoveNode

It is created in the Structure class and thereafter used in the AnimationController class.

Every node keeps track of at least three things: id, value and type. Some might have to keep track of its parent.

Frame

The frame structure represents a picture of the animation in a specific condition. It contains a list of Drawable objects that knows how they should be drawn to the screen. This Drawable object contains all the necessary information to draw the object, such as coordinates, object type, etc. Examples of Drawable objects are:



4. Pointers
5. ArrayNodes
6. LinkedListNodes
7. SortedBinaryTreeNodes
8. TemporaryNodes
9. Header

This structure is used by the Animation window when it draws the animation on the screen. It is created by the FrameCreator class which sends a certain amount of Frames per second.

Methods

This list of methods contains all major methods. Various GUI methods, smaller helper methods and private methods are not included.

Class: MainWindow

Method: Int main(String[] args)

Program starting point. It sets up all the GUI classes, creates an Animation Controller and a DataStructures object.
Parameters: Not used
Return value: Zero for successful execution, otherwise non-zero.
Pre-conditions: Nothing is set up at all.
Post-Conditions: The program is up and running successfully or a non-zero integer is returned from the function.
Called by: Java Virtual Machine
Calls: DataStructures.getAvailableStructures(),DataStructures.getAvailableFunctions()

Class: DataStructures

Method: Void createStructure(String structure)

Creates a structure. It replaces the old structure if necessary.
Parameters: String structure tells the method which structure to create.
Return value: none
Pre-conditions: none
Post-conditions: a data structure is loaded and filled with numbers, if a old one existed, it was replaced.
Called by: Java Swing GUI callbacks
Calls: Structure's constructor.

Method: Animation runFunction(String function)

This method is just a wrapper for the equivalent method in the current structure.
Parameters: A string specifying the desired function.
Return value: An animation describing the method execution, or null if an error occurred.
Pre-conditions: There must be a current structure loaded.
Post-conditions: See the structure method.
Called by: Java Swing GUI callbacks.
Calls: Equivalent structure method.

Method: String[] getAvailableFunctions(void)

This method is just a wrapper for the equivalent method in the current structure.
Parameters: none
Return value: A list of the available functions in String form, nothing is return if a structure is not loaded.
Pre-conditions: none
Post-conditions: The list of strings is returned.

Called by: Java Swing GUI callbacks and the main method
Calls: Structures.getAvailableFunctions()

Method: String[] getAvailableStructures(void)

A complete set of available Structures is returned.
Parameters: none
Return value: A list of the available structures in String form.
Pre-conditions: none
Post-conditions: The list of strings is returned.
Called by: Java Swing GUI callbacks and the main method
Calls: none

Class: Structure

Method: String[] getAvailableFunctions(void)

Depending on what structure is currently loaded, a complete set of available functions is returned.
Parameters: none
Return value: A list of the available functions in String form, nothing is return if a structure is not loaded.
Pre-conditions: none
Post-conditions: The list of strings is returned.
Called by: DataStructures.getAvailableFunctions()
Calls: none

Method: Animation runFunction(String function)

This method executes a function and returns an animation describing the function.
Parameters: A string specifying the desired function.
Return value: An animation describing the method execution, or null if an error occurred.
Pre-conditions: The function specified by the string must be available.
Post-conditions: Same as before. The structure isn't actually changed before the animation is drawn.
Called by: DataStructures.runFunction()
Calls: none

Class: AnimationController

Method: Void tick()

This method tells the FrameCreator what is changed during a step in the animation.
Parameters: none
Return value: none
Pre-conditions: That an animation is running.
Post-conditions: The animation step has been sent to the FrameCreator.
Called by: Java timer object
Calls: FrameCreator.createFrame(Animation,int)

Method: Void updateGUI()

This method tells the MainWindow what buttons the user can press.
Pre-conditions: none

Post-conditions: The appropriate AnimationControll buttons gets updated.
Called by: Java timer object
Calls: none

Class: FrameCreator

Method: Void createFrame(Animation,int)

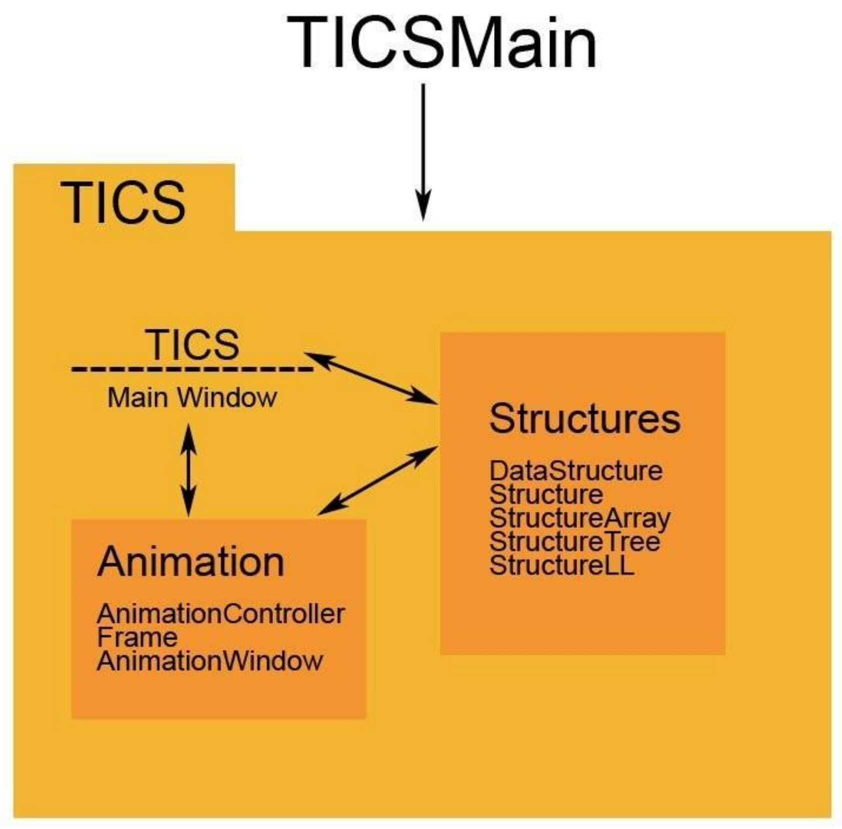
This method takes an Animation and an int and creates a number of Frames which it sends to the AnimationWindow.
Parameters: Animation and an integer.
Return value: none
Pre-conditions:none
Post-conditions: The Frames are sent to the AnimationWindow.
Called by: AnimationController.tick()
Calls: AnimationWindow.draw()

Class: AnimationWindow

Method: void Draw(Frame arg)

Draws a frame in the animation area.
Parameters: a frame with all objects' positions.
Return value: none.
Pre-conditions: none.
Post-conditions: the frame has been drawn.
Called by: FrameCreator.createFrame()
Calls: none

Package Diagram



Functional Test Cases

Create data structure

These tests correspond to the Data structure session in our RD which can be found on page 9.

Create a Data Structure 1

Description: This test case is to assure that creating a data structure works correctly when no other data structure has been created.
Preconditions: No data structure has been created since the program started.
Expected outputs: The specified data structure is created and visible in the drawing area.
Step by step procedure: Choose a data structure from the menu. Restart the program. Repeat this for all data structures.

Create a Data Structure 2

Description: This test case is to assure that creating a data structure works correctly when another data structure has been created and no functions have run.
Preconditions: A data structure has already been created and no functions have run since the program started.
Expected outputs: The specified data structure is created and visible in the drawing area.
Step by step procedure: Choose a data structure from the menu. Repeat this for all data structures.

Create a Data Structure 3

Description: This test case is to assure that creating a data structure works correctly when another data structure has been created and a function has run.
Preconditions: A data structure has already been created and a function has run since the program started.
Expected outputs: When an animation is running the “run function” option is disabled. When the animation is paused the “run function” option is enabled.
Step by step procedure: Choose a data structure from the menu. Repeat this for all data structures.

Run a function

These tests correspond to the Available functions session in our RD which can be found on page 9.

Run a function 1

Description: This test case is to check that running a function works as described in our RD. A function should not have run before on that data structure.
Preconditions: A data structure has already been created and no function has run.
Expected outputs: A function is run as it should.
Step by step procedure: Select a function and specify any eventual parameters.

Run a function 2

Description: This test case is to check that running a function, which takes no parameter, on a data structure works as described in our RD. A function should have run before on that data structure.
Preconditions: A data structure has already been created and a function has run on that data structure.
Expected outputs: A function is run as it should.
Step by step procedure: Select a function and specify any eventual parameters.

Add first

These tests correspond to the Add first function in our RD which can be found on page 10.

Add first 1

Description: This test case is created to assure that the Add first function is working as described in our RD when the data structure is empty.
Run on: Linked list and array.
Preconditions: A data structure has been created and the data structure is empty.
Inputs: An integer.
Expected outputs: The parameter value is inserted first.
Step by step procedure: Select the AddFirst function. Select a correct parameter.

Add first 2

Description: This test case is created to assure that the Add first function is working as described in our RD when the data structure is full.
Run on: Linked list and array.
Preconditions: A data structure has been created and the data structure has the maximum amount of elements.
Inputs: An integer.
Expected outputs: An error message notifies the user that the structure is full. Nothing in the structure is changed.
Step by step procedure: Select the AddFirst function.

Add first 3

Description: This test case is created to assure that the Add first function is working as described in our RD when the data structure has some elements but is not full.
Run on: Linked list and array.
Preconditions: A data structure has been created and the data structure has some elements but is not full.
Inputs: An integer.
Expected outputs: The parameter value is inserted first.
Step by step procedure: Select the AddFirst function. Select a correct parameter.

Add first 4

Description: This test case is created to assure that the Add first function is working as described in our RD when bad parameters are inserted.
Run on: Linked list and array.
Preconditions: A data structure has been created.
Inputs: Characters, floats or integers not in our range.
Expected outputs: An error message notifies the user. Nothing in the structure is changed.
Step by step procedure: Select the AddFirst function. Select an incorrect parameter.

Insert

These tests correspond to the Insert function in our RD which can be found on page 10.

Insert 1

Description: This test case assures that the Insert function works as specified in our RD when the data structure is empty.
Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created and is empty.
Inputs: Two integers.
Expected outputs: The parameter value is inserted at the specified index if singly linked list or array. Insert in sorted binary tree will place the parameter depending on the parameter value.
Step by step procedure: Select the Insert function. Select two correct parameters**.**

Insert 2

Description: This test case assures that the Insert function works as specified in our RD when the data structure is full.
Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created and is full.
Inputs: Two integers.
Expected outputs: An error message notifies the user. Nothing in the structure is changed.
Step by step procedure: Select the Insert function.

Insert 3

Description: This test case assures that the Insert function works as specified in our RD when the data structure non-empty and non-full.
Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created and has elements in it but is not full.
Inputs: Two integers.
Expected outputs: The parameter value is inserted at the specified index if singly linked list or array. Insert in sorted binary tree will place the parameter depending on the parameter value.
Step by step procedure: Select the Insert function. Select two correct parameters.

Insert 4

Description: This test case assures that the Insert function works as specified in our RD when the bad parameters are used.

Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created.
Inputs: Characters, floats or integers not in our range.
Expected outputs: An error message notifies the user. Nothing in the structure is changed.
Step by step procedure: Select the Insert function. Use an incorrect value parameter. Select the Insert function. Use an incorrect index parameter.

Search

These tests correspond to the Search function in our RD which can be found on page 10

Search 1

Description: This test case assures that the Search function works as specified in our RD when the data structure is empty.
Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created and is empty.
Inputs: An integer.
Expected outputs structure.
Step by step procedure: Select search function**.**

Search 2

Description: This test case assures that the Search function works as specified in our RD when the element searched for does not exist.
Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created.
Inputs: An integer.
Expected outputs: Will go through the search algorithm and when it's done a message will notify the user that there are no element in the data structure that match your input.
Step by step procedure: Select search function. Select a parameter that does not exists in the data structure.

Search 3

Description: This test case assures that the Search function works as specified in our RD when the element searched for does exist.
Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created and has some elements.
Inputs: An integer.
Expected outputs: Will go through the search algorithm and when it finds the element you searched for a message will notify the user that it found an element that match the input parameter.
Step by step procedure: Select search function. Select a parameter that do exist in the data structure.

Search 4

Description: This test case assures that the Search function works as specified in our RD when the parameters are invalid.
Run on: Linked list, array and binary tree.
Preconditions: A data structure has been created.
Inputs: Characters, floats or integers not in our range.
Expected outputs: An error message notifies the user. Nothing in the structure is changed.
Step by step procedure: Select search function. Select an incorrect parameter.

Set random values

These tests correspond to the Set random values function in our RD which can be found on page 10

Set random values 1

Description: This test case is to check that the function set random values doesn't accept parameters that aren’t allowed**.**
Run on: Linked list, array and binary tree.
Precondition: A data structure has been loaded.
Inputs: Characters, floats or integers not in our range.
Expected outputs: An error message notifies the user. Nothing in the structure is changed.
Step by step procedure: Select set Random values function. Use incorrect parameter.

Set random values 2

Description: This test case is to check that the function will change the elements correct when you have a correctly input parameter.
Run on: Linked list, array and binary tree.
Precondition: A data structure has been loaded.
Inputs: An integer.
Expected outputs: All elements will be removed and the number of the input random elements will be created.
Step by step procedure: Select set Random values function. Use correct parameter.

Remove

These tests correspond to the Remove function in our RD which can be found on page 11.

Remove 1

Description: This test case is to check that the function remove doesn't accept parameters that aren’t allowed.
Run on: Linked list, array and binary tree.
Precondition: A data structure has been loaded.
Inputs: Characters, floats or integers not in our range.
Expected outputs: An error message notifies the user. Nothing in the structure is changed.
Step by step procedure: Select the remove function. Use incorrect parameter.

Remove 2

Description: This test case is to check that the function remove still works after you try to remove an element that doesn't exist.
Run on: Linked list, array and binary tree.
Precondition: A data structure has been loaded and has some elements.
Inputs: An integer.
Expected outputs: A message notifies the user. The parameter does not exist.
Step by step procedure: Select the remove function. Use correct parameter.

Remove 3

Description: This test case is to check that the function remove, removes the element after a correctly input parameter.
Run on: Linked list, array and binary tree.
Precondition: A data structure has been loaded, has some elements and has at least one element with the input value.
Inputs: An integer.
Expected outputs: Will go through the remove algorithm and when it finds the element you wanted to remove, it removes the element.
Step by step procedure: Select the remove function. Use correct parameter.

Remove by index

These tests correspond to the Remove by index function in our RD which can be found on page 11.

Remove by index 1

Description: This test case is to check that the function remove by index doesn't accept parameters that aren’t allowed.
Run on: Linked list and array.
Precondition: A data structure has been loaded.
Inputs: Characters, floats or integers not in our range.
Expected outputs: An error message notifies the user. Nothing in the structure is changed.
Step by step procedure: Select the remove by index function. Use incorrect parameter.

Remove by index 2

Description: This test case is to check that the function remove, removes the element after a correctly input index parameter.
Run on: Linked list and array.
Precondition: A data structure has been loaded and has at least one element with the input value.
Inputs: An integer.
Expected outputs: The value specified by the index parameter is removed.
Step by step procedure: Select the remove by index function. Use correct parameter**.**

Sort

These tests correspond to the Sort function in our RD which can be found on page 12.

Sort 1

Description: This test case assures that each of the sorting algorithms work as specified in the RD when the data structure is empty.
Run on: Array.
Precondition: A data structure has been loaded and is empty.
Expected outputs: Nothing happens.
Step by step procedure: Select Insertion sort function. Select Merge sort function. Select Quick sort function.

Sort 2

Description: This test case assures that each of the sorting algorithms work as specified in the RD when the data structure has only one element.
Run on: Array.
Precondition: A data structure has been loaded and contains one element.
Expected outputs: Nothing happens.
Step by step procedure: Select Insertion sort function. Select Merge sort function. Select Quick sort function.

Sort 3

Description: This test case assures that each of the sorting algorithms work as specified in the RD when the data structure has several randomly generated numbers.
Run on: Array.
Precondition: A data structure has been loaded and has some randomly generated elements.
Expected outputs: After the sorting is run, the structure is sorted.
Step by step procedure: Select Insertion sort function. Fill structure with random values. Select Merge sort function. Fill structure with random values. Select Quick sort function.

Sort 4

Description: This test case assures that each of the sorting algorithms work as specified in the RD when the data structure is sorted.
Run on: Array.
Precondition: A data structure has been loaded and has sorted elements.
Expected outputs: Nothing happens.
Step by step procedure: Select Insertion sort function. Select Merge sort function. Select Quick sort function.

Sort 5

Description: This test case assures that each of the sorting algorithms work as specified in the RD when the data structure is sorted in reverse order.
Run on: Array.
Precondition: A data structure has been loaded and has elements sorted in reversed order.
Expected outputs: After the sorting is run, the structure is sorted.

Step by step procedure: Select Insertion sort function. Fill structure with random values. Select Merge sort function. Fill structure with random values. Select Quick sort function.

Sort 6

Description: This test case assures that each of the sorting algorithms work as specified in the RD when the data structure has a number of identical elements.
Run on: Array.
Precondition: A data structure has been loaded and has elements with the same value.
Expected outputs: Nothing happens.
Step by step procedure: Select Insertion sort function. Select Merge sort function. Select Quick sort function.

Animation

These tests correspond to the Animation part of our RD which can be found on pages 12 to 14.

Press a button

Description: This test case assures that none of the Animation control buttons can be pressed when neither a structure is not loaded nor a function has been run.
Input: The user presses a button.
Precondition: A function has not been run.
Expected outputs: Nothing happens.
Step by step procedure: Try to press an animation control button.

Play

Description: This test case assures that the Play button only works when a structure is loaded, a function has been run and the animation is in a paused state.
Precondition: A data structure has been loaded, a function has been run and the animation is in paused state.
Expected outputs: The animation is running.
Step by step procedure: Press Play.

Pause

Description: This test case assures that the Pause button only works when a structure is loaded, a function has been run and the animation is in a running state.
Preconditions: A data structure has been loaded, a function has been run and the animation is in running state.
Expected outputs: The animation is paused.
Step by step procedure: Press Pause.

Step forward

Description: This test case assures that the StepForward button only works when a structure is loaded, a function has been run and the animation is in a paused state.
Preconditions: A data structure has been loaded, a function has been run and the animation is in

paused state.
Expected outputs: The animation starts to play and pauses when it finishes a step.
Step by step procedure: Press StepForwards.

Step backwards

Description: This test case assures that the StepBackwards button only works when a structure is loaded, a function has been run and the animation is in a paused state.
Preconditions: A data structure has been loaded, a function has been run and the animation is in paused state.
Expected outputs: The animation starts to play backwards and pauses when it finishes a step.
Step by step procedure: Press StepBackwards**.**

Stop

Description: This test case assures that the Stop button only works when a structure is loaded, a function has been run and the animation is in a paused state.
Preconditions: A data structure has been loaded, a function has been run and the animation is in paused state.
Expected outputs: The animation returns to the beginning immediately.
Step by step procedure: Press StepFirst.

Finish

Description: This test case assures that the Finish button only works when a structure is loaded, a function has been run and the animation is in a paused state.
Preconditions: A data structure has been loaded, a function has been run and the animation is in paused state.
Expected outputs: The animation reaches the end immediately.
Step by step procedure: Press StepLast.

Index