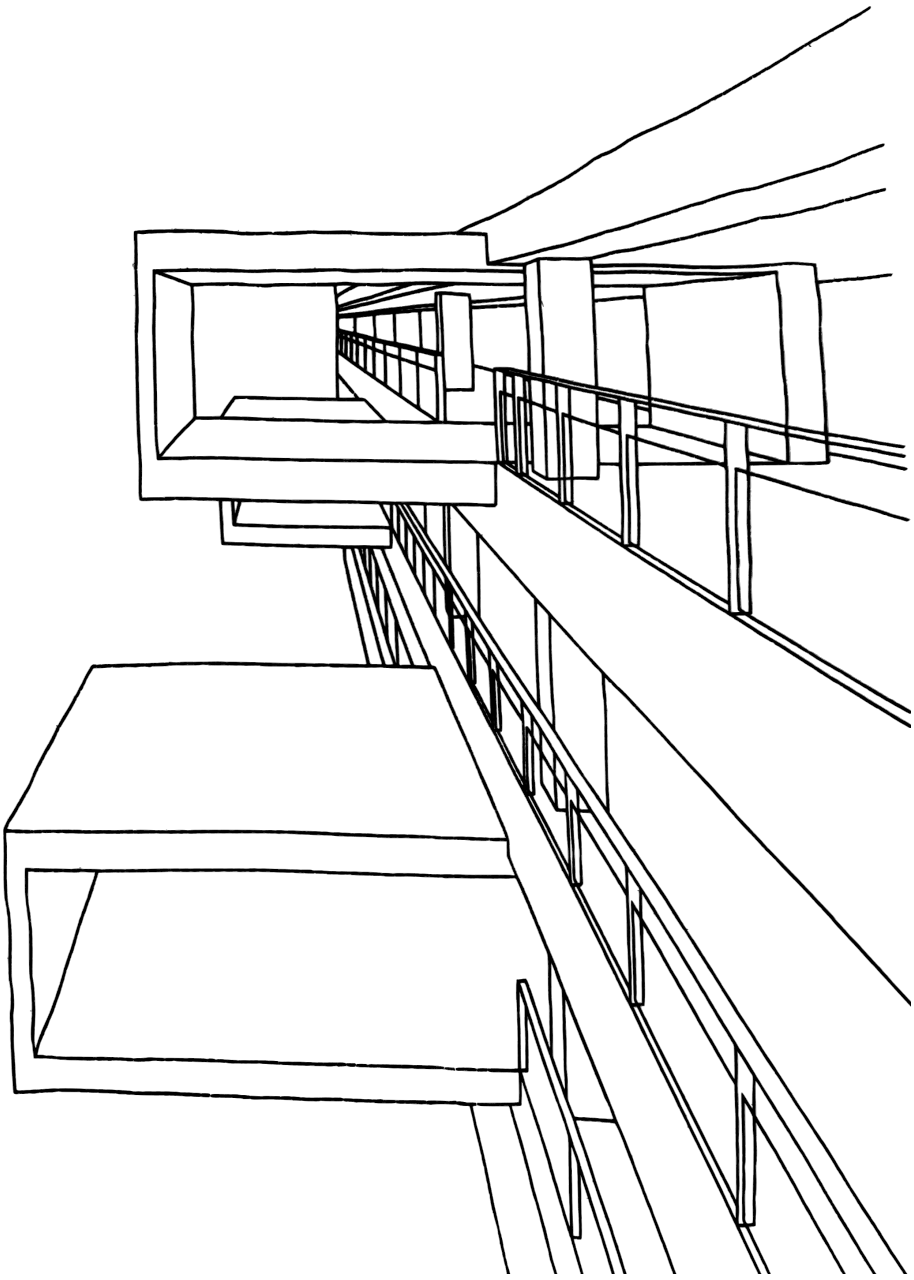
IT UNIVERSITY OF COPENHAGEN

Master Thesis

Abstractions and Visualizations, Enhancing Basic Data Structure Understanding

MSc Software Design
IT University of Copenhagen
Johannes Rolf Guthey Schwartzkopff (rosc@itu.dk),
Nicholas Gioachini (ngio@itu.dk)

Supervisor: Thore Husfeldt
Activity code: KISPECI1SE
Submission Date: 02 January 2024



1 Abstract

Data structures constitute fundamental principles in computer science, but they can be challenging to learn and teach. This thesis investigates how visualizations can help students and educators grasp these notions more effectively, by showing different aspects and levels of abstraction of the same concept.
To achieve this, the present project develops a theoretical framework that combines educational theories from computer science, philosophy, and mathematics. This framework supports the utilization of multiple representations and layers of abstraction for understanding abstract concepts.
The developed framework is used to create a software product based on the Manim Python library. Such product is able to generate dynamic animations and visualisations of data structures, allowing users to explore these structures at different levels of details and abstraction. Examples include: Resizing Arrays and Hash Tables.
A final evaluation and comparison with other existing visualisation tools and resources shows our product's usefulness as an educational resource.

Keywords: *Data Structures, Visualisations, Abstraction, Layers of Abstraction, Manim.*

Codebase for developed library: `https://github.itu.dk/rosc/Manim_RP`

Contents

2 Introduction

In the realm of Computer Science education, visualisations play a crucial role in clarifying complex concepts. This thesis explores the intricate interplay between visualizations of data structures and the added value they may provide to both students studying Algorithms, and teachers exploring new tools to facilitate the subject's comprehension.

It is argued that depicting visually data structures when learning them, may enhances their understanding and prevent misconceptions. Drawing from theoretical foundations laid by Hermans, Floridi, and Duval[23][18][16], this thesis synthesizes a theoretical framework advocating for the representation of concepts at different, connected, layers of abstraction, to assist in the discernment of the entirety of different data structure concepts. This theoretical synthesis serves as our starting point for developing a software product which implements its findings.

Inspired by the open-source Python framework for mathematical animations, *Manim*[12], we endeavour to develop a custom library which applies our theoretical framework, for the purpose of crafting animations and visual representation of data structures at different layers of abstractions.
In the scope of this project, we focus on investigating two specific data structures:

- Resizing Arrays
- Hash Tables

The motivation behind this thesis is to contribute to the didactic development in the educational environment of computer science. Our objective is to enhance the already present illustrations and visualisations present in Algorithms' textbooks through a custom library that can produce dynamic animations as an additional educational resource.

The paper commences with an exploration of key concepts including the chosen data structures (resizing arrays and hash tables), past visualizations of these, and an examination of Manim. Following this is an in-depth theoretical analysis covering mental models, abstraction, and similar visualisation tools.
The methodology section details our ways of working, encompassing our approach to existing Software Engineering theories. This leads to the presentation and discussion of results, which showcase the developed product and detailed animations created with it.
The final discussion covers the effects of visualization within didactics, the effectiveness of Manim, a comparative analysis of our work with an established educational resource, and potential future work, concluding with an acknowledgment of contributions and limitations.

We now outline our Research Questions, which range from broad to narrow, covering four levels of specificity. These questions serve as a guide in this project, shedding light on whether visualisations can aid the understanding of data structure concepts.

Research Questions[1]

1. Are visualizations of data structures in teaching material a tool used to promote their understanding?
2. Can visualizations of data structures lead to misconceptions regarding their functioning?
3. Does a combination of visual representations help prevent such misconceptions?
4. Can a synthesis of existing theory guide the creation of a software product facilitating the combination of visual representations?

[1]Research Questions are mentioned in the text interchangeably as both 'Research Question' and 'RQ', followed by their related number.

3 Background

The background section serves as an introduction to the foundational data structures of resizing arrays and hash tables, chosen for the scope of this thesis. It provides an in-depth exploration of their operational intricacies and visualization history. The background also introduces Manim, an open-source visualization library, and explains how it may be used to create visualizations.

3.1 Resizing arrays in short

Resizing arrays are fundamental data structures used extensively in computer science and software engineering to manage collections of data of varying sizes efficiently[41]. They offer a dynamic and flexible approach to storage allocation, enabling applications to adapt to changing requirements without sacrificing performance. This foundational concept lies at the core of many critical algorithms and data structures, making it an essential topic of study in the realm of computer science.

At its core, a resizing array is an ordered collection of elements with the ability to dynamically adjust its size during runtime. Unlike static arrays, which have a fixed size upon creation, resizing arrays allow for automatic expansion or contraction as elements are added or removed. This dynamic behavior makes them versatile and adaptable, as they can efficiently manage both small and large datasets without excessive memory allocation or wastage. When an array reaches its capacity, a new, larger block of memory is allocated, and the existing elements are copied over. Conversely, when elements are removed, the array may shrink to prevent unnecessary memory consumption. The functioning of this resizing is designed to provide amortized constant-time complexity for key operations like appending and removing elements. Although resizing itself takes linear time, it occurs infrequently, so the amortized time complexity remains constant.

3.1.1 Computational costs

The following subsection describes the computational costs incurred when using resizing arrays, as well as the amortized analysis of these costs.

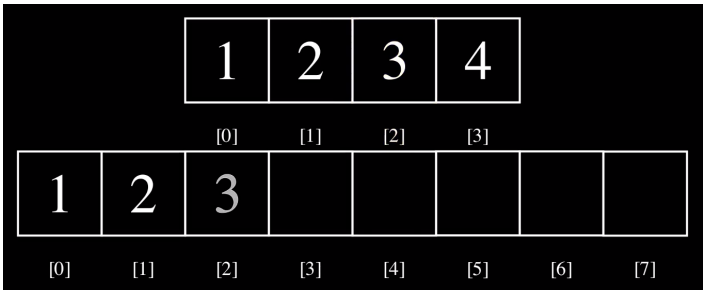


Figure 1: Resizing array during increasing resizing

In Figure 1 we see a visualization of a resizing array performing a resizing, which has occurred due to an additional value, '5', being appended. The upper array, which has reached capacity, cannot hold the value, necessitating the resizing. To describe computational cost, we denote the amount of elements in the resizing array by n.

As the '5' has not yet been appended, $n = 4$ in Figure 1. The new, lower array, is created at twice the size of the original, resulting in a computational cost of $2n$, as the cost of a single array slot creation is 1. The values of the smaller array are then copied to the new array by reading and subsequently writing them. Individual reads and writes have a cost of 1, resulting in the total copying cost being $2n$. Finally the new value is added to the new array, adding 1 to the cost. As such the total cost of appending a value when the array is full is $4n + 1$.

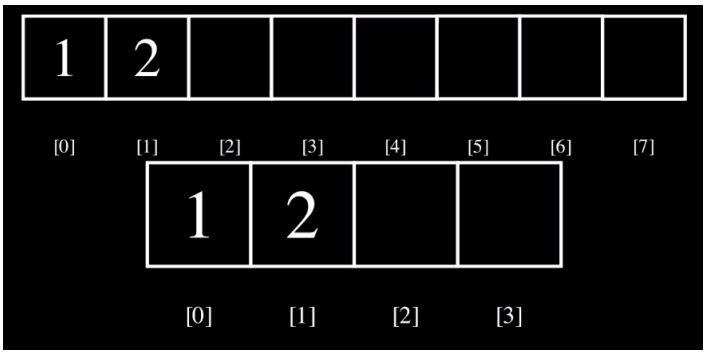


Figure 2: Resizing array during decreasing resizing

Conversely, Figure 2 shows a resizing array which is shrinking to avoid unnecessary memory use. The upper array holds too few elements when a value is removed, causing the creation of a new, smaller array. At the point of resizing, the upper array holds $\frac{1}{4}$ of its total capacity. The new array, half the size of the original, will then be of size $\frac{n*4}{2} \rightarrow 2n$, resulting in a computational cost of $2n$. Copying the elements into the new array similarly costs $2n$ operations, with one additional operation to remove the last element, resulting in a total cost of $4n + 1$, just as that of an increasing resizing.

The act of shrinking a resizing array holds two implementation-specific factors which affect the computational cost analysis. Firstly, at what point of array capacity a shrink can be set to occur. In the case of this project, shrinking occurs when $n = \frac{1}{4}$ of array capacity. Secondly, the choice of whether the last element is copied over and then deleted or simply not copied over, resulting respectively in a cost of 3 or a saving of 2 computational operations. This project uses the former, so the same copying method can be used for growing and shrinking arrays, and resizings always cost $4n + 1$. This results in the action costs shown in Figure 3.

	W/o resizing	W/ resizing
Add()/Remove()	1	4n+1

Figure 3: Computational costs for Resizing Arrays.

Amortized analysis is used in assessing the efficiency of dynamic data structures like resizing arrays, particularly when their operations incur occasional costly processes. In the case of resizing arrays, where the typical cost of adding or removing an element is 1, but occasionally requires a more resource-intensive operation with a cost of $4n + 1$, amortized analysis provides a balanced perspective. It allows us to distribute the expensive cost over multiple operations to provide an average cost that remains close

to 1 per operation. This smoothing out of costs, achieved by redistributing the extra overhead incurred during resizing, ensures that the resizing array maintains its overall efficiency. This makes it a practical and robust choice for managing dynamic data, even in the face of rare, expensive operations. The amortized analysis is a valuable tool for understanding the long-term performance characteristics of resizing arrays.

3.2 Hash Tables explained

Hash tables are likewise fundamental data structures used to efficiently store and retrieve key-value pairs. Due to their ability to provide constant-time average-case access and retrieval[41], they play a crucial role in various applications, such as databases, caches, and symbol tables. We here explain their basic structure and functioning, and create an understanding for their broader application in computer science.

A hash table is a data structure that utilizes a hash function to map keys to indices within an array, often referred to as the "hash array" or "buckets". Each index in the array is associated with a linked list or an array of key-value pairs, and the key is used to determine the index where the corresponding value will be stored. The structure of a hash table can be summarized as follows:

1. *Array*: The core data structure of a hash table is an array. This array typically contains a fixed number of slots or buckets, where each slot can hold one or more key-value pairs. The size of the array is chosen based on the expected number of key-value pairs to be stored in the hash table. It is important to choose an appropriate array size to balance the trade-off between memory usage and the likelihood of collisions.
2. *Hash Function*: The hash function is a critical component of a hash table. It takes a key as input and returns an index (often an integer) within the range of the array size. The primary goal of the hash function is to distribute keys uniformly across the available slots in the array, minimizing collisions. A well-designed hash function ensures that different keys are mapped to different indices, reducing the chances of multiple keys hashing to the same slot.
3. *Collision Resolution*: Collisions occur when two or more keys map to the same index in the array due to the finite range of the hash function. Hash tables employ various techniques for handling collisions. Common methods include separate chaining (see Figure 4), where each slot contains a linked list of key-value pairs, or open addressing, where the algorithm searches for the next available slot in the array. Each method has its advantages and disadvantages, and the choice depends on the specific use case and design considerations.

Through the use of the hash function, a hash table inserts, retrieves and deletes objects efficiently, as inputting a given value in the hash function yields an index which, dependent on the chosen collision resolution, is or is close to the index where the value belongs. Taking separate chaining as an example, the cost of accessing an element is at most the cost of hash function calculation (constant), and the length of the chain at the relevant index. Such chains are often limited in size, and the occurrence of an insert into a at-limit chain will trigger a resizing of the hash table. Similar to the resizing array, a new, larger array is created, the hash function recalibrated, and the held values copied into a new, appropriate index. Due to the many different ways of implementing a hash table, this thesis will not go further into calculating computational costs, however it is suffice to say that such resizings are rarely occurring expenses, and amortized analysis

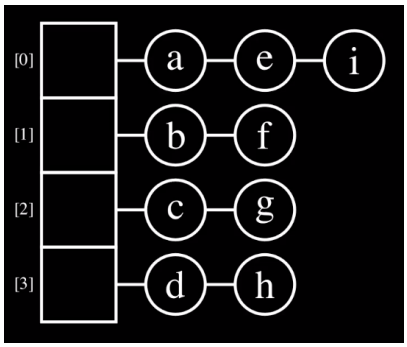


Figure 4: A hash table using separate chaining.

for hash table implementations is possible and has been done[29].

3.3 Past visualizations

As this thesis examines the benefit of viewing data structures through different visual representations, and the related software product creates such visualizations, we here briefly examine how arrays and hash tables have been visually represented in other works.

3.3.1 Array Visualization According to The Von Neumann Model

The Von Neumann Model, developed by John von Neumann, has significantly shaped how we visualize data and data structures, and we here explore how the visualization of arrays typically occurs within the framework of this model. We show how the model's representation of data, its memory addressing scheme, and its array-based structure serve as the primary reference point for comprehending the spatial and operational aspects of array-based data structures.

Within the Von Neumann Model, the memory addressing system shapes array visualization. It treats memory as a linear, uniformly addressable space, with each block of memory assigned a unique address. Each block is instantly accessible, and may hold data of a certain size[20]. This leads to a simple visualization of arrays, where elements in array visualizations are organized linearly, typically as boxes with content and indices. The Von Neumann Model based representation of arrays thus mirrors the contiguous nature of memory and facilitates a direct mapping of arrays to memory addresses.

Figures 1 and 2 in the previous Section and Figure 5 use the Von Neumann based way of visualizing arrays, and examining popular Computer Science Textbooks such as *Algorithms*[41] and *Introduction to Algorithms*[14], we see how a Von Neumann based array visualization approach is utilized in some of the most popular data structure teaching materials. Indeed we make the claim that this is the ubiquitous way of visualizing arrays.

Such visualization aids in grasping array operations, offering a visual framework for illustrating behaviors during common operations like insertion, deletion, and traversal. This representation provides a clear depiction of how array elements are shifted

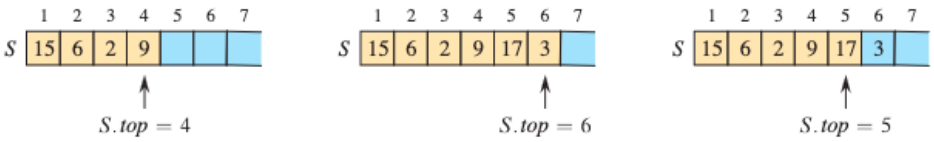


Figure 5: Visualization of a Stack, an array-based data structure, in *Introduction to Algorithms*[14].

or modified in memory, offering valuable insights into the dynamic nature of arrays. However, this Von Neumann-based array visualization offers only one perspective for viewing array-based data structures, which as we shall see is not necessarily sufficient for a full understanding of a given data structure.

3.3.2 Past Hash Table Visualizations

As previously discussed in Section 3.2, Hash Tables can be comprised of an array, where each of its slot contains a reference to a linked list. The linked list is comprised of a series of nodes, where each one of them links to the following node.
Each node holds two elements, data and a reference pointer to the following node in the sequence. To indicate its start, a linked list holds a reference to the first node in its sequence of nodes, commonly known as the "head". The last node in a linked list has a null reference pointer, which indicates the end of the list. Similarly to resizing arrays, the linked list has the dynamic capability of adjusting its size by expanding or shrinking according to the capacity required. Unlike arrays, a linked list connects their nodes through pointers, which allows smooth insertion, deletion and traversal.

In their *Algorithms* book Sedgewick & Wayne [41] illustrate a linked list, as depicted in Figure 6 below.

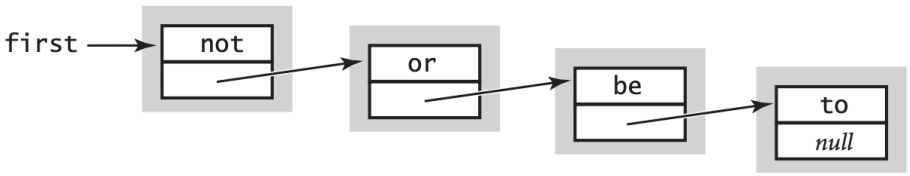


Figure 6: Linked list in *Algorithms*[41]

Their visualisation makes use of boxes and arrows, where the first represents a node and the latter a pointer to the following node. Here, the head of the list is denoted as "first". The depiction aids the understanding of the linked list's adaptability through its operations. When inserting new nodes or deleting existing ones, the list re-positions the pointers without moving slots in memory. A traversal operation explores nodes through their pointers, from head to the last node in the sequence. The nature of

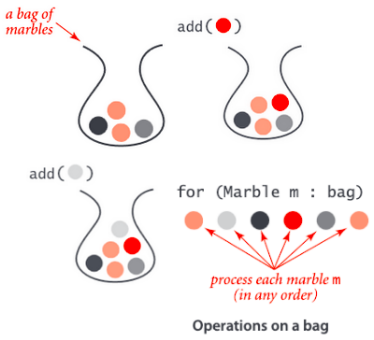


Figure 7: Drawing of a Bag [41]

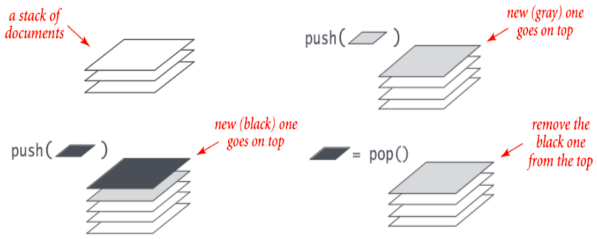


Figure 8: Drawing of a Stack [41]

linked list makes it a fast and versatile data structure, which adapts quickly to varying data sizes especially when accessing or modifying data sequentially.

The combination of the Von Neumann model-based representation for arrays with a linked list's representation, make up our visualisation of a hash table with separate chaining, as presented in Figure 4.

3.4 Visualizations in Teaching Materials

As hinted at by our previous mentions of them, an examination of two widely used data structure textbooks, *Algorithms*[41] and *Introduction to Algorithms*[14] with a focus on how these materials facilitate the teaching of data structures, reveals how these books make use of visualizations to convey information.

Algorithms introduces data structures through their API, providing an overview of their functioning.
Visual representations, including drawings and traces, are included in the textbook to allow for intuitive understanding of the concepts. Figures 7 and 8 show how *Algorithms* displays Stacks and Bags at a high abstraction level. The book also offers full class implementations for comprehensive student understanding.

Introduction to Algorithms covers a range of data structures, including Stacks, Queues. Descriptions are primarily textual, relating data structures to real-life concepts. While sparse in graphical representations, the book includes Von Neumann-based array visualizations, such as those seen in Figure 5.

Both textbooks use visualization to varying degrees, with *Algorithms* excelling in using a range of different visual models for the data structures it displays. The extensive use of visualizations in the two widely used textbooks shows us that visualizations are an important tool for understanding data structures, thereby answering RQ1.

3.5 An Overview of Manim

As mentioned in the introduction, the software product accompanying this thesis is developed using the open-source *Manim* library [12]. The choice of the Manim library for this thesis is motivated by its versatility and active community. Developed by Grant Sanderson for the purpose of creating videos for his YouTube channel, *3Blue1Brown*, which hosts educational videos on mathematics. Similar to other open-source projects,

Manim benefits from active support with its online community, which contributes further to its development and maintenance.

Manim lets programmers create animations of mathematical objects (referred as 'MObjects' in the library) such as Circles and Squares, as well as Graphs, etc. The library offers standard functionalities to modify objects' characteristics both statically (e.g. assigning a colour to a shape) and dynamically through animations. This enables the creation of images and videos seamlessly through code.

The Manim visualization library revolves around its class `Scene`, which holds objects that appear on the screen, controls animations, and renders file. An example is Figure 9, which shows a Scene with a red Circle and a blue Square. Both shapes appear on screen through the Scenes `.add()` method. The objects instantiation, appereance in the scene, and rendering of the visualization occur within the Scenes `construct()` method, which is Manim's way of creating an image or video [12].

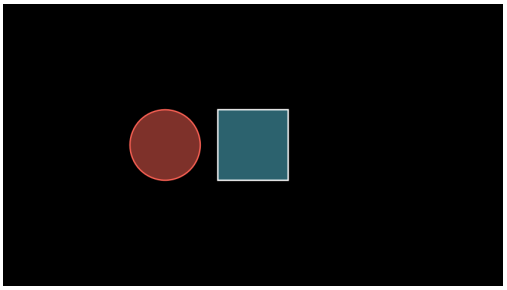


Figure 9: An example of a static Manim Scene.

Animations in Manim function as a defined sequence of images between a starting and an ending frame. An example could be the animation of a circle shifting to the left. In this case, Manim registers the initial and final position of the circle and uses the defined steps within the `.shift()` function to create the intermediary frames, thereby creating an animation. It is said that the Scene shifts the circle.

The importance of this method of animation lies in the fact that an animation must always be a transition from one image to another, both which are specified prior to the creation of the intermediary frames, which are in turn also predefined in a given method capable of producing animation. An effect of this design is that two separate animations cannot be played on the same object at the same time, as this messes with Manim's ability to properly display both the final and intermediary frames. In the case of data structures, one might wish to `Indicate` (an animation which temporarily enlarges and colors an object) and write a value in a slot simultaneously. This is not possible using Manim's own animation methods, and would require the creation of a custom animation method, an effort which this project foregoes.

4 Theory

In the following section we introduce a range of theory and research that are deployed and considered within the thesis and during the development of the visualization library. The section thoroughly describes the most relevant theory, before synthesizing this into a general approach to visualization. Finally, related work is addressed.

4.1 The Role of Mental Models in Data Structures

4.1.1 The Nature of Mental Models

In chapter six of her work *The Programmer's Brain*[23] Hermans describes how in the pursuit of problem-solving and comprehension, humans often rely on mental models. These mental constructs facilitate our understanding of complex concepts by abstracting essential details and highlighting relevant information. In the world of programming, where intricate systems and algorithms abound, mental models play a crucial role.

Mental models are cognitive frameworks that individuals create to navigate the complexities of the world around them. Just as a child may construct a number line to solve arithmetic problems or a programmer may sketch an architectural diagram to reason about a software system, these models serve as cognitive tools for problem-solving and comprehension[23].

One of the central functions of mental models is to provide constraints on how we perceive a concept. In this sense, they function as both enablers and limiters of understanding. By focusing our attention on specific aspects of a concept, mental models allow us to reason effectively about the subject at hand. However, this selective focus can inadvertently obscure other facets of the concept, leading to an incomplete or biased understanding. As such the models we use to represent concepts present trade-offs, meaning we must be specific in our use of models.

For example, the conversion of a number to its binary representation presents a model of the number which simplifies its division by 2. This illustrates an argument for the utility of models tailored to specific tasks. The trade-off here may be that others find understanding the number hard, limiting the representation's usefulness in terms of communication. Different models provide different perspectives on a concept, each suited differently for a range of tasks. Using different models may thus enrich our understanding by revealing previously overlooked dimensions, and enable different use-cases.

4.1.2 Visual Metaphors and Abstraction

To enhance the effectiveness of mental models, individuals often employ visual metaphors and abstractions. Concepts such as "child of node", "root", "branches" and "leafs" are employed to abstract and explain references between pieces of memory, aiding in the understanding of tree data structures[23]. These visualizations play a pivotal role in the creation of mental models. They provide a means to structure and organize information, enabling individuals to generate coherent and accurate representations of abstract concepts.

The Johnson-Laird's *Table-setting* experiment[24] provides empirical evidence of the use of mental models potentially functioning in a visual fashion. Participants read

a description of a table-setting scenario and were then asked to remember and reason about it following a series of unrelated tasks. Among other questions, participants were asked to choose written explanations that matched what they originally read. Interestingly, participants would just as often select descriptions different from, but describing the same table-setting, as descriptions that exactly matched what they previously read. Johnson-Laird inferred from these results that participants referenced their mental models of what they read more often than they remembered the exact words[24]. This underscores the importance of mental models and their form in both understanding and retaining complex information.

4.1.3 Metaphoric Misunderstandings and The Cognitive Ease of Metaphors

The choice of metaphors in mental models can greatly influence understanding. While metaphors help in the creation of mental models and the understanding of concepts, they may also lead us to misunderstanding. Hermans[23] writes about how describing a variable as a box into which we may place values, can lead to confusion for new programmers. That a variable can have no more than one value, i.e. only one thing can go into the box, is not immediately obvious. Hermans argues that a nametag metaphor for variables is more accurate and would lead to a better understanding of variables as a concept.

Hermans also introduces the idea of cognitive ease to describe how easily people can adapt to a metaphor. Again discussing the concept of variables in coding, Hermans presents them described metaphorically as both boxes and monocycles. Disregarding the appropriateness of these metaphors allows us to hone in on the cognitive ease of them. Hermans claims people in general have a more intuitive and clear understanding of the concept of boxes rather than monocycles, making boxes a cognitively lighter metaphor to use.

Looking at the literature describing one of the data structures relevant to this thesis, the stack, we can see how the choice of metaphor affects the concept's understanding. The two algorithms textbooks, *Algorithms* and *Introduction to Algorithms*, use different metaphors for stacks, the first describing them as stacks of paper, the latter as spring-loaded plate stackers in cafeterias. Quite obviously the first metaphor is cognitively easier and more immediately accessible to most, while the second requires knowledge of such plate stackers. However the second metaphor is technically more accurate, as it is possible to flip paper stacks over, or retrieve a paper from the middle, actions which are not possible with the stack data structure or plate stackers, where only the top element may be retrieved at any time.
Exactly here, we witness the trade-offs that arise when creating mental models. Some metaphorical abstractions facilitate more intuitive understandings, while others, though potentially more challenging to grasp, offer a more thorough description of the concept at hand.

Through Herman's work, we gain an appreciation for how mental models, visualizations, metaphors and abstractions are tools utilized by the programmer to manage their knowledge and understanding within computer science. Herman's identifies several pitfalls of this practice, showcasing how misconceptions and misunderstandings can occur. Metaphorical misunderstandings, the removal of information during abstraction, and the cognitive ease of metaphors all play roles, for better or worse, in

programmers' understanding of computer science concepts. These challenges set the stage for the creation of tools to enhance their comprehension, as we delve into the concept of abstraction in the following section.
We explore the concept of abstraction to gain perspective on its benefits and drawbacks, as well as how it may be optimally used for conceptual understanding.

4.2 Abstraction

This paper deals in length with the concept of abstraction and its use within computer science, by producing representations of data structures at different levels of abstraction, and arguing that the examination and understanding of these should be done at multiple levels of abstraction to master their comprehension. It is therefore helpful to have a thorough understanding of the abstraction concept itself, to frame its use within the paper. By examining the definitions and uses of abstraction as a concept by different researches, we arrive at an understanding of the abstraction concept useful for reasoning about the visualization product.

Kramer[27] argues that abstraction skills are crucial in the work of software engineers and computer scientists, claiming that the presence, or lack thereof, of such skills is what separates those who can and cannot create clear and elegant designs and programs. Kramer sees the furthering of research into abstraction as a way forward for computer science, and urges further work on the understanding, teaching and testing of abstraction skills. In determining how well their students *'thought like computer scientists'*, Perrenet et al.[34] measured students' ability to think abstractly. Leung et al.[17] believe that abstraction is *'one of the core cognitive practices that computer scientists deploy...'*, while Colburn and Shute attribute major breakthroughs in computer science primarily to the use of abstraction[11].
Kramer identifies important aspects of the definitions of abstraction, the first of which point to decomposition, the removal of details, simplification and focused attention of the abstraction process:

- The act of withdrawing or removing something, and;
- The act or process of leaving out of consideration one or more properties of a complex object so as to attend to others.

And secondly the aspect of generalization to arrive at a common core or essence:

- The process of formulating general concepts by abstracting common properties of instances, and;
- A general concept formed by extracting common features from specific examples.

These separate ways of abstracting are described by Leung et al.[17] as *decomposition* and *generalisation*. The decompositional approach employs a top-down, reductionist or modelling perspective, to focus on the essence of the concept at hand, while removing unnecessary details. The generalisational approach formulates concept by identifying common traits among many instances, and as such performs a bottom-up generation of a concept. Kramer provides examples for the two methods with a Matisse painting and the map of the London Underground. Matisse creates a picture of a woman using paper cutouts, which holds barely any detail, yet still conveys the subject, exemplifying generalisational abstraction.

In his study, Kramer[27] explores these methods further, presenting two maps: the London Underground map from 1928 and the map made by Harry Beck in 1993. Figure

10, illustrates the two maps, with the latter (bottom) revealing a simplified map, well-suited for subway travel but little else, showcasing decompositional abstraction.

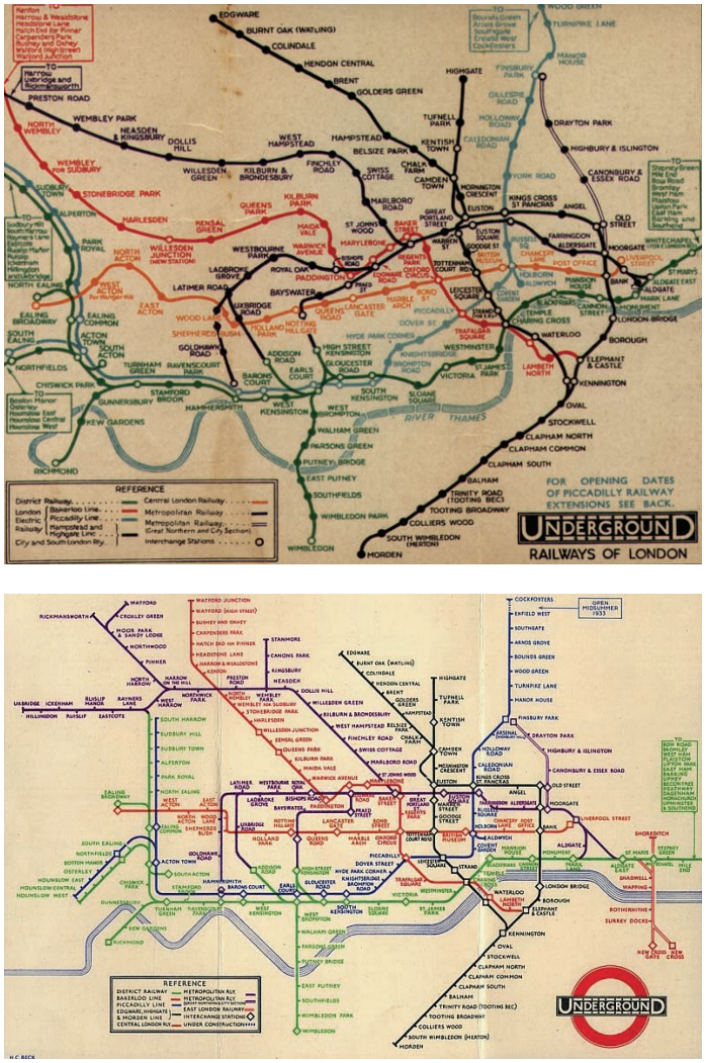


Figure 10: Comparison of the two London maps by Kramer[27], top is from 1928 and bottom is the abstracted 1933 map by Harry Beck.

4.2.1 Floridi on Layers of Abstraction

In thr paper *Levelism and the Method of Abstraction* Floridi et al.[18] perform considerable work to create a thorough explanation of what they call the *'Method of Abstraction'*. Floridi describe how systems (objects, theories, concepts, etc.) are viewed through their "observables", which may be physical features or constructed conceptual artefacts. Some examples are the height of a person, and the colour and sweetness of

wine, which are directly observed. An observable may be constructed from another observable, such as the quality of wine being determined, in part, by its age. A collection of observables is often needed to properly understand and access a system. Floridi asks us to imagine the observable *files* in chess, being an observable describing the state of the columns stretching from player to player. If we only view a chess game through this observable, it will make little sense. Each file holds a collection of pieces, and these pieces move from file to file during a game, but how the game functions would seem almost random. Adding the observable *ranks*, being the rows on the board, we would be able to reconstruct and discern the game of chess.

A given combination of the observables of a system results in what Floridi calls a "Layer of Abstraction" (hencefort referred to as "LoA"), which is a way to view a system. Floridi again uses the example of the quality of wine to showcase their theory. Quality can be seen as an observable itself, but it can also be seen as a LoA combining the observables; *Nose*, *Robe*, *Colour*, *Acidity*, *Fruit* and *Age*. Another LoA could be the purchasing LoA, being the way a potential buyer views a wine. This LoA might contain observables such as: *Maker*, *Region*, *Vintage*, *Supplier*, *Quantity*, and *Price*. In this way, systems have many LoA, the usefulness of which depend on the perspective one is taking. The relationships and interactions between observables within a LoA are denoted with a predicate, stating which combinations are possible for the system being modelled. A wine cannot for example be white and highly tannic[18]. Viewing the different depictions of basic data structures through this lens proves very helpful, as we shall see.

Aligning with our ambition to provide comprehensive visualizations of basic data structures, is the authors' idea of the "Gradient of Abstraction" (hencefort referred to as "GoA").
In their words *"A ... GoA is a formalism defined to facilitate discussion of ... systems over a range of LoAs"* which *"... provides a way of varying the LoA in order to make observations at differing levels of abstraction"*[18]. A GoA is made up of the different representations of a system, ie. LoAs, and allows for viewing the system from different perspectives.

Using discrete mathematics terminology, Floridi says there exists a family of relations between LoAs within a GoA, meaning observations in one LoA are explicitly related to observations in other LoAs of the GoA. In this way, observations, predicates and behaviours are consistent between LoAs, thereby connecting them. The nature of relations within an LoA may take various forms. Those of special interest are disjoint relations, which provide complementary information, and nested relations, which provide progressively more/less information. A disjoint relation occurs when two LoAs share no observables. A nested relation occures when one LoA contains all observables of another, along with additional observables. In this context, when LoA_j contains more observables than LoA_i and thereby more detail, it is said that LoA_j is at a lower abstraction level than LoA_i. An example of a disjoint relation is that between services in a house, which could include the LoAs; plumbing and telephone service might share no observables. Nested relations may be exemplified by a returning theme, wine, where two separate "kind" of LoAs, one cosisting of [*red, white, rose*] and another of [*stillred, sparklingred, stillwhite, sparklingwhite, stillrose, sparklingrose*], show the nested relation[18].

Floridi describes how models over systems are arrived at, at a chosen LoA for a specific purpose. Being explicit about the LoA one operates at, and why, is what Floridi

calls *the method of abstraction*, which is the primary contribution of their paper[18]. This specificity provides several advantages. One of these is that the explicitness of LoA provides context for what questions a model or view can be asked. They describe how an LoA can be seen as taking input, being data on a system, and outputting a model. What information and answers the model holds, depends on the chosen LoA. This frames the scope of the model being developed, and thereby its uses. Additionally, explicitly stating a LoA prevents ambiguity regarding what details, and at what level, we examine a system. Furthermore, explicitly chosing a LoA essentially makes clear and specifies what one believes about the fundamental nature of the subject one is studying, providing context for the subsequently developed models.

In a similar fashion, we aim to express our findings in this paper through different explicit LoAs. The goal is, together with preventing ambiguity on the concepts explained, to explore and describe in the best possible and comprehensive way, the notions of specific data structures within computer science. Using the epistemological method of abstraction, which can be simplified as *the way we look at things and what rules we use to understand them*, we can provide valuable visualisations and animations for enhancing the understanding of certain concepts.

As Floridi suggests, different levels of abstraction need to relate to each other, and the manner we intend to link the different levels is from higher (more generic) to lower (more specific) level. For example, the lower level of abstraction in our produced visualisation is how memory in a computer works, while a level higher, visualisations depict how a resizing array doubles or halves its size when needed. In a disjoint layer, a graph showing the computational cost of array operations enhances the explanation of the concept.

4.2.2 Koppelman and Van Dijk on teaching abstraction in Computer Science

Similar to F & S' work, Koppelman and Van Dijk's [26] *Teaching Abstraction in Introductory Courses* investigates the intricacies of abstraction and its various levels. The authors' aim is to explore the challenges confronted by novice students when learning abstraction and research methods.

One argument that can be deducted from their work, is that the learning journey in introductory courses within the field of computer science, should ultimately lead to mastering the abstraction skill. Drawing inspiration from E.W. Dikstra's argument[15], that one of the most important activities of a knowledgeable programmer is exploiting their power of abstraction, the authors suggest that the ability to correctly and effectively use abstraction skills is vital within programming. Koppelman and Van Dijk's analysis finds a prevalent shortcoming in existing learning and educational resources, notably books, which predominantly focus mostly on describing the concepts' results, and not the abstraction itself. For example, it is argued that one of the notions that require to master abstraction is recursion, since it is often an unfamiliar conception and needs the learner to think in a different manner. The authors insist that one of the turning points in understanding it, is to avoid focusing on too many details(using a *generalisational* abstraction [17]).
On the contrary, they assert that many textbooks focus specifically on describing the various processes generated by recursion, but not its abstraction. Furthermore, an essential element of their argument, is that he idea that abstraction is complex and hard

to understand has to be removed and therefore the development of these skills, should happen early and gradually.

4.3 Duval on Inempirically Accessible Knowledge

The philosopher Raymond Duval has done work within mathematics education which may be used to inspect how we understand computer science concepts. In his book *Understanding the Mathematical Way of Thinking - The Registers of Semiotic Thinking*, Duval outlines how semiotic representations of mathematical concepts allow us to learn, understand and work with them. This section delves into this thinking and proposes its transfer to the realm of computer science. The following sections are based on previous work by one of the authors, from a research assignment at the IT University of Copenhagen[40].

Duval examines the assertion that the process of accessing knowledge objects begins with direct experiences involving tangible objects, followed by their symbolic depiction, which eventually culminates in mental representation and conceptualization. Duval challenges this notion within the realm of mathematics, asserting that mathematics constitutes a unique case. At the heart of his argument is the proposition that mathematical concepts are beyond empirical reach, preventing direct observation. Consequently, mathematical concepts must be approached through representations[16].

A mathematical entity that lies beyond empirical observation is the concept of integers. To depict integers, one can employs unit markers, such as dots or lines, a method advocated by both Husserl and Wittgenstein[19][28]. When we combine these visual representations with words and a assortment of numerical system representations, the result is the juxtaposition depicted in Figure 11.

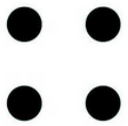
Unit marks	Language based	Number system based
	Fire Four Vier Quatre	Decimal: 4 Binary: 100 Fraction form in the decimal system: 64/16

Figure 11: Juxtaposing representations of the number 4.

Among these representations, one may ask which is actually the integer and not a representation? Duval would claim they are all representations. The concept of "four" is only conveyed by the unit marks under specific conditions, such as having the ability to count, while the presentations based on language and numerical systems are unquestionably representations, as they are products of semiotic systems rooted in social conventions[38]. This underscores how all these presentations are indeed representations and serves as an exemplification of a mathematical object's inherent inaccessibility through empirical means.

4.3.1 Duval's Semiotic Registers

To enhance our comprehension of various methods for representing mathematical objects and to categorize these representations based on their distinctive attributes, Duval introduces a spectrum of semiotic registers, as illustrated in Table 12.

A key distinction in the realm of representations lies in the differentiation between discursive and non-discursive registers. Discursive registers encompass languages, both spoken and written, that convey meaningful units of thought and thought processes, as well as mathematical expressions and symbols. In essence, they are linked to activities involving the interpretation and comprehension of meaning. These registers are process-driven, emphasizing the dynamic manipulation and transformation of information to convey ideas and concepts. In contrast, non-discursive registers are characterized by static visual depictions of objects or entities. Examples of non-discursive registers encompass diagrams, graphs, images, and other visual representations[16][33].

	Discursive registers	Non-discursive registers
Multifunctional registers	Natural language, spoken or written that creates meaning units.	Iconic imaging such as drawings, sketches, and non-iconic geometric figures
Monofunctional registers	Symbols, including number systems and formal writing	Cartesian diagrams and graphs, including strokes and arrow joining marks or nodes

Figure 12: Semiotic registers[33]

The second distinction involves multifunctional and monofunctional registers, concerning the versatility of semiotic systems in cognitive tasks. Monofunctional registers are designed for specific cognitive functions, following strict protocols and rules. They excel in precision and efficiency within their designated task but lack versatility. In contrast, multifunctional registers are highly adaptable, accommodating a wide range of functions within a single system. They support diverse cognitive tasks and facilitate complex cognitive processes across various domains and contexts[16][33].

4.3.2 Duval and Computer Science

The non-empirical nature of mathematical knowledge draws a parallel to computer science (Hencefort referred as "CS"). In CS, we often handle layers of abstraction that combine to create understandable and functional constructs. As depicted in Figure 13, we witness the transformation of Java code into lower-level programming languages before reaching the hardware level. Each of these layers represents a distinct language employed to represent the underlying object, which, ultimately, comprises bits of silicon and other compounds manipulated by electrical impulses.

In computer science, when working with data structures, the hardware-level, tangible object is nearly imperceptible and lacks significance for most individuals. It is through semiotic representations, such as human-readable code and graphical visualizations, that we gain comprehension and harness the potential of these data structures. In

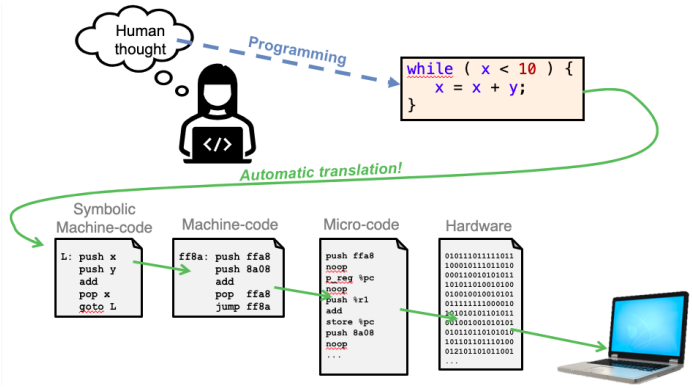


Figure 13: Layers of computer languages[6].

a manner similar to how we grasp mathematical objects through semiotically generated representations, we likewise access and engage with objects in CS.

How should we then categorize code? We can start with human-readable code like Java or Python, which intuitively falls into the discursive, multifunctional category due to its linguistic nature. For instance, Python's versatility spans various functions such as web development, data analysis, and more, rendering it multifunctional. On the opposite end of the spectrum, binary code operates at the lowest level within computing systems, serving as the fundamental language for digital communication and computation. It excels in representing data, instructions, and commands for electronic circuits but remains confined to this singular function, placing it in the monofunctional, nondiscursive register.

4.3.3 Understanding Empirically Inaccessible Knowledge Objects.

Duval states that knowledge begins when we do not mistake the representation for the object, as this shows our understanding of both what the object is and how it is represented[16]. This raises an issue within mathematics and CS where knowledge objects are non-empirically accessed. If we cannot directly access the object, and representations always differ from the object, as they are not the object, how can we then ever truly know the object? The answer lies in the juxtaposition of different representations.

Figure 14 shows a collection of different representations of a parabola. A student with knowledge of the separate semiotic systems and the meaning each representation conveys, may juxtapose these presentations with each other to find similarities. Hereby carrying out *"[the] only cognitive operation for ... giving access to mathematical objects"*, which is done by *"... the one-to-one mapping [of] meaning units from two semiotic representations differing from each other by their respective contents"*[16](p.30). I.e. recognizing how the meaning of an (x,y) pair, a point in a graph, and a mathematical equation can be mapped to each other, and thereby the object they represent. This operation then leads to *"...the recognition of the object represented by two [or more] different representations"*[16](p.31). From this way of accessing concepts, Duval concludes that the key feature to mathematical activity is the transformation of representations into other representations, to both obtain and show knowledge regarding

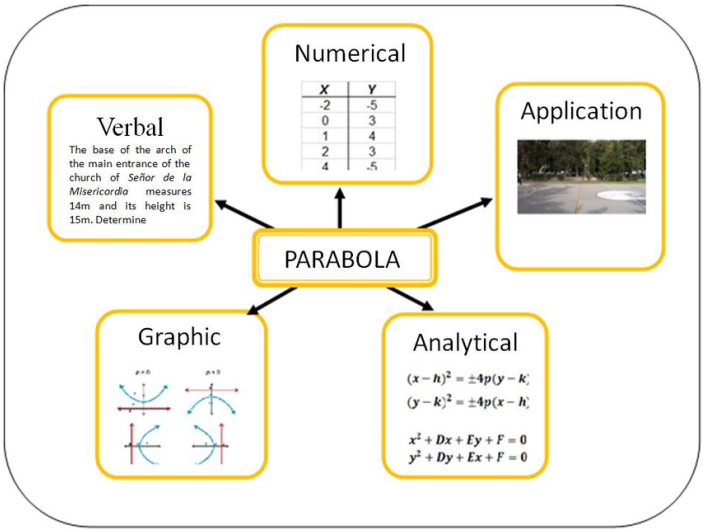


Figure 14: Semiotic records of the parabola object[35].

a mathematical object.

The idea of semiotic transformations is so central to Duval's views, that he claims truly mastering a mathematical concept involves the ability to transform a semiotic representation of a concept into all its possible representations. It is of course not feasible for an individual to know of all representations, as there are infinitely many. Duval's approach is then to propose that learning and the evaluation of knowledge can be done through practicing or evaluating an individuals ability to translate between the four semiotic registers of Table 12. Mastering translation between registers is then both a measurement for knowledge, as well as the aim and focus of teaching and learning activities. Recalling the similarities between how we access mathematical and CS objects, this framework of thought can be applied in a CS context in this project, and will be used to support the creation of multiple data structure visualizations.

4.4 Synthesizing Hermans, Floridi, and Duval: A Multifaceted Approach to Understanding

To investigate how computer science concepts are understood requires a nuanced theoretical foundation that considers the complexities inherent in conceptualizing abstract ideas. We therefore create a synthesis which draws upon the theories proposed by Hermans, Floridi, and Duval, showcasing a compelling argument for the multifaceted nature of comprehending programming concepts, specifically in the realm of data structures.

Hermans' theories on mental models and metaphors emphasizes how the selective inclusion of information necessary for understanding concepts discards extraneous details. The inherent danger of this practice lies in the potential for misconceptions. Metaphors, according to Hermans, serve as vital tools in constructing mental models, enabling us to relate programming concepts to real-world analogies. This notion sets the stage for appreciating the significance of varied perspectives in comprehending

abstract ideas.

Floridi's Method of Abstraction introduces a formalized structure to the understanding of concepts through Layers of Abstraction. Each layer provides a unique viewpoint on a concept, with observables acting as key elements within layers. Shared observables explicitly connect the layers, which together form a comprehensive Gradient of Abstraction. Floridi's framework suggests that a thorough understanding of a concept emerges from exploring its layers, thereby reinforcing the idea that concepts are multifaceted and should be understood from different angles.

Duval's perspective adds a crucial dimension by asserting that concepts which are not empirically accessible must be viewed through representations, a thought readily applied to computer science. He offers a method to categorize representations, facilitating a nuanced approach to their exploration and discussion. Duval's claim that understanding a concept involves grasping a multitude of its representations underscores the necessity of exploring diverse facets of a concept to gain a comprehensive understanding.

Synthesizing these three theories reveals a common thread: the importance of viewing concepts in different forms to achieve a comprehensive understanding. The selective nature of mental models, the layered perspectives in Floridi's Method of Abstraction, and Duval's emphasis on varied representations are interconnected concepts that collectively advocate for a multifaceted approach to the understanding of abstract concepts. Their mental models, Layers of Abstraction and representations are remarkably similar, to the point of almost being interchangeable, highlighting the complementary nature of these theories of comprehension. By recognizing the inherent limitations of singular perspectives, this synthesis substantiates the need for a holistic exploration of data structures through diverse lenses, ensuring a more robust and nuanced understanding of these fundamental elements.

This synthesis of theories thereby answers RQ2 and RQ3 by providing several solid frameworks advocating for the use of multiple representations of a given concept to foster understanding. With an affirmative response to Research Questions 1, 2, and 3, a compelling purpose for the development of a visualization library emerges, aiming to answer RQ4. The subsequent sections of the thesis will delve into the development and implementation of our proposed solution to such a library.

4.5 Related work

This section examines the existing work and literature related to the project's topic. The first examined paper[46] is an attempt to research what difficulties students may face when tackling the study of algorithms, while the following subsection investigates existing tools used for visualizing algorithms, akin to Manim.

4.5.1 Identifying student difficulties

Understanding difficulties that students face while studying Algorithms & Data Structures is a crucial point of departure for enhancing their learning experiences. In their work Zingaro et al.[46] emphasize a need for a standardised concept to assess the students' knowledge. Their work, through conducting a qualitative study of computer science students from different institutions, identifies a critical issue, so called *"over memorization"*. The concept emphasizes the necessity for students learning about al-

gorithms and data structures to develop a deeper comprehension of how to apply their knowledge, rather than memorizing a multitude of facts about data structures. Similarly, our research aims to redirect learners' focus from memorizing abstract concepts to understanding the operational dynamics of specific data structures. Visualizing their behavior through animations has the potential to bridge the gap between theory and application, fostering a deeper understanding. This shift can significantly enhance learning outcomes and alleviate difficulties associated with grasping abstract concepts.

4.5.2 Other existing visualisation tools

Java Software Package by Chen and Sobh's A few years back, Chen and Sobh in 2001[9], presented a tool to draw data structures and visualise algorithm execution for students. Their paper emphasises the fact that, although Algorithms and data structure being a very central topic for Computer Science education, its teaching has not been modernised with the latest knowledge available. In a similar manner, our research is thought to pursue an active learning approach, through visualisation and animation techniques.
Using visuals and/or video animations our aim is to enhance the learner's comprehension of data structure, by establishing a robust visual model.

JSAV In the realm of algorithm visualization for educational purposes, the JSAV (JavaScript-based Algorithm Visualization) library is a valuable tool. JSAV enhances data structure and algorithm visualisation for educational purposes. The objective of JSAV development, was to bridge the gap between modern web technologies and algorithm visualisation, making a valuable tool that is used by educators when seeking to improve their students' engagement and understanding within data structure and algorithms[25]. The Library was applied practically in a university course, which project was called OpenDSA to show its effectiveness in teaching scenarios. Similarly, this thesis could set the base for creating educational material for Algorithms courses, and future endeavours could assess its use in classrooms.

Heapviz In the context of software development, Heapviz is a tool researched in 2010 by Aftandilian et al.[3] Its primary purpose is to assist programmers in debugging programs by offering interactive data structure visualisation, particularly focused on heap snapshots during operations. As the name may suggests, the tool was initially developed for visualising heaps. When dealing with large, pointer-based data structures in a program, the paper investigates the challenges programmers face and proposes a visual solution through their tool. This visual representation enhances comprehension, especially for operations and their associated costs, offering a more intuitive understanding that can be valuable in both debugging and learning contexts. Comparably, we aim to establish a foundation for visualisation material, which will aid *"debugging"* the understanding of data structure related concepts. Unlike Heapviz, our produced visuals are not dealing with large data sets; instead, their core focus is to present, in an animated manner, the behaviour of specific data structures.

4.5.3 Online tools

Online tools such as Visual Algo[2] (in Figure 15) [2], Data Structure Visualization from the University of British Columbia [30] [3], and CS1332 [1] [4] serve as valuable platforms for learning algorithms by practicing and visualizing their operations.

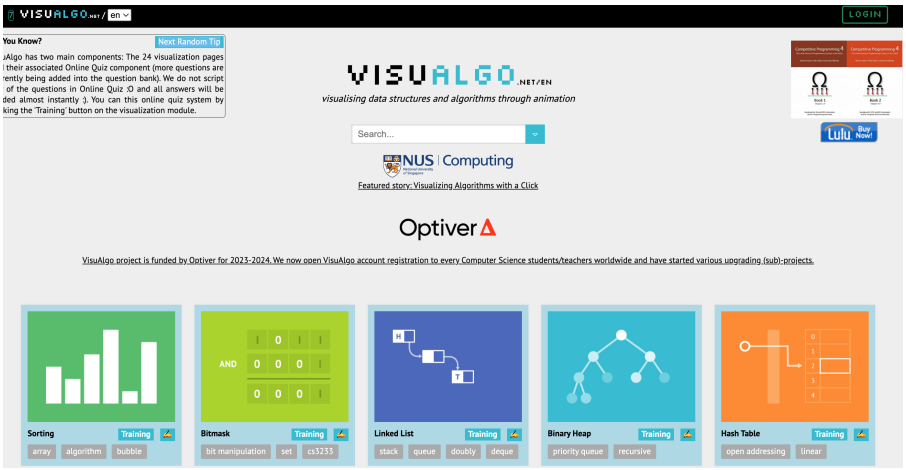


Figure 15: Home Page of Visual Algo[2].

A common thread among these tools is their emphasis on intuitiveness and visual representation. Users can input values, typically integers, into various data structures, and these platforms visualize the operations through animated simulations. The covered structures include Heaps, Resizing Arrays, Stacks, and Trees, with operations like sorting and searching at the users' disposal. These website platforms actively engage users, prioritizing practice as the initial step to enhance comprehension. Users can observe the results of operations in real-time, fostering a deeper understanding. The gamified experience makes the entire learning process not only visually appealing but also highly engaging and informative. Furthermore, several YouTube content creators, such as Grant Sanderson on his channel *3blue1brown*[37] [5] and *Reducible* [36] [6], delve into the world of algorithm operations. Grant Sanderson, the primary developer of the Manim library, extensively employs it in his content, creating visually pleasing and engaging videos that serve as effective tools for explaining and learning algorithmic concepts. This aligns seamlessly with the objectives of our work, as we try to investigate and develop similar data structure visualization aids to improve its comprehension.

[2] visualgo.net
[3] ubc.ca/Algorithms
[4] csvistool
[5] 3blue1brown YouTube
[6] Reducible YouTube

5 Method

This section outlines our methodical approach to thesis development. It describes how we explore the Manim library, conduct a targeted literature search, deploy an Agile-based software development methodology, and iteratively design our visualizations. Practical considerations and tool utilization are briefly discussed.

5.1 Exploration of Manim

The development of this thesis' library of custom Manim classes, along with the supporting research, was inspired by several sources. These include the YouTube channel '3blue1brown' [7], our supervisor Thore Husfeldt, and the previous work done by students at The IT University of Copenhagen. Initially, our experience with the Manim library[12] [8] was through the work done by these three sources. This motivated us to harness its potential for educational purposes, especially for elucidating complex concepts related to data structures.

However, before doing so, we embarked on a thorough exploration of the Manim library and documentation(Fig. 16).

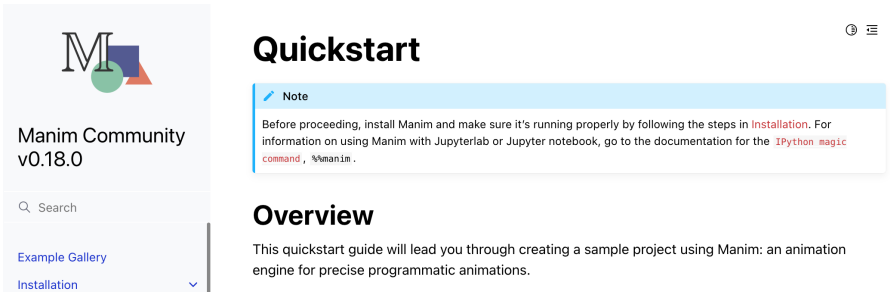


Figure 16: Snapshot of Manim Documentation[12].

During our initial exploration phase, we engaged with the Manim library to gain insights into its structure and functionality. Through the use of Manim tutorial videos, we learned about the library's fundamental features and its capabilities in crafting technical visualizations. Concurrently, a thorough examination of the Manim documentation deepened our theoretical understanding, familiarizing us with the nuances of its existing classes and methods.
Subsequently, we followed a series of guides within the documentation, where we recreated a set of animations, providing hands-on experience with the library. Experimenting with animating various Mobjects, and creating different animated videos, provided a deeper understanding of Manim and its uses. This experimentation allowed for us to begin developing our own Manim classes, and thus begin developing our data structure visualization library.

[7] 3blue1brown YouTube
[8] Manim Community

5.2 Literature Research

In conducting our literature research, we strategically navigated the scholarly landscape to gather valuable insights and inform the theoretical underpinnings of our study. Our approach involved utilizing prominent academic resources, notably Google Scholar[9] as a primary tool for exploration. To narrow our focus and ensure relevance, we employed key phrases such as *'computer science education'*, *'abstraction'*, *'levels of abstraction'*, and *'data structure visualization'*. This targeted search allowed us to sift through a vast array of scholarly works, ranging from articles to conference papers and beyond. The initial screening involved reading abstracts to gauge the alignment of each work with our research objectives. Through a meticulous selection process, we identified and prioritized articles that were most pertinent to our study, ensuring a comprehensive and nuanced understanding of the existing literature landscape. Subsequently, we engaged in a thorough examination of the selected articles, delving into their content to extract valuable insights and perspectives. This systematic literature search served as a robust foundation, enriching our conceptual framework and providing a contextually informed basis for the development of our thesis on creating visual representations of data structures using the Manim library.

5.3 Practical Workflow

The development team for this project comprises Johannes and Nicholas, both students at the ITU of Copenhagen, graduating from the MSc of Software Design. The team worked in-person twice a week, and utilised remote work when necessary. During the project Notion has been used for organising note-taking and keep the backlog, Whatsapp for communication, Microsoft Teams and E-mails to communicate with the project's supervisor. The software repository has been kept in ITU's Github, and code has been edited locally with Visual Studio Code editor. Additionally AI tools such as Chat-GPT, Bing Copilot and GitHub Copilot have been used throughout the project. While the first two tools helped when needing to rephrase and correct our academical writing, GitHub Copilot has helped produce boilerplate code and auto completion suggestions based on project's existing files.

5.4 Method Reflections

The following section presents a reflective analysis on the methodologies employed throughout the work of this thesis. It provides insights on the interplay between our day-to-day approach and established Software Development theories. Furthermore, explains the designs considerations made along the research with the tools adopted, such as the Manim library.

5.4.1 Agile Approach

The development methodology employed in this research draws inspiration from the field of Software Engineering, with a specific emphasis on an Agile approach[5] that takes loose inspiration from the Scrum Framework[39]. The adoption of selected Scrum concepts enhanced our workflow and processes, aligning with the iterative nature of

[9] scholar.google.com

Agile development. The decision to selectively incorporate Scrum elements is due to certain Scrum aspects being tailored for larger software development teams, and their inclusion would not have benefited our smaller team. The flexibility in adapting Scrum principles allowed us to tailor the methodology to the specific needs and size of our project, reinforcing the agile and iterative essence of our development process.

The deployed Scrum concepts were those of: 1. *Incremental delivery*, 2. *Iterative development*, 3. *Product backlog*, and 4. *Sprint backlog*.
In practice, this was done by brainstorming ideas, internally refining them, and subsequently discussing them with our supervisor (Thore Husfeldt), before placing them in our a product backlog, in the form of a to-do list. Keeping this list updated was helpful to break down features and actions needed to further development our product. For example, to solve the visual overlapping of arrays when resizing, we first needed to brainstorm ideas by drawing sketches(Figure 17). Once completed, we needed to explore Manim's documentation and look at our existing files to find the correct shapes and figures that could help us develop the visualization, then develop different iterations until satisfied with the result. Ensuring that continual new development did not interfere with existing features made our iterative development function as incremental delivery. On a given day of development, the backlog was reviewed to identify the most important tasks, leading to a sprint-backlog like daily to-do list, an example can be seen from Figure 18.

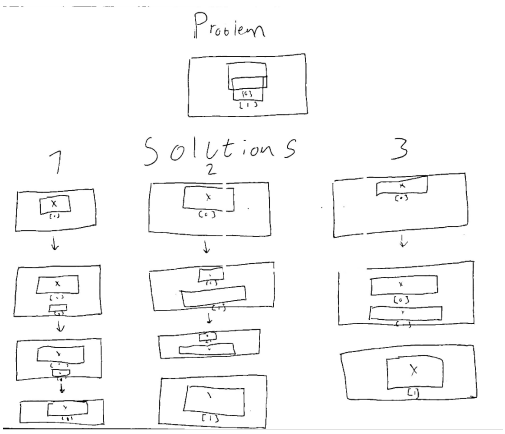


Figure 17: Sketch depicting our brainstorming process

In the course of our development activities, the practice of Pair Programming emerged as a central Software Development technique. This collaborative approach involved two developers actively engaging at a single computer, seamlessly alternating between distinct roles. The so-called "driver", responsible for code composition, and the "observer", scrutinizing and reviewing the ongoing work, created a dynamic partnership. This intentional collaboration not only elevated the overall quality of our codebase but also proved to be exceptionally advantageous for the mutual sharing and enhancement of our proficiency in the Python programming language, particularly in the nuanced application of the Manim library. The synergy achieved through Pair Programming contributed significantly to the refinement and optimization of our coding practices throughout the project.

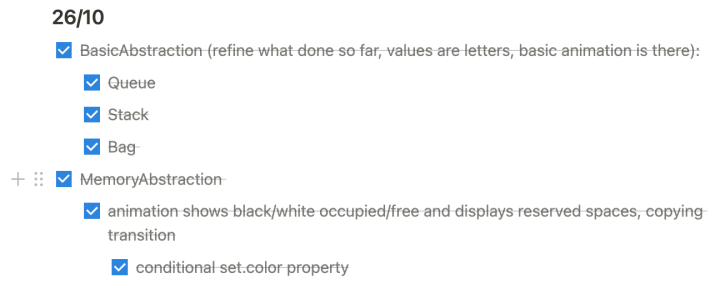


Figure 18: Excerpt of our backlog for 26.10.2023

The approach to writing the thesis you are now reading closely mirrored the agile development process employed in our code creation. The iterative and flexible nature of our development methodology, seamlessly extended to our thesis writing process. The incorporation of product backlog, and sprint backlog helped in structuring our writing tasks, and identifying sections needing to be written. This dual process of coding and writing, occurred in parallel, exemplified the agility and adaptability we aimed to achieve in both our software development and academic endeavors.

In our context, the use of theory served as a deliberate replacement for user feedback, which is a common practice in typical real-world software development team scenarios. This offered a unique perspective which, while valuable, highlights a future need for balancing theoretical foundations with user-centric approaches when shaping successful software products.

5.4.2 Visualization Development & Design

The development of the visualizations in this project was guided by a collaborative and iterative approach as well. The focus on creating representations that facilitate understanding of basic data structures is grounded in the theoretical frameworks provided by Hermans, Floridi, and Duval; this not only shapes and guides our choices, but lays the basis for our objective: to seamlessly integrate these insights into the design and refinement of the visualisations.
The iterative nature of the development process allowed for real-time feedback (from ourselves and our supervisor), brainstorming, and exploration of various design choices. The emphasis was on achieving a balance between conceptual accuracy, visual clarity, and educational effectiveness.

The development of our visualizations was intricately intertwined with the characteristics of the Manim library and the software development thoughts we adhered to, such as the principles of high cohesion and low coupling. The modular nature of the Manim library influenced how we structured our code to ensure each custom class cohesively collaborated with the other custom classes, while simultaneously minimizing dependencies and achieving low coupling. The inherent structure of Manim, with its scene, MObjects, and animation components, significantly shaped the architecture of our code, impacting the resulting visuals. This interplay between the software architecture and the visual development process underscores the importance of thoughtful code design when creating effective visualizations. The synergy between the Manim library's structure and our deliberate coding practices played a pivotal role in producing

compelling visuals, while also at times preventing the creation of desired visuals.

The final visualizations are the result of a thorough development process, supported by the synthesis of relevant theory. Even still, we invite critique and refinement. The substitution of user testing with the approaches pushed by our theoretical synthesis, has allowed for a well-informed development. However, with user testing being deemed beyond the project's scope, the visuals remain untested by their target audience. The Discussion section of the thesis serves as a platform for a thorough assessment of our visuals, examining the potential impact of selected representations, identifying conceivable challenges, and exploring opportunities for improvement.

By openly acknowledging the ongoing nature of the development, the thesis embraces a collaborative and evolving nature of educational resource creation. The intention is to inspire future work in the intersection of Computer Science education and visualization design.

6 Results

Through our development efforts, we have developed a Python library[10] which functions as an extension to the Manim library. Within this section we give an overview of the product and its functioning, describe the classes relevant to animation, examine utilized software patterns, and provide an in-depth view into the design and workings of select animations.

6.1 Product Overview

Our library includes classes for the visualization of resizing arrays (in various implementations), linked lists, and hash tables. Resizing arrays are visualized at four separate layers of abstraction, while linked lists and hash tables are visualized at two.

6.1.1 Product Use and Structure

Use of the library happens through files located within the main folder, "lib", of the project, which by convention are all named *"...Scene.py"*. Within these files, one can create `Scenes` following standard Manim protocol, and import the relevant visualization files from further down the directory. All concrete data structures and their supporting classes are housed in the *DataStructure* folder, while the CostGraph and ActionCostText classes are in their own folders. This folder structure is essential for the functioning of the library, due to how Manim executes files and handles imports. Figure 19 shows a reduced and *decomposed* [17] abstracted view of the folder structure.

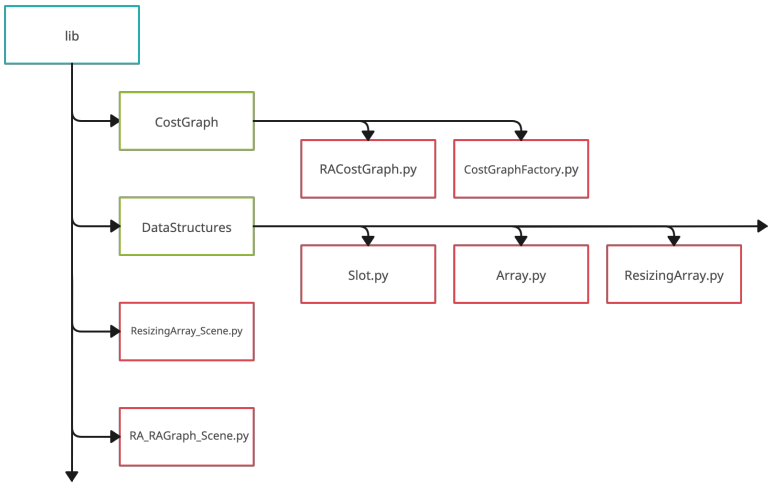
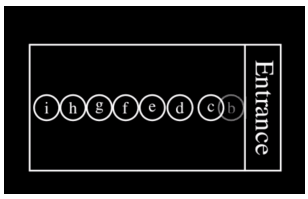


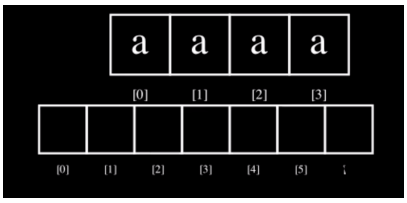
Figure 19: Edited folder structure of the created library

When executing Manim through a Scene class, all code, including that which has been imported, runs as if it was positioned in the directory of the Scene class. This

[10] Github.itu/Manim_RP



(a) Queue



(b) Resizing Array

(c) Resizing Array in Memory.

Figure 20: three LoA for Resizing Arrays.

can be described with an example: The **lib** directory of Figure 19 holds **ResizingArray_Scene.py** and the folder **DataStructures**, holds **Slot.py** and **Array.py**. A Manim Scene in **ResizingArray_Scene.py** uses the following import statement: `from DataStructures import Array`. Subsequently, **Array.py** imports Slot.
This follows the normal Python import conventions, but does not work when using Manim. **Array.py** will be executed as if it was in the **lib** directory and will therefore fail to import **Slot.py**.
This caveat necessitates our chosen folder structure, as well as all data structures classes dependent on each other importing in this manner: `from DataStructures import xyz`. This folder structure is not an intuitive one, and as thus does not relay much information, as folder structures usually do [45][8], yet is what works given our use of Manim.

6.1.2 Layered visualisations

Our aim is to simplify the understanding of computer science concepts through visualizations. To achieve this, we have implemented various *layers of abstraction*[18], resulting in different visualizations for a range of concepts.

Aligned with our deployed theory, the example of resizing arrays showcases intentional structuring in our visualizations, covering a range of abstraction levels. At the highest level, we delve into the workings of specific implementations of resizing arrays, such as Stacks and Queues. This exploration is comprehensive and multifunctional.[11] Moving a level down, we employ the widely recognized Von Neumann-based visualizations[41][14] to represent an array. Going further down, we use the same Von Neumann-based visualization to depict a resizing array in memory (See Figure 20). On a separate disjointed level of abstraction, we use a graph to illustrate the computational cost of resizing arrays. Our depiction of hash tables and linked lists are limited to a ubiquitous representation and one in memory. The combination of the four separate visualizations, are what speak into the framework created by combining Hermans, Floridi, and Duvals theories.

[11] A Layered Stack Visualization

6.2 Class Implementation

In the following section we provide a brief description of each class we developed to create our data structure visualizations.

By adopting an Object-Oriented approach, each class includes its constructor *__init__* to initialize an instance, its variables, the class definition and methods to perform operations. We will not delve into Facades and Factory classes, as their logic is discussed in Section 6.4. Scene classes will not be described either as their sole purpose is to render animations related to their corresponding objects with Manim.

6.2.1 Slot

The Slot class represents a slot of memory as well as a slot in an array, according to the Von Neumann model.

Slot is derived from the Square Class, a Standard object in Manim, making use of its predefined methods and fields(e.g. `get_center()` provides its center point or `side_length` corresponds to the side length of the square).

While each slot can be instantiated individually, its primary use is as part of a group, which represents a related data structure, e.g. an array.
Each slot instance has an Integer field called *label* which indicates its index position and a String field *value* corresponding to the data value it holds(Figure 21 below).

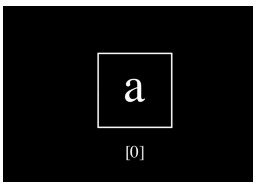


Figure 21: A Slot

Slots incorporate coloring functionality, which will be further described in the sections that discuss the classes making use of this feature.

6.2.2 Array

The Array class represents its namesake, and is composed by a group of slots. It holds a variety of methods, such as `read()`/`write()`, which collectively or individually manipulates the slots. The visual render of the class depends on whether it is used in a HashTable or ResizingArray. Figures 22 and 23 show how arrays are either arranged vertically or horizontally.

Array has various methods to perform operations, e.g. `create_array()` returns an animation that creates the figure in a Manim animation with the `Write()` animation functionality.

6.2.3 Node

A Node is similar to a Slot, in that it represents a piece of memory where a value is held. It differentiates itself by not being bound to a certain location in memory per se. Nodes are used in both high level abstractions, as well as in linked lists.

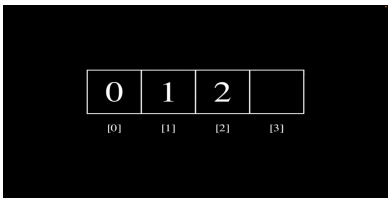


Figure 22: Resizing Array

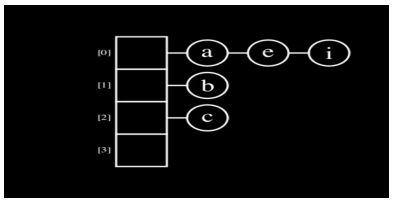


Figure 23: HashTable

From a visual perspective, it's worth noting that, for simplicity, our nodes do not have their own pointers, setting them apart from the actual nodes in linked lists.
Instead, the Linked List class manages the visual representation of pointers.

The Node class consists of a Circle and a Text object, and it is the simplest class of our library. (Figure 24)

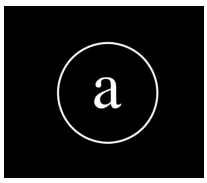


Figure 24: A Node

6.2.4 ResizingArray

The Resizing Array (henceforth referred as "RA") class and its concrete implementations (Stack, Bag, etc.) are central classes in the visualization library. They provide the visualizations and animations of the ubiquitous array representation[12], and can be used to either convey the resizing array concept in isolation, or combined with other representations to foster greater understanding.

The RA class is, by convention, an abstract class. It can be instantiated, but is intended not to, and also lacks any `add()`/`remove()` methods which makes the visualization function. Its concrete implementations implement these methods and behave according to their conventions. The class consists of an Array, except during resizing, where it consists of two. All the methods of the RA class, such as `create_new_array_on_screen()` and `is_full()`, are related to performing array resizing, and are called in the specific implementations' `add()`/`remove()` methods when necessary.

In this project, RAs are implemented as Stacks, Bags and Queues, Priority Queues, and Hash Tables with linear probing to represent their abstract data structure, as well as lending some of its functionality to hash tables with separate chaining.

Upon resizing, the creation of new arrays happens through the classes `ArrayFactory`, which is described in Section 6.4. Figure 25 shows a UML diagram of the ResizingArray class and its related classes, hash tables excluded.

[12] A Queue visualization

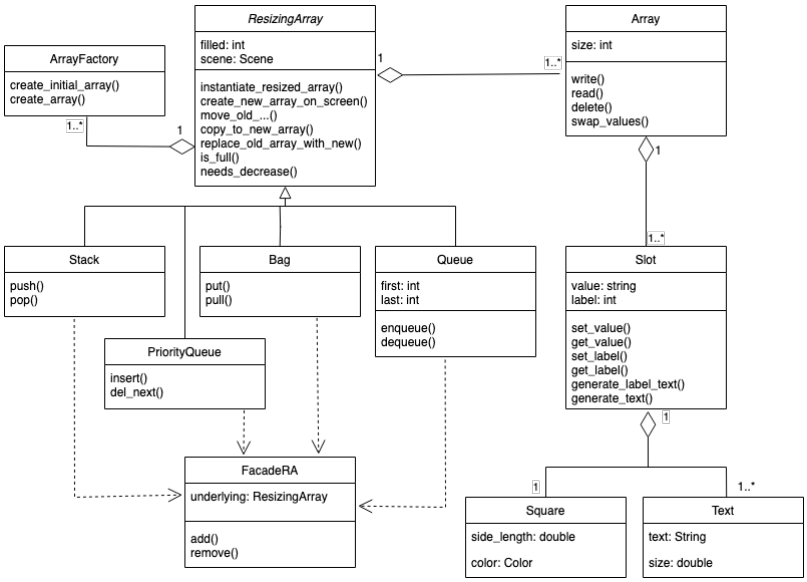


Figure 25: UML Class Diagram for Resizing Array.

6.2.5 LinkedList

A Linked List is a sequence of one or more nodes, which get added, removed, or searched through. It can be used independently, and within the library is used in the HashTable class.

Apart from its nodes, the class holds a list of Line Mobjects, which in the visualization conveys the Nodes pointing toward each other. Currently these are just lines, and no consideration as to whether they are a singly- or doubly linked lists has been made. The first line originates at a chosen position, while the subsequent nodes/lines are alternatingly positioned to the right of the rightmost Mobject, resulting in a Line, Node, Line, Node, etc. structure as depicted in Figure 26.

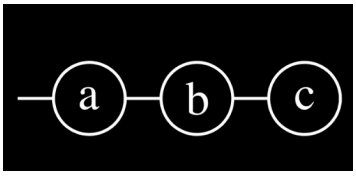


Figure 26: a Linked List

The class can create animations when adding/removing a node (by value or index), and when searching for a given value. Adding/removing adjusts the position of existing nodes at need, and creates or fades out a node and line pair.

6.2.6 HashTable

The HashTable class in our project depicts a hash table with separate chaining[13] and uses the Array and LinkedList classes to do so. The class has methods specific to its homonymous data structure, for example the method `add_value()`. This method calculates a hash value for a given input, searches the corresponding hash bucket for the input value, and adds a node to the linked list if the value is not already present. Once instantiated, HashTable is equipped to generate Manim animations that render visually what `add_value()` does in the code.

Along with the hash table with separate chaining described above and shown in Figure 4, the library also holds a Hash Table with Linear Probing class, which visually equals the resizing array visualizations(Figure **??**) [14].

6.2.7 Memory

The Memory class provides an abstract representation of a computer's memory according to the Von Neumann model, by expanding upon the Array class through inheritance. Memory distributes its slots over a set amount of horizontal lines. The lines always reach the edge of the screen, where the last slots are cut off and have an index [...], signifying the continuation of memory. Within a line, each slot is indexed incrementally from left to right, showing them to be adjacent in memory. The line indexes are numerically separated, indicating that the corresponding locations in memory are also physically apart from each other.

Similar to how they function in other classes, each slot represents a space in memory. However, we do not wish to display values, as this would lead to a cluttered visualization, and therefore slots function differently within Memory than they do in our ResizingArray class.
Instead of holding values, each slot conveys its current state. Slots have two fields: an integer *state* and a *color*. By default *state* takes integer values from 0 to 2, and *color* takes the values BLACK, RED, and BLUE, signifying the condition of the memory space. Their correspondence is as follows and as shown in Figure 27:

- A state of 0 signifies that the slot is available, corresponds to the color BLACK.
- A state of 1 signifies that the slot is occupied, corresponds to the color RED.
- A state of 2 signifies that the slot is reserved, corresponds to the color BLUE.

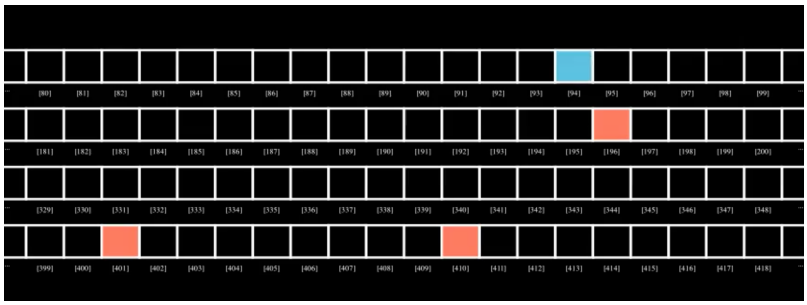


Figure 27: Memory visualisation

[13] A Hash Table with Separate Chaining Visualization
[14] A Hash Table w. linear probing Visualization

Whenever the state of a slot changes, its color is automatically updated to reflect the change.

6.2.8 RACostGraph

The RACostGraph class, short for *"Resizing Array Cost Graph"*, is used to create an animation displaying the computational costs associated with a Resizing Array data structure[15] . The class leverages the `CostGraphFactory` factory, to initialise the x and y axes, as well as dots for representing the operations. In the graph, the x-axis represents the number of operations performed, while the y-axis indicates the related cost. The data structure's operations are represented by dots in the graph, which are color-coded: red for resizing operations, which are costly, and green for normal operations, which are less expensive.

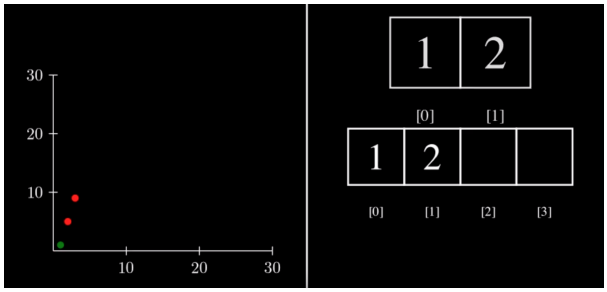


Figure 28: Cost Graph and Resizing Array

One of the key feature we implemented in the RACostGraph class, is its ability to dynamically increase the scale of its axes. This means that if the number of operations exceeds the current range of the x-axis, the graph automatically adjust its scale. This ensures that all operations are accurately represented on the graph, regardless of their quantity.

When a Manim animation is rendered using RACostGraph, it provides a real-time visualisation of the operations performed by the resizing array and their related costs. see Figure 28.

6.2.9 HTMemory and ResizingMem

HTMem ("Hash Table" + "Memory") and ResizingMem ("Resizing Array" + "Memory") are classes designed for rendering visualizations that simultaneously depict two distinct levels of abstraction. The top level on the screen showcases the specific data structure, while the bottom level illustrates the memory-level operations that occur during the data structure's functioning. Figure 29 is an example of ResizingMem while footnote[16] displays HTMem.

It is necessary to create custom classes for these combinations, due to how our classes operate. If we were to render a ResizingArray and Memory separately in a scene, the animations of their operations would be asynchronous. For instance, a resizing trigger would result in the full animation occurring in one before the other.

[15] A CostGraph + Resizing Array video.
[16] A HTMem video

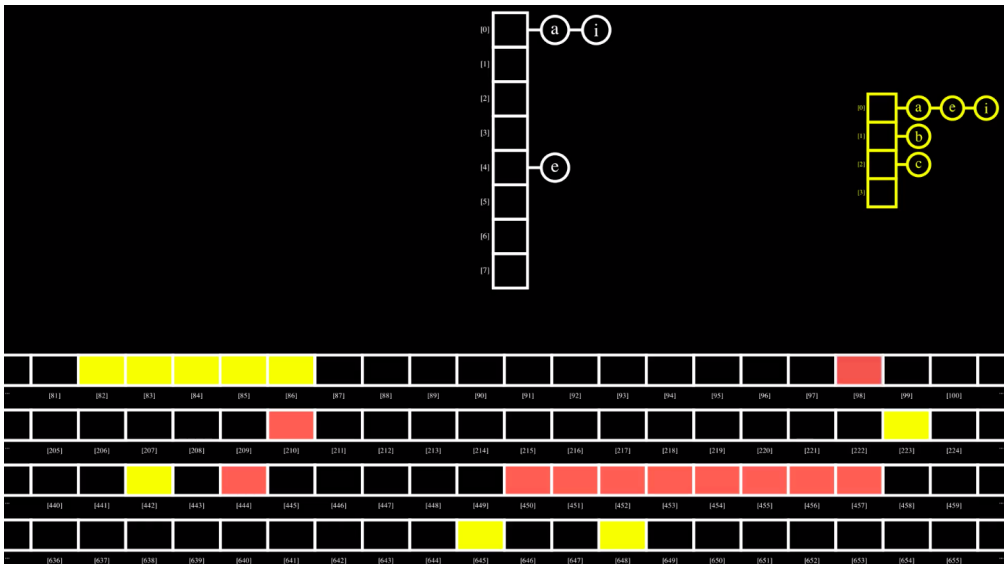


Figure 29: Resizing Array and Memory Visualisation

By amalgamating the objects into classes, we enable animations where the step-by-step operations unfold (almost) simultaneously in both representations.

The classes instantiate the data structure and memory as fields and define separate areas, which confine the different visualizations. Both classes have the standard functionalities of adding, removing, and searching for elements which they hold. How these functionalities are implemented in code and visually carried out is described in further detail in Section 6.3.

6.3 Animations in depth

As mentioned in subsection 5.4.1 our ideation process to transform initial animation concepts into developed and executable code in our library began by envisioning our desired visual outcome. To facilitate translating our ideas in code, we chose to visually sketch our depictions of the data structures and their animations(see Figure 30). The drawings nurtured the discussions and provided a quick and visual overview of our intended results.

Once satisfied with our sketches, we began coding the first prototypes. Following peer review and feedback discussion with our supervisor, we iteratively refined the prototype until achieving a satisfactory product. Using low-fidelity sketches in our development proved to be beneficial in addressing the visual feasibility of the animation, for example when determining how to represent the copying of elements when Arrays were at full capacity.

Given the fixed size of the Scene in Manim, we created a solution to illustrate data structure operations while minimizing on-screen disturbance. By shrinking the size of the source array and positioning the new array at its base, we prevented the scene from becoming overcrowded, while simultaneously displaying both data structures.

In the following subsections, we delve into the intricate code-level details of a selec-

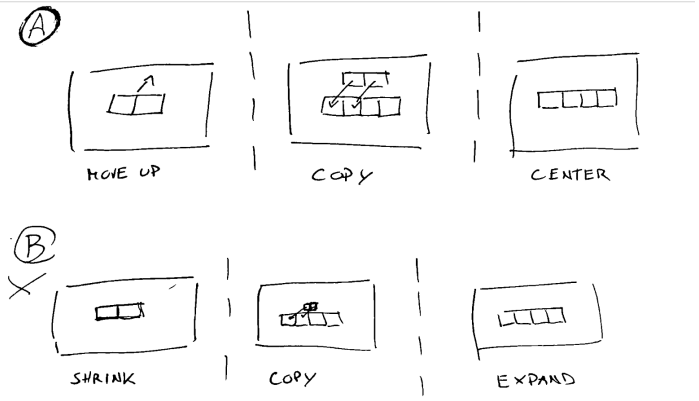


Figure 30: Sketch of ideas when copying values from full array.

tion of the library's animations.
This includes depicted actions like copying slot values, navigating a linked list, and smoothly adjusting graph axes.
It's crucial to emphasize that the advanced code behind these animations represents strategic workarounds tailored to our specific purposes. Due to the unique structure of the Manim Library, we carefully crafted these mitigation strategies to achieve precise visual effects.
The complex processes occurring in the code behind the scenes contribute to the seamless realization of the intended visual outcomes in our rendered animations.

6.3.1 Choosing Animations for Behaviours

Some functionality of our data structures have behaviours with self-evident visualizations. An example of this is the writing of a value within a slot. The exact creation animation can be discussed based on preference, but regardless the animation should simply make a value appear. Other behaviours do not have obvious animations, and here design choices must be made. An example of this is the swapping of values which occurs during the `sink()` and `swim()` operations of a priority queue(henceforth referred as "PQ").

Within `sink()`/`swim()` operations, values in a PQ swap places within an array. At the actual code level, what occurs is: value a is assigned to a variable $x \rightarrow$ The slot of value a is set to value $b \rightarrow$ The slot of variable b is set the value of variable x. This use of a temporary variable x to store the value of a during the operation is necessary in PQ implementation, but does not necessarily fit visually into our library, because we decided we did not want to animate a floating slot for variable x outside of the array.

Instead we abstract away this detail and have values a and b directly swap places by floating to each others slot, as seen in Figures 31 and 32. Such a choice is one in a long range of choices made during this project regarding which details should be included and which should be abstracted away within our visualizations.

Another behaviour which does not have a self-evident animation, is that of reading a value. When the computer reads the value of a slot in memory, what it is actually

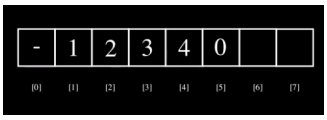


Figure 31: PQ prior to swap.

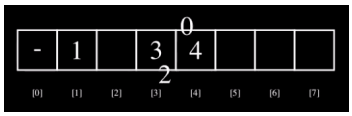


Figure 32: PQ during swap.

doing is copying that value from some level of memory to a level at or close to the CPU. Our animations are nowhere near diving into the concept of separate memory levels, so a design decision must be made. The important part is that the animation must reveal that there is some computational action, and thus a computational cost involved in reading. We here decided on the indication of a slot signalizing a read.
This draws the viewers attention to the slot, and takes time to carry out, which signifies computational cost. The reading of nodes is likewise signified by an indication animation. Figure 33 depicts a simultaneous yellow indication of a node in the hash table and the slot in the memory.

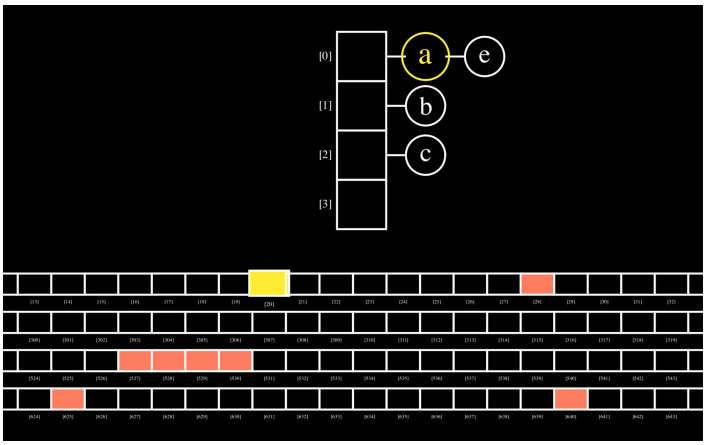


Figure 33: Indication of node and memory slot

6.3.2 Searching through a Linked List

A fundamental operation in linked lists involves searching for a specific value. Search is used for retrieving values and when performing addition or removal operations. To vividly illustrate the traversal of a linked list during a search, the library generates a dynamic visualization that persists until the target is found or the entire list is covered (See video in footnote [17]).
The traversal unfolds as follows:

1. Within our library, each node is linked to others via a Line Manim object. Once a search is initiated for a specific value, two copies of the initial line get instantiated.
2. One of the copies is flipped (its start is positioned at the end of the original line and vice versa). Both copies are colored yellow.

[17] A Linked List Search video

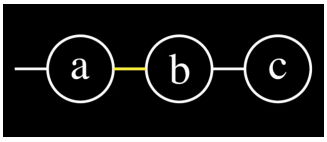


Figure 34: Searching

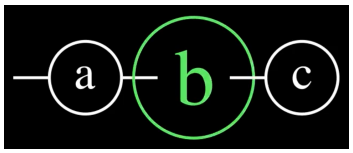


Figure 35: Value found

3. Creation of the non-flipped copy using the Manim animation `Create()`, animating its appearance from the starting point of the original line.
4. Upon the full creation of the non-flipped copy, it is immediately replaced by the flipped copy in an instantaneous and unseen transition.
5. Removal of the flipped copy with the `Uncreate()` animation, causing it to fade out from its ending point (the original line's starting point). This symbolizes reaching the Node at the end of the line.
6. Indication of the arrived-at Node, signifying the reading of its value. The indication is large and green for a successful search hit and small and yellow for a search miss.
7. Repeat steps 1-6 for subsequent Line-Node pairs until either a search hit or the end of the List.

This sequence of animations provides a visual representation of traversing linked list, offering viewers insight into the behavior of the data structure. Figure 34 and 35 depicts the yellow highlighting of the line when searching, and the green indication once the value('b') is found.

6.3.3 CostGraph dynamic re-scaling

To achieve a smooth transition effect in our graph animations, which involve instantiating the CostGraph class, we needed to implement a re-scaling mechanism for the x and y axes when the number or cost of operations exceeded the current scale[18].

Each new operation in the data structure corresponds to a new dot in the graph. The method *add_element()* which handles the logic of adding a new dot to the graph works as follows:

- First it checks if the number of elements (dots) is equal to the highest number on the x-axis or if the cost of the new operation is the greater than the highest number on the y-axis.
  - If either condition is true, the respective axes need to be resized.
  - If only one condition is true, it only re-scales and re-positions the relevant axis.
  - If both conditions are true, it means both x and y axes need to be re-scaled, as well as re-positioning the already present dots, before adding a new one.
  - If neither condition is true, it simply calls *addDots()* method, which places a dot at the corresponding x and y coordinates in the graph.

- Using then the RACostGraphFactory class, it calls *create_graph()* to create a new underlying graph.

[18] A Cost Graph video

- *create_graph()* creates a new Axes object, which represents a pair of x and y axes and takes four parameters: *x_ticks_list, y_ticks_list, resize_x, and resize_y.* The first two are lists of numbers to be displayed on the x and y axes, respectively. The last two are boolean values indicating whether the x-axis and/or y-axis need to be resized.
- If either *resize_x* or *resize_y* is true, the *create_graph()* method generates a new tick list for the respective axis by calling *generate_new_tick_list()* with either *x_ticks_list* or *y_ticks_list* as a parameter.
  - The new list is then assigned to the *numbers_to_include* attribute of the x or y axis of the new Axes object, which determines the number that will be displayed on the axes' ticks.

- It groups all the existing dots in a a Manim *VGroup()*.
- Plays an *AnimationGroup()* where the dots are shifted down, while a *ReplacementTransform()* animation transforms the old existing graph into the new one.
- Then removes the old graph and adds the new one to the vgroup.
- Finally, it calls the method *addDots()*.
  - The color of each dot is determined by the cost of the operation. If the cost is greater than 1, then the dot is red, indicating a costly operation, otherwise is green.

These operations ensure a seamless adjustment of the graph preventing any unintended changes in its size throughout the course of the animation.

6.3.4 Copying values from an old to a new array

In the Von Neumann-based ResizingArray class within the library, values from an original array are copied over to a new, resized array as part of the data structures functioning. The class needs to properly convey how the original array is read, and then copied to the new array. This happens through the following loop:

1. $Old\_Array\_Slot_i$ is indicated with an increase in size and change of color.
2. $Old\_Array\_Slot_i$ has a copy instantly created and moved in front of itself. The copy does not include an index and this cannot be seen by the viewer.
3. The copy moves directly to its destination, usually at $New\_Array\_Slot_i$, while simultaneously changing size to fit into the new array.
4. $New\_Array\_Slot_i$ is instantly given the value of $Old\_Array\_Slot_i$. This is not seen, as the copy is in front of it.
5. The copy is instantly removed, and a single copying operation is finished.
6. Increment i and repeat steps 1-5 for the length of *Old_Array*.

The use of a copy of $Old\_Array\_Slot_i$ achieves several things visually. As the copy moves to the new array, the original slot is left behind, still holding its value. This signals to the viewer that we are indeed copying, and not moving the value. The other visual effect is the clear explanation of where the value appearing in the new array is coming from.

Under the hood this method of copying is favorable, as it allows for the new array to be given the relevant values at its indexes through the Array classes' usual way of setting values, so it is prepared for its further functioning. Having the copying occur slot by slot signifies how the data structure carries out the copying, and thereby conveys the computational cost of the action.

The current copying animation[19] replaces our initial animation.
Originally we: highlighted the old slot → wrote in the new slot → highlighted the new slot. This conveyed the copying, but in a less pleasing manner for the viewer.
The dragging of the copy from the old to new array more clearly signifies that the value comes from the old to the new, as it is quite literally doing so, as seen in Figure 36.

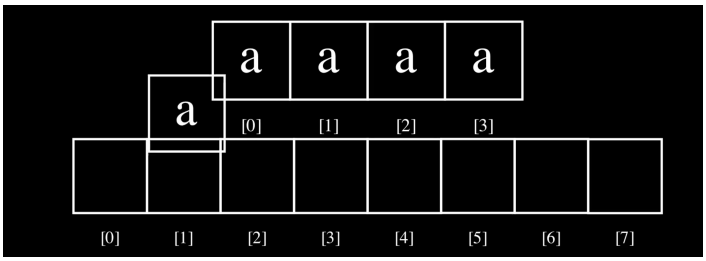


Figure 36: Copying animation in Resizing Array

6.3.5 Changing Slot state in Memory

Similar to the use of slot copies described above in resizing arrays, the Memory class also uses copies to create animations of state changes in its slots.[20]
When a slot is set to change state, it should change from one color to another. This occurs by the Memory class creating a copy of the given slot and placing in front of it. The given slot then changes color behind its copy, before the copy fades out, creating the visual illusion that the slot fades from one color to another. It is possible for the slot to simply animate its color change, but this requires the Slot to be able to play animations, which our implementation does not allow. Additionally, the implemented way of enacting color changes allows for the appearance of fading, which we found aesthetically pleasing.

6.3.6 HTMem and ResizingMem as custom classes

To understand how HTMem and ResizingMem create "simultaneous" animations, we can inspect their `add()` function. In the event of a regular add (no resizing), the classes simply call the `add()` of one and then the other visualization. As these are quick animations, we say they are simultaneous. In the Discussion section, we elaborate on how true simultaneity could be achieved and the reasons for not implementing it.

Inspecting a resizing `add()` reveals why custom classes for the combinations of the two objects are necessary. The first action of a resizing is creating a new array/allotting new space in memory, which in this case should then happen in both visualizations, requiring the respective functions that carry out these actions to be called simultaneously. Subsequent actions, such as copying values from old to new, also need to occur together. This is achieved by having the `add()` function of HTMem/ResizingMem call these underlying functions themselves, instead of relying on the `add()` functionality of their subclasses. By having the combined classes call related functions of their subclasses together, the actions of these functions happen simultaneously, and thus the

[19] A Queue video, where copying can be seen
[20] A video of slots in memory changing state

visualization of a resizing in both data structure and memory representation occur together. For continuities sake, we have chosen that "simultaneous" actions occur first in the data structure visualization and then in the memory. Figure 33 is an example of this, where node and memory are indicated at the same time. The full method may be examined in the library's GitHub repository [21].

6.3.7 Color coding of HTMem

For better comprehension of hash table functioning, the HTMem class introduces color coding. This overrides the default coloring of memory and imparts color to the hash table.
Figure 37 and footnote [22] exemplifies such rendering, where each slot in the hash table has a unique color. Correspondingly, equating slots in memory and those slots representing nodes linked to them display the same color, as depicted by the blue slots in Figure 37. This design choice makes explicit where nodes of a given linked list are in memory by providing visual cues. The color coding aligns with the utilized theory regarding layers and gradients of abstraction[18]. Layers are explicitly connected by the observables they share, and these connections serve as their relation.
The relation between LoAs is what allows people to observe them as separate representations of a singular concept. By introducing color coding, we are adding additional observables to the abstractions, in an attempt to make the relation between them more explicit.

Integrating color coding into the HTMem class prompted a reevaluation of how the hash table resizing process was managed. We opted to designate the old hash table and its associated memory slots as grey during resizing initialization, resulting in several key advantages.
Firstly, the fading to grey serves as a visual cue, indicating that this component is outdated. It communicates that the data structure is of reduced significance and is undergoing replacement.
Secondly, employing grey for the old elements contributes to a cleaner display with less color saturation, in contrast to maintaining color for the old data structure while introducing color for the new one.
Thirdly, the contrast between the new and old, with the former in color and the latter in grey, naturally directs attention to the evolving data structure where changes are occurring. Although the loss of detail in the old elements turning grey may seem significant, we deemed it inconsequential. While viewers may no longer discern which slots belong to specific linked lists, this information was previously conveyed, and the intentional redirection of attention to the new structure justifies this acceptable loss of detail.

[21] HTMemory's code
[22] Video of color coded HTMemory

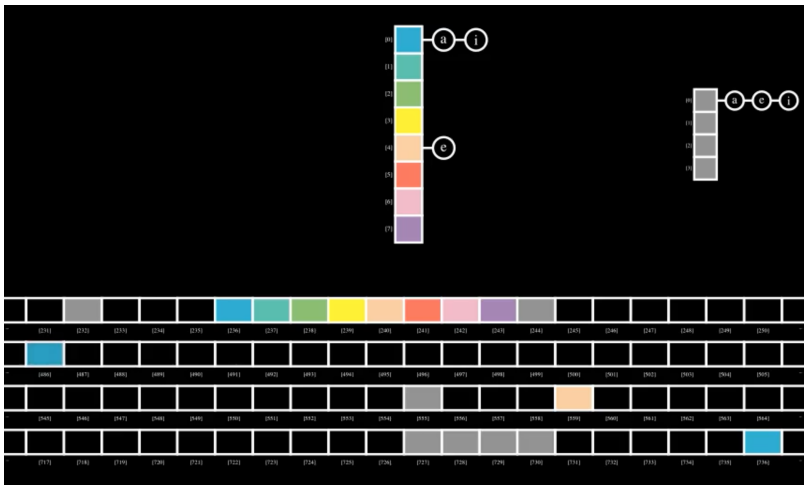


Figure 37: color-coded Hash Table Memory visualisation

6.4 Software Patterns

In the realm of software engineering, software patterns (or simply 'patterns') represent structured responses to frequently recurring design problems or dilemmas. Their origins can be traced back to the field of design and architecture.

Christopher Alexander's work at the end of the 20th century revolved around architectural solutions that cater to people's needs, resulting in patterns that can be reused to address building and architectural challenges[22]. During a workshop at the ACM Conference on Object-Oriented Programs, Systems, Languages, and Applications in 1991, Alexander's patterns began to garner attention and study within the field of software engineering. The *Design Patterns* publication by Gamma et al. in 1995, cemented the connection between these two worlds[13].

Ward Cunningham and Kent Beck were profoundly inspired by Alexander's work and applied his concepts to software development. In 1987, their book, *Using Pattern Languages for Object-Oriented Programs*[4], established the first pattern system for designing user interfaces.

Patterns are akin to architectural paradigms. They embody time-tested strategies based on best practices, leveraging the collective wisdom of senior development experts. Utilizing such patterns benefits the development process by facilitating streamlined, sustainable, and scalable systems. Numerous scientists have contributed to the evolution of software design patterns.
Nowadays, some of the most frequently used concepts include Factory and Facades which we will now explore in more detail as they pertain to our work.

6.4.1 Factory

In object-oriented programming, one well-known pattern is the Factory, which provides an interface for creating objects in a superclass. All subclasses can alter the type of objects to be created. The pattern's utility lies in avoiding the creation of diverse objects through individual classes, opting instead for objects that share a common

interface, created via a simple method call.[32]

In our project, we implemented the Factory concept for two objects: the Array and the Cost Graph. This implementation has improved the readability and maintainability of our code by encapsulating the logic in separate classes. Furthermore, we employed this pattern to streamline our development process, simplifying the creation of these two objects through factory methods.

The `ArrayFactory` class serves as a simple interface for creating arrays of different types and sizes. The primary method for doing so is `create_array()`, which takes two parameters: `array_type` and `array_size`. These parameters specify the desired type of array (e.g., `int`, `float`, `str`, etc.) and its size.

The `ArrayFactory` class is instantiated as a field of the `ResizingArray` class, and its methods are accessed through a call to it, such as
`arrayfactory.create_array()`.

Similarly, the `CostGraphFactory` is designed to create graphs with various configurations. Like the `ArrayFactory`, its method `create_graph()` requires parameters such as `x_tick_list`, `y_tick_list`, `resize_x`, and `resize_y`. Depending on the specified parameters, you can customize the size and layout of your graph. As per the functioning of the Manim library, we continuously create new cost graph objects and fade out the previous ones, rather than increasing the cost limit. Consequently, we indicate the y and x tick lists at the time of creation. Similarly, a cost graph instantiates a `CostGraphFactory` as a field, through which it accesses its logic, e.g. `costgraphfactory.create_graph()`.

6.4.2 Facade

The Facade pattern is a structural design pattern that provides a simplified interface to a set of interfaces in a subsystem. It acts as a unified entry point, encapsulating the complexity of the underlying system and presenting a cleaner and more manageable API for clients[21]. In the context of this project, the Facade pattern serves as a higher-level abstraction that hides the intricacies of interacting with different resizing array implementations.

The `FacadeRA` class functions by having a field `data_structure_type`, which holds the desired data structure object, and two functions, `add()` and `remove()`.
The functions simply call the corresponding function of the facades' underlying data structure, and in this manner abstract away both the code-level functioning of these functions, as well as their varying names. The use of the facade is intended to increase code maintainability and reusability[21] in cases where one might want to substitute the different resizing array implementations with each other.

Take the example of a user creating a visualization script deploying the Von Neumann style of a stack, along with a corresponding cost graph, and subsequently wanting to replicate the same animations with a queue. The use of the `FacadeRA` allows for this, only requiring the user to change the facades implemented data structure. This is opposed to having to both change the instantiated object and all the method calls to `add()` and `remove()`, which may in the concrete display be linked to the additional visualization used. The exact same situation may occur when making use of the higher level abstraction visualizations, hence the `FacadeAbstract` class.

7 Discussion

This section provides a thorough overview of our research. We confirm the positive impact of visualization on learning, align our library with Bruner's didactic principles, and evaluate the effectiveness of our use of Manim.

Our considerations on user experience emphasize the dynamic relationship between code design and video outcomes. A comparative analysis with *Algorithms* by Sedgewick & Wayne[41] highlights strengths and weaknesses, encapsulating theoretical foundations, practical insights, and future directions of our visualization library. In discussing future work, we underscore the importance of comprehensive user testing and acknowledge the limitations inherent in this thesis.

7.1 Effects of Visualization on Learning

According to our deployed theory, and the resulting synthesis of combining Hermans, Floridi and Duval's central points, we answer Research Questions 1,2 and 3 with a resounding yes. Having shown how popular algorithms and data structures text books use visualizations[41][14], and recognizing the synergies of the aforementioned researchers, these answers can be given with a high degree of confidence.
However, it is relevant to discuss the extent to which visualizations are important for learning. For while Hermans, Floridi and Duval's theories can readily be applied to the visual representations of concepts, they are not exclusively about visualization.

Using visualizations to convey information is ubiquitously accepted as a positive, as the popular saying, *"a picture is worth a thousand words"*, indicates.
Indeed a range of research supports the idea that visualizations help learners understand concepts. Early research into visualisation within mathematics frames learning as the process of constructing abstract mental representations by aligning imagery with logical verbal thought [10][31]. Edward Tufte, a well-known Yale professor, has published a range of highly respected books on visually conveying information[42]. Vellido et al.[43] present the importance visualization trends in the field of machine learning, while the related work section of this thesis presented work within data structure visualization. Given the existence of such research, visualizations' positive effect on learning is well-documented. As such the argument for our visualizations being helpful for data structure understanding is strong.

7.2 Didactic Considerations

As our visualization library is made to be used in the context of learning, it is imperative to examine its alignment with established didactic theories. Bruner, an important figure within didactics, created several central concepts that we can use to evaluate our library. He emphasizes that concepts should be taught at different levels and advocates for the gradual accumulation of details each time a concept is revisited, which is known as his Spiral Curriculum theory[7].
Given our library's capacity to visualize data structures at various abstraction levels, it emerges as a fitting tool for implementing a Spiral Curriculum.

Bruner also coined the term "scaffolding" within didactics, which describes tools and tactics used to help learners achieve tasks they otherwise could not[44]. Scaffolding can be the help of a teacher or fellow student, teaching materials, guides, etc.

Just as the pictures in *Algorithms*[41] help students understand data structures, so too can our visualizations, classifying them as scaffolding tools. In fact, our visualization library provides a unique form of scaffolding, in that it dynamically represents the functioning of data structures, and displays the time aspect of their operations. The computational cost of an array resizing or a linked list search are conveyed in our library through time, *scaffolding* the viewers' learning process. The explicit connections between different LoA's present in our visualizations, present another form of scaffolding which they provide the viewer. As such, our tool aligns well with central didactic theory.

7.3 Effectiveness of Manim

The Manim library offers a distinctive approach to visualization by being code-based. Its characteristic precision, where objects behave exactly as instructed by the code, aligns well with visualizing data structures. Given that data structures exist within computers and computer memory, representing them through code is particularly apt. The code implementing the visualization often closely follows generic implementations of the structures, incorporating the additional aspect of manipulating both the underlying data and the objects in the Manim scene. As such, Manim is a very appropriate tool for visualizing data structures. The following subsections examine and discuss specific consequences of our use of Manim.

Functionalities such as swapping values in priority queues and copying values are implemented within custom classes, with animated behaviors seamlessly following along. this process typically involves manipulating both the underlying data and Manim Mobjects, although in some cases, Mobject manipulation alone suffices.
Tracking the length of a linked list and the number of occupied slots in an array is best managed with underlying fields separate from the Mobjects. Conversely, the storing of values in a slot/node can be left to the Mobjects themselves.

7.3.1 Dynamic Animations

Certain operations in the implemented data structures depend on the completion of previous actions.
The `sink()`/`swim()` operations are an example of this. If a given value needs to swim two places upwards, it must complete the first swap before the animations for the second swim can be created. These second animations rely upon the values involved in the first swap to be in place, before it can begin being rendered.
This has a substantial impact on our library, as it compels that all classes handling dynamic operations must hold the Manim `Scene` as a field. These classes then play animations of operations sequentially as they unfold.

This class structure is essential for animating the dynamic operations of our chosen data structures. Additionally, it is also intertwined with the concept of the "simultaneousness" of animations featuring separate depictions of a data structure

7.3.2 Achieving Simultaneity

As previously mentioned, the nature of Manim and our way of enabling dynamic operations influences the library's ability to create simultaneous animations within two separate data structure depictions. In the case of a function requiring singular operations, such as a regular *add()* of a resizing array, we technically animate the operation in one depiction and then in the other, pretending it is visually simultaneous.

However, the situation becomes more intricate with a resizing *add()*. If combining the ubiquitous resizing array visualization with a CostGraph or high-level abstraction, the solution is simple:
Play the animation in the CostGraph/High-level abstraction first, as these are quick, and then the full dynamic operation of resizing in the array, still pretending the animations are simultaneous.
This solution is basically optimal, as true simultaneity would simply mean having the quick animation vastly slowed down and occurring in the same time frame as the long, dynamic operation, which seems pointless.

In the case of combining the array visual with memory depiction, achieving the desired effect is more complex. As the array and memory visualization have operations which timing-wise go together, such as the instantiation of the new array, and reading and writing of values during copying, these are displayed along with each other, as described in Section 6.3.6[23].
However, we still have to pretend to have simultaneity. This is caused by the ResizingMem class holding the array and memory depiction as fields, and playing their related operations one after the other. For them to truly be simultaneous, the dual visualization would need to not **have** an array and a memory depiction, but **be** an array and a memory depiction. In this way a single function could, for example, cause the indication of both a slot in the array and a slot in memory simultaneously.

We have omitted implementing this, as it would require the custom classes of HTMem and ResizingMem to not only use the functionality of their parts, but implement it anew. This would have no effect on the use of the library, as the API of the classes would not change, but it would lead to bad coding practices, due to the creation of duplicate code.
In the context of our library being a prototype, we have omitted achieving true simultaneity, and accept the shortcomings of our implementation as a result of the Manim library's underlying way of handling animations.

7.4 User Experience

When discussing the user experience of our product, we may examine two separate users: the seasoned programmer/professor using it to create videos, and the student/learner who views such videos.

From these roles, two separate user experiences emerge, involving the use of the library and the viewing of videos. As the videos are created from the code, code composition and resulting videos are interlinked. This interconnection implies that decisions regarding one aspect can influences the other, and vice versa. Therefore, In the development of the library, both of these experiences informed our design decisions and continually shaped our development process.

[23] A HTMem video

Some decisions made to increase code quality and ease of use had little effect on resulting videos. An example is the use of the facade software pattern, which provides a simplified interface to a collection of classes, which in our case is for the abstract and Von Neumann-based resizing array classes. The facade encapsulates the inner functioning of the chosen class, and allows the user to simply call *add()/remove()* functions. Similarly, the the use of the factory pattern encapsulates array creation logic, promoting best practice, code use and maintainability.

Likewise there are design choices which predominantly affect viewer experience, with minimal impact on the underlying code. Decisions such as which colors to use, where to place old and new array in relation to one another, when to play indication animations, etc., all affect the viewer experience while only changing small bits of code. An example is the turning grey of the old hash table and its corresponding memory slots in our HTMem class, discussed in depth in Section 6.3.7.

Other decisions influence both users. The previously discussed omission of implementing true simultaneity was taken primarily to avoid duplicate code, which bloat the code based and make it harder to maintain, but forces some animations to not be simultaneous. In Future Work we inspect how user testing could inform decisions on these matters.

7.5 Comparative Analysis

The created visualization library, with the intention of furthering learners understanding of data structures, was in part inspired by the visualizations featured in *Algorithms*[41]. It therefore makes sense to compare a selection of the produced visualizations with those of Sedgewick & Wayne. There are several comparisons to make, which highlight strengths, weaknesses, similarities and differences of the two learning tools.

7.5.1 Depiction of Resizing Array Operations

One comparison which emphasizes a central aspect of the created library, is that of the visualization of dynamic operations and the representation of time. As the library creates videos where data structure operations unfold, these of course do so over the course of a period of time. The resizing of an array happens operation after operation, which naturally takes longer the larger the arrays.
As each operation requires some time to complete, the time of a given video corresponds in some way to the computational cost of the operations it depicts. This is not a one-to-one correspondence, as operations are not bound to some operation-time ratio, but instead a communication of the essence of computational cost. *Algorithms* depicts the dynamic unfolding of events in a variety of comparable ways, which are exemplified by Figure 38. The figure depicts a resizing array over the course of a sequence of operations. The unfolding of time happens moving down line to line. Similar to the library's array representation[24], this is a monofunctional, non-discursive representation, with a defined and fixed structure of information.

This visualization, while conveying the state of the array over time, does not, as a static image, convey the passing of time. Apart from the changing of the array, there is no difference between line jumps with and without resizing. As such the depiction does little to convey computational cost. Along with this comes the fact that the mechanics

[24]Resizing Array as a Stack

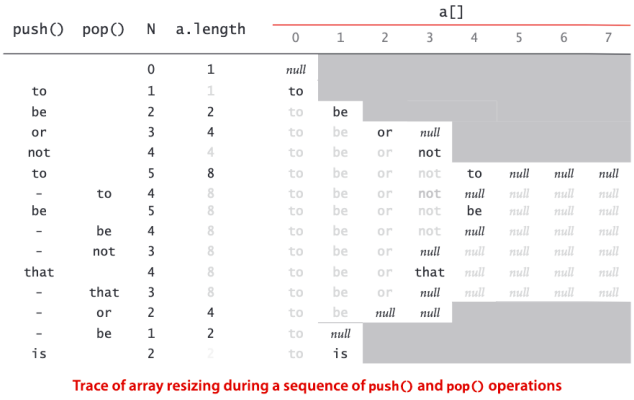


Figure 38: Trace representation of resizing array [41]

of the resizing are hidden. It is not clear from the depiction that a resized array is in fact a new array replacing the old, and instead the viewer is left with the impression that the array simply grows. *Algorithms* of course explains this elsewhere, but fails to convey it visually[41].
It does however have several advantages over the created visualization library. Namely, we are able to see the entire unfolding of events in a single image, allowing for an easy tracing of event. The inclusion of which operations that are called adds clarity to the depiction, helping the reader understand what is happening.

The advantages of both depictions actually complement each other rather well, and a combination, for example a video from our library[25] ending with a corresponding image in the style of Figure 38 might be very helpful to learners. Such a combination would also align with our deployed theory, by both practicing the translation between representations which, Duval promotes, and presenting two separate LoA.

7.5.2 High Abstraction

Inspecting our high abstract visualization of a Bag side-by-side with *Algorithms* in Figures 7 & 39, the inspiration we take from the book is clear. The aim with these visualizations was to emulate *Algorithms* approach as much as possible, to create classes which could operate in conjunction with the other representations, while utilizing their refined visualizations.
The result of any visual comparison between these abstract depictions will inevitably be the need for visual refinement of our library's high abstract depictions, as they need quite some polishing to match those of *Algorithms*. However, we are satisfied with our classes as an initial visuals, despite their shortcomings.

The combination of the library's high abstract depictions with its other depictions are their primary strength, allowing for the creation of videos which align with the deployed theory.
While *Algorithms* different visuals could be viewed side by side, they are mostly conceptually, and not explicitly linked. The appearance of a value within a bag, along with the addition of the same value in an array, is exactly the point of our library, where

[25] Resizing Array as a Stack

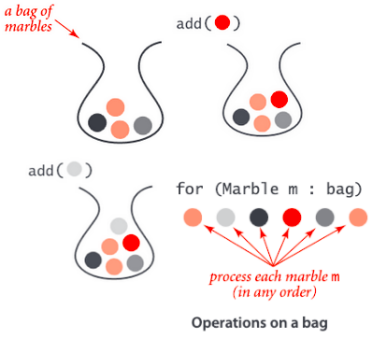


Figure 7: Drawing of a Bag [41]

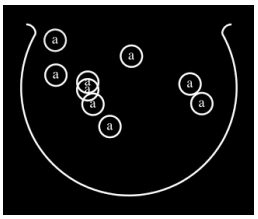


Figure 39: our library's abstract depiction of a Bag.

relations between layers of abstraction are shown to the viewer. These connections justify the development of our high abstract depictions, even if their visuals fall short compared to existing visualizations.

7.6 Future work

In the following section, we delve into future ways to expand the current research and our software product. Some ideas are natural next steps to be taken, while others are considered out of scope for this thesis.

7.6.1 Lack of User Testing

A significant constraint of this thesis and in library development is the limited scope of user testing, with feedback primarily gathered through meetings with our supervisor. Although this provided valuable insights and refinement opportunities, a more robust and systematic approach to user testing could further enhance the solidity and applicability of our findings.

Explicitly defining the distinction between users of our library lays a solid foundation and a need for potential future work to incorporate a more extensive user testing methodology, envisioning a thoughtful evolution, where the distinct needs of each user type remain at the core of development for a continuous product improvement.

7.6.2 Hash Table Memory Upgrade

Among the different visualizations produced, we identified one that could benefit from further development: the Hash Table Memory.
As the animation displays both a hash table and computer memory on the screen, it may be beneficial for the viewer to establish a stronger relation between the two abstractions by showing the connection of memory slots, corresponding to the linked lists' nodes of the hash table.

To prevent visual clutter, this connection could be briefly display on the screen, fading out after a few seconds. Furthermore, this logic can be encapsulated in a function that users can call at a given time. This would allow users, for instance, to visually

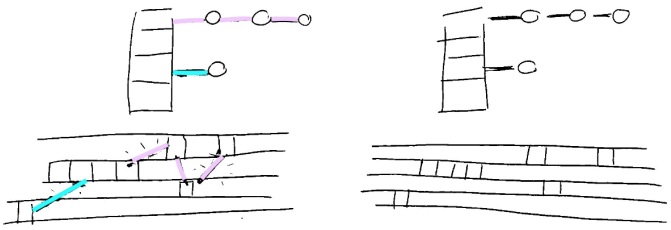


Figure 40: Sketch of upgraded Hash Table Memory

verify the positioning of nodes in memory.
The sketched drawing in Figure 40 illustrates the concept described above. Progressing from left to right, the initial frame highlights the linked slots, and the right frame displays the faded-out links in memory.

7.6.3 Descriptive Text

Another element that may enhance the understanding of our visuals is to include text complementing our visual animations.
Throughout our library's development, we explored various types of text for this purpose and prototyped different animations with it.
Examples include a legend that explains the state of slots in computer memory (e.g. GREEN for Available, RED Occupied and BLUE for Reserved as shown in Figure 41) or text displaying the cost of each operation for the data structure.

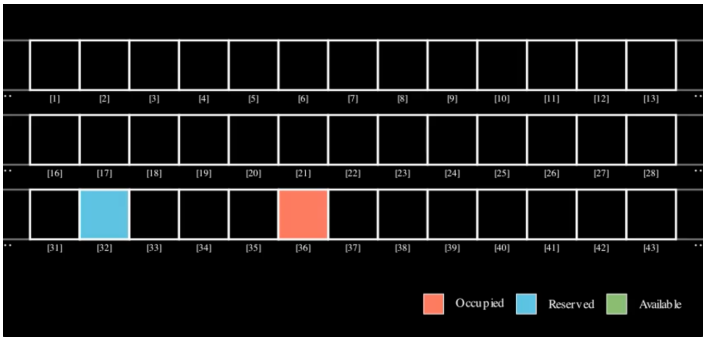


Figure 41: Prototype of legend in Memory

Despite implementing these texts in initial prototypes, we have yet to find a suitable implementation which we feel confident does not overwhelm the viewer's visual impression. Future user-testing efforts may provide valuable insights in identifying an optimal solution for enhancing the viewer's experience.

7.6.4 Further Inspiration from *Algorithms*

Algorithms includes visualizations which could inspire further development within the created library. The textbook naturally explores a plethora of data structures and algorithms, many of which could be well suited for depiction in a dynamic, video format.

Union-find and Quick-union data structures are an example of this. The visualization are quite complete, and it is hard to imagine how they could be improved upon in isolation. However having them represented in our library in combination with out memory visualization may serve to offer new perspectives on their functioning for students and learners.

A visualization which would lend itself well to a more or less one-to-one copy is *Algorithms*'s depiction of a priority queue in both array and graph representation. Figure 42 shows how the priority queue is structured within an array, what it looks like as a graph, as well as an in between representation. Yet again *Algorithms* aligns itself with our deployed theory by presenting different representations of the same concept, along with a third representation for the translation between them. The current priority queue class within the created visualization library is limited to the array representation, but here shows the dynamic nature of operations. The addition of a graph representation performing the same operations is a good candidate idea for expansion of our library.

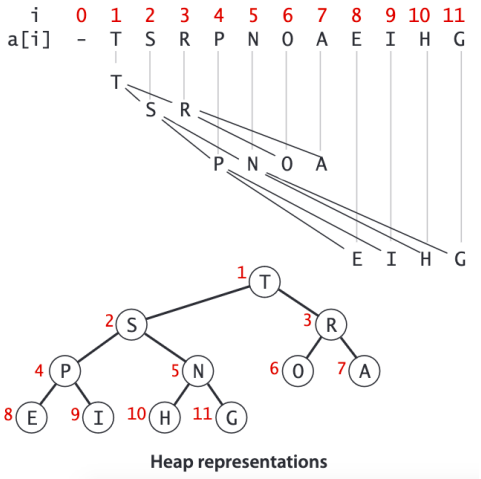


Figure 42: A priority queue in two representations[41].

Other data structures of interests for depiction are binary search trees, tries, various forms of graphs, etc. All of these are depicted in *Algorithms*, and could be developed with our visualization library.

7.7 Acknowledging Limitations

In acknowledging the achievements of this thesis, it is essential to recognize certain limitations that warrant consideration. Firstly, the absence of user testing poses a constraint on the definitive assessment of the software library's efficacy.
While the theoretical underpinnings supporting the development are robust, the lack of direct user feedback raises questions about the practical usability and refinement opportunities that could enhance its effectiveness. Furthermore, the need for empirical evidence to validate the effectiveness of mobilizing our synthesis of theories remains a pertinent concern. Quantitative studies into both the validity of our synthesis and the effects of our library, would serve to solidify their reliability. This is however out of scope of this project.

These acknowledged limitations serve as a starting points for future investigations, urging a more comprehensive exploration of user experiences and the efficacy of our synthesized theories within the field of didactic research.

8 Conclusion

This thesis has delved into the need for visualizations of data structures, along with an overview of existing examples in the field.
The contributions of this work are as such two part: *1.* The synthesis of a collection of different theories regarding the nature of understanding abstract concepts, and *2.* the production of a visualization library based on this synthesis.

By combining Hermans thoughts on understanding and misconception, which are explicitly about the programming, with the more general formulations of Floridi, we successfully combine complementary theories from within the discipline of computer science. Adding Duval's perspective from mathematical didactics on understanding non-empirically accessible knowledge objects, creates a unique and new theoretical perspective on the nature of understanding.
Examining these theories and evaluating the utilization of visual representations of data structures in current didactic materials, we affirmatively addressed Research Questions 1, 2, and 3:

1. *Are visualizations of data structures in teaching material a tool used to promote their understanding?*
2. *Can visualizations of data structures lead to misconceptions regarding their functioning?*
3. *Does a combination of visual representations help prevent such misconceptions?*

Deploying our novel synthesis in the development process of our visualization library lead us to a software product with merits which we are thoroughly confident in.
Implicitly, this led us to answer affirmatively to the Research Question 4:

4. *Can a synthesis of existing theory guide the creation of a software product facilitating the combination of visual representations?*

The implications of our work similarly relates to the theoretical synthesis and the produced software. Our theoretical synthesis and its insights may be applied widely

within computer science education. The perspective on how different representations further understanding, and the framework with which to discuss such representations, allows for further informed reflections on teaching practices. For example, individuals teaching computer science concepts may use our synthesis to evaluate their own or others' teaching and/or teaching materials, inspecting them for a varied approach to the representation of abstract concepts.

The visualization library is expressly created as a tool for understanding, meant to be used by educators to create teaching materials. It is our hope that the library can be used as such, as its deployment would naturally result in teaching aligning with the deployed theory.

In exploring future directions for our research and software product, we have identified key areas for potential development of the product which may further enhance its usefulness. Such further work would not only refine our product but also to underscore its impact on enhancing the understanding of fundamental data structures.

References

[1] Csvistool. `https://csvistool.com/`, 2023. Accessed on: Sep. 2023.

[2] Visualgo. `https://visualgo.net/en`, 2023. Accessed on: Sep 2023.

[3] Edward E Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L Su, and Samuel Z Guyer. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th international symposium on Software visualization*, pages 53–62, 2010.

[4] Kent Beck. Using pattern languages for object-oriented programs. In *OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming*, 1987.

[5] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Agile manifesto. `https://agilemanifesto.org/`. Accessed on: Dec. 2023.

[6] C. Brabrand and S. M. Nicolajsen. Forelæsning 01b: Sekventielle konstruktioner, 2022. [The figure is a translated version of a slide from Brabrand and Nicolajsens lecture.].

[7] J. Bruner. *The Process of Education*. Harvard University Press, revised edition. edition, 1977.

[8] A. Capiluppi, M. Morisio, and J.F. Ramil. The evolution of source folder structure in actively evolved open source systems. In *10th International Symposium on Software Metrics, 2004. Proceedings.*, pages 2–13, 2004.

[9] Tao Chen and Tarek Sobh. A tool for data structure visualization and user-defined algorithm animation. In *31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No. 01CH37193)*, volume 1, pages TID–2. IEEE, 2001.

[10] Ken Clemens. Visual imagery and school mathematics. *for the learning of mathematics*, 1981.

[11] Timothy Colburn and Gary Shute. Abstraction in computer science. *Minds and Machines*, 17:169–184, 08 2007.

[12] Manim Community. Quickstart. `https://docs.manim.community`. Accessed on: Nov. 2023.

[13] James O Coplien. Software design patterns. In *Encyclopedia of Computer Science*, pages 1604–1606. 2003.

[14] T. H. Cormen, C. E. Leierson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 4th edition, 2022.

[15] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[16] R. Duval. *Understanding the Mathematical Way of Thinking – The Registers of Semiotic Representations*. Springer Cham, 1 edition, 2017.

[17] Ndudi Ezeamuzie, Jessica Leung, and Fridolin Ting. Unleashing the potential of abstraction from cloud of computational thinking: A systematic review of literature. *Journal of Educational Computing Research*, 12 2021.

[18] Luciano Floridi and J. W. Sanders. Levellism and the method of abstraction. In *IEG Research Report*. 2004.

[19] Gottlob Frege and EW Kluge. Review of dr. e. husserl's philosophy of arithmetic. *Mind*, 81(323):321–337, 1972.

[20] D. Fussel. The von neumann model. *The University of Texas in Austin - cs310h Course, Lecture 9*, 2010.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[22] Marc Gregoire. History and classification of patterns. `https://www.modernescpp.com/index.php/history-and-classification-of-patterns/`, 2019. Accessed on: Oct. 2023.

[23] F. Hermans. *The Programmer's Brain*. Manning Publications, 2021.

[24] Philip Nicholas Johnson-Laird. *Mental models: Towards a cognitive science of language, inference, and consciousness*. Number 6. Harvard University Press, 1983.

[25] Ville Karavirta and Clifford A Shaffer. Jsav: the javascript algorithm visualization library. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 159–164, 2013.

[26] Herman Koppelman and Betsy Van Dijk. Teaching abstraction in introductory courses. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, pages 174–178, 2010.

[27] Jeff Kramer. Is abstraction the key to computing? *Commun. ACM*, 50:36–42, 04 2007.

[28] Georg Kreisel. *Wittgenstein's Remarks on the Foundations of Mathematics*. JSTOR, 1958.

[29] Dapeng Liu, Zengdi Cui, Shaochun Xu, and Huafu Liu. An empirical study on the performance of hash table. In *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, pages 477–484, 2014.

[30] Yann Lucet. University of british columbia - data structures and algorithms. `https://cmps-people.ok.ubc.ca/ylucet/DS/Algorithms.html`, 2023. Accessed on: Sep 2023.

[31] C Murphy. Exploring the role of visual imagery in learning mathematics. 2019.

[32] Javin Paul. Design patterns 101: Factory vs builder vs fluent builder. https://medium.com/javarevisited/design-patterns-101-factory-vs-builder-vs-fluent-builder-da2babf42113, 2021. Accessed on: Oct. 2023.

[33] Mathilde Pedersen, Cecilie Bach, Rikke Maagaard Gregersen, Ingi Højsted, and Uffe Jankvist. Mathematical representation competency in relation to use of digital technology and task design-a literature review. *Mathematics*, 9, 02 2021.

[34] Jacob Perrenet, Jan Friso Groote, and Eric Kaasenbrood. Exploring students' understanding of the concept of algorithm: Levels of abstraction. *SIGCSE Bull.*, 37(3):64–68, jun 2005.

[35] Rafael Rangel, Lourdes Guerrero, Ricardo Ulloa-Azpeitia, and Elena Nesterova. Mathematical modeling in problem situations of daily life. *Journal of Education and Human Development*, 5, 01 2016.

[36] Reducible. Reducible youtube channel. https://www.youtube.com/@Reducible, 2023. Accessed on: Sep. 2023.

[37] Grant Sanderson. 3blue1brown youtube channel. https://www.youtube.com/@3blue1brown, 2023. Accessed on: Sep. 2023.

[38] F. Saussure, C. Bally, and A. Sechehaye. *Course in General Linguistics*. 1916.

[39] Ken Schwaber and Jeff Sutherland. The scrum guide. *Scrum Alliance*, 21(1):1–38, 2011.

[40] Johannes Schwartzkopff. Examining and developing visual learning tools for resizing arrays. 2023. Contact at rosc@itu.dk for the paper.

[41] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 4th edition, 2011.

[42] Edward Tufte. The work of edward tufte and graphics press. https://www.edwardtufte.com/tufte/. Accessed on: Oct. 2023.

[43] Alfredo Vellido, José Martín-Guerrero, Fabrice Rossi, and Paulo Lisboa. Seeing is believing: The importance of visualization in real-world machine learning applications. 01 2011.

[44] D. J. Wood, J. S. Bruner, and G. Ross. The role of tutoring in problem solving. *Journal of Child Psychiatry and Psychology, 17(2)*, pages 89–100, 1976.

[45] Yiran Zhang, Zhengzi Xu, Chengwei Liu, Hongxu Chen, Jianwen Sun, Dong Qiu, and Yang Liu. Software architecture recovery with information fusion. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1535–1547, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Daniel Zingaro, Cynthia Taylor, Leo Porter, Michael Clancy, Cynthia Lee, Soohyun Nam Liao, and Kevin C Webb. Identifying student difficulties with basic data structures. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 169–177, 2018.