CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Visualisering av datastrukturer och algoritmer:

Utveckling av ett visualiseringsbibliotek till stöd för studenters inlärning

Kandidatarbete i data- och informationsteknik

Sondre Fransson
Simon Grandin
Noel Heimer
Aron Sigfridsson
Shawn Wang
Erik Zetterlund

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Kandidatarbete 2026

Visualisering av datastrukturer och algoritmer:

Utveckling av ett visualiseringsbibliotek till stöd för studenters inlärning

Sondre Fransson
Simon Grandin
Noel Heimer
Aron Sigfridsson
Shawn Wang
Erik Zetterlund

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Gothenburg, Sweden 2026

Visualisering av datastrukturer och algoritmer:
Utveckling av ett visualiseringsbibliotek till stöd för studenters inlärning
Sondre Fransson Simon Grandin Noel Heimer Aron Sigfridsson Shawn Wang Erik Zetterlund

Supervisor (Handledare): Peter Ljunglöf, Department of Computer Science and Engineering
Examiners: Patrik Jansson, Department of Computer Science and Engineering
Graded by teacher (Rättande lärare): Birgit Grohe, Department of Computer Science and Engineering

Kandidatarbete 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Visualisering av datastrukturer och algoritmer:
Utveckling av ett visualiseringsbibliotek till stöd för studenters inlärning
Sondre Fransson, Simon Grandin, Noel Heimer, Aron Sigfridsson, Shawn Wang, Erik Zetterlund

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Sammandrag

Datastrukturer och algoritmer utgör grunden för de allra flesta programvaror, men att lära sig hur de är uppbyggda och koncepten de bygger på kan ofta vara svårt. Detta arbete presenterar ett bibliotek av visualiseringar som kan stödja inlärningen av dessa koncept. Arbetet genomfördes genom att vidareutveckla ett befintligt bibliotek av visualiseringar som skapats för en introduktionskurs till datastrukturer och algoritmer som ges vid Chalmers tekniska högskola och Göteborgs universitet. Biblioteket utökades med nya datastrukturer och algoritmer i form av köer, dynamiska arrayer, hashtabeller, sorteringsalgoritmer och grafer. Även användargränssnittet förbättrades för att ge en bättre och tydligare användarupplevelse och för att stödja användare med olika typer av färgblindhet. Resultaten av användartester visade tendenser på att det utvecklade biblioteket upplevdes uppfylla sitt syfte. Slutligen framställs möjliga förbättringsområden och framtida problemställningar.

Abstract

Data structures and algorithms form the foundation for most software, but understanding how they work and the concepts they build on can often be difficult. This project presents a library of visualizations that can support in learning these concepts. The work was carried out by extending an existing library of visualizations made for an introductory course in data structures and algorithms given at Chalmers University of Technology and the University of Gothenburg. The library was expanded with new data structures and algorithms such as queues, dynamic arrays, hash tables, sorting algorithms, and graphs. The user interface was also improved to provide a better and clearer user experience and to support users with different types of color blindness. The results of user tests showed tendencies that the enhanced library was perceived to fulfill its purpose. Finally, opportunities for further improvements and future areas of research are presented.

Nyckelord: algoritmer, datastrukturer, visualisering, inlärning

Förord

Vi vill tacka vår handledare Peter Ljunglöf för all stöttning och hjälp under projektets gång.

Sondre Fransson, Simon Grandin, Noel Heimer, Aron Sigfridsson, Shawn Wang, Erik Zetterlund, Göteborg, juni 2026

Innehåll

Figurer

Tabeller

1

Introduktion

Grunden för en programvara utgörs i de flesta fall av datastrukturer och algoritmer, och de är nödvändiga komponenter för att skapa effektiva och korrekta program [1]. Tiden det tar att utföra en uppgift kan skilja sig drastiskt åt beroende på val av underliggande datastrukturer och algoritmer, alltifrån ett par sekunder till flera dagar eller till och med år. Det kan utgöra den enda skillnaden mellan en programvara som anses riktigt bra, och en som anses vara oanvändbar. Vid programvaruutveckling är det därför av högsta vikt att känna till olika datastrukturer och algoritmer, veta hur de fungerar och i vilka fall man ska använda dem.

Att förstå uppbyggnaden av dessa datastrukturer och algoritmer, samt de koncept de bygger på, är dock inte alltid trivialt. Ett exempel på ett sådant koncept är balanseringen av ett *AVL-träd*[1], som innefattar olika typer av rotationer av trädets noder. Det är för många svårt att ta till sig endast utifrån teoretiska beskrivningar och statiska illustrationer, vilket kräver ytterligare verktyg för att öka förståelsen. Studier har pekat på att interaktiva visualiseringar kan vara ett bra alternativ som ett sådant verktyg [2].

På Chalmers tekniska högskola och Göteborgs universitet ges kursen *Datastrukturer och algoritmer*, vars syfte är att ge studenterna den grund inom området som krävs för vidare studier och arbete inom programvaruutveckling av olika slag. För att underlätta inlärningsprocessen för studenterna har ett bibliotek för att visualisera datastrukturer och algoritmer utvecklats av Peter Ljunglöf, en av lärarna på kursen [3]. Biblioteket är en omarbetning av ett tidigare bibliotek av David Galles, University of San Fransisco [4]. Omarbetningen gjordes för att kodbasen skulle vara lättare att förstå och underhålla, samt förbättra visualiseringarna. Det ursprungliga biblioteket innehåller dock betydligt fler algoritmer och datastrukturer, och många av dem är en väsentlig del av undervisningen i kursen på Chalmers och Göteborgs universitet.

Utöver Ljunglöfs egna arbete utfördes även ett tidigare kandidatarbete med syftet att omvandla bibliotekets kodbas från JavaScript till TypeScript [5]. Detta gjordes för att skapa bättre struktur, minska förekomsten av fel i koden och förenkla vidareutveckling. Det tidigare kandidatarbetet innebar även tillägg av prototyper av ett antal sorteringsalgoritmer som ingår i den kurs biblioteket utvecklats för.

[1] Förkortningen kommer från Adelson-Velsky och Landis, uppfinnarna av AVL-trädet

1.1 Syfte

Projektets syfte är att utveckla interaktiva visualiseringar som ska underlätta studenters inlärning av olika koncept inom kursen *Datastrukturer och algoritmer*, som ges av Chalmers tekniska högskola och Göteborgs universitet [1]. Projektet utgår från ett befintligt bibliotek [3], och fokuserar främst på hur både nya visualiseringar ska utformas och existerande visualiseringar utvecklas utifrån pedagogisk tydlighet, samtidigt som de ska ge en korrekt representation av de bakomliggande strukturerna och algoritmerna samt överensstämma med övrigt kursinnehåll. Utöver detta syftar projektet även till att utveckla webbapplikationen där visualiseringarna presenteras, så att den blir mer användarvänlig och tillgänglig för alla användare.

1.2 Avgränsningar

Inom ramen för detta projekt har det inte funnits utrymme att behandla allt innehåll i kursen som projektet har utgått ifrån [1]. Av de datastrukturer och algoritmer som redan fanns implementerade i biblioteket har därför ett antal valts ut som bedömdes vara i störst behov av utveckling. De som valts ut är de fyra sorteringsalgoritmerna *insertion sort*, *selection sort*, *quicksort* och *merge sort*, samt datastrukturen *länkad lista*.

Utöver dessa har även några nya visualiseringar för andra strukturer och algoritmer skapats, däribland köer, stackar, dynamisk arrayer och hashtabeller, som är några av de mest centrala strukturerna i kursen. Ytterligare har de tre sorteringsalgoritmerna *bubble sort*, *radix sort* och *heap sort* lagts till. Till sist har också visualiseringar för grafer och några av de vanligaste grafalgoritmerna utformats.

När det kommer till att visualisera grafer av okänd storlek, antal noder och kanter, så är det ett väldigt svårt programmeringsproblem [6], [7]. En bra visualisering av en graf innebär bland annat att noderna positioneras på ett sådant sätt att så få kanter som möjligt korsar varandra. Att lösa detta problem ligger utanför ramen för projektet, och därför gjordes avgränsningen att endast utveckla några fördefinierade grafer, istället för att låta användaren skapa en egen graf.

2

Utgångspunkten för projektet

Som tidigare nämnts skrevs biblioteket ursprungligen av Peter Ljunglöf, då i JavaScript, som en omarbetning av David Galles bibliotek [4]. Ljunglöfs bibliotek omarbetades sedan till TypeScript och utvecklades vidare genom ett tidigare kandidatarbete [5], och resultatet av det arbetet var utgångspunkten för detta arbete. I detta kapitel beskrivs den övergripande uppbyggnaden och innehållet i den version av biblioteket som detta projekt har utgått ifrån. Först beskrivs hur biblioteket ser ut och fungerar utifrån användarens perspektiv, sedan den bakomliggande kodstrukturen.

2.1 Bibliotekets funktionalitet

Biblioteket är en webbapplikation skapad med hjälp av HTML,[1] CSS,[2] och TypeScript (vilket översätts till JavaScript) som helt och hållet körs i användarens webbläsare. Applikationen består av två typer av sidor, dels startsidan, dels sidor som innehåller de olika visualiseringarna. Funktionaliteten och uppbyggnaden för dessa två typer av sidor beskrivs i följande avsnitt.

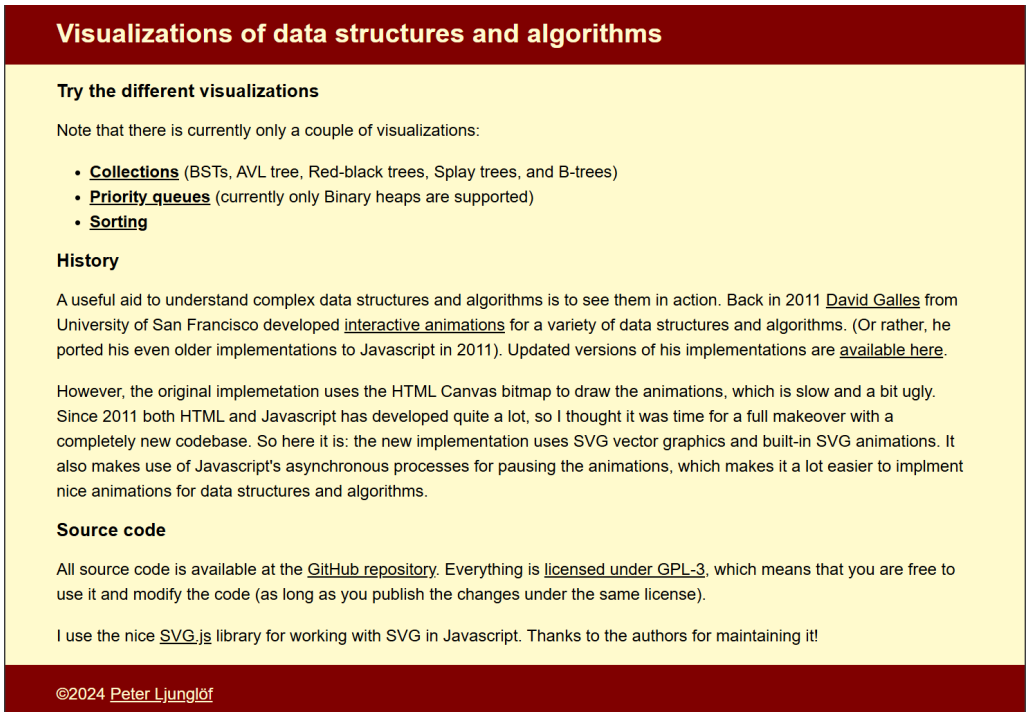
2.1.1 Startsidan

På startsidan presenteras generell information, en kort bakgrund till biblioteket och en länk till källkoden. Den innehåller också länkar till de separata sidorna för olika kategorier av visualiseringar. Det finns tre kategorier av visualiseringar, *Collections*, *Priority queues* och *Sorting*. *Collections* innehåller visualiseringar för olika samlingar av värden i form av några typer av träd och länkad lista. *Priority queues* innehåller visualisering av en prioritetskö med binär heap, och under *Sorting* finns visualiseringar för de fyra sorteringsalgoritmerna *insertion sort*, *selection sort*, *merge sort* och *quicksort*. Hur startsidan ser ut visas i figur 2.1.

2.1.2 Sidor för visualiseringar

Varje kategori av visualiseringar består av en egen sida i applikationen. Sidorna för de olika kategorierna är uppbyggda på samma sätt, med samma grundläggan-

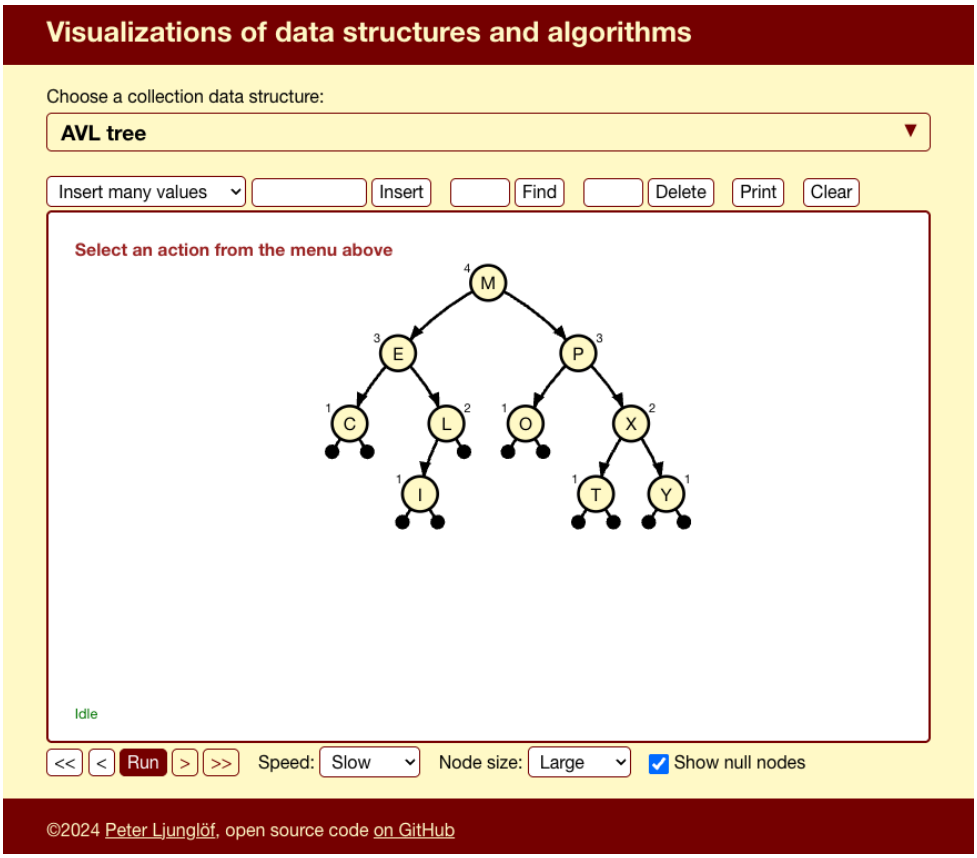
[1] Hypertext Markup Language: https://developer.mozilla.org/en-US/docs/Web/HTML
[2] Cascading Style Sheets: https://developer.mozilla.org/en-US/docs/Web/CSS



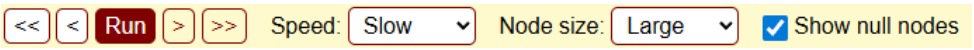
Figur 2.1: Figuren visar startsidan för den version av biblioteket som projektet har utgått ifrån. Genvägarna till visualiseringarna presenteras genom fetmarkerade länkar i punktlistan.

de gränssnitt, för att göra applikationen så enhetlig som möjligt. På varje sida får användaren välja en av de olika datastrukturerna och algoritmerna som finns tillgängliga inom den kategorin, genom att markera ett alternativ i en rullgardinsmeny överst på sidan.

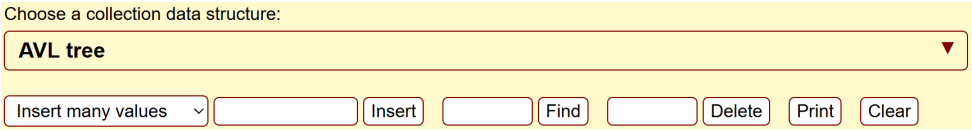
Den största delen av sidan tas upp av ett fönster där visualiseringarna visas. I fönstret ritas två typer av information ut, dels de objekt som bygger upp visualiseringarna, dels ett antal informationstexter som talar om vad som händer i visualiseringen. Under fönstret finns en uppsättning generella kontroller, det vill säga kontroller som är gemensamma för alla visualiseringar, med några undantag. Dessa kontroller består av knappar för att styra hur animationen spelas upp (vilket beskrivs närmare i avsnitt 2.1.3), och rullgardinsmenyer för att välja animationshastighet och storlek på objekten som ritas ut. Utöver de generella kontrollerna finns även specifika kontroller ovanför fönstret som beror på vilken kategori av visualiseringar som valts. I kategorin *Collections* består de specifika kontrollerna exempelvis av knappar och inmatningsfält för att hitta, lägga till eller ta bort ett värde i samlingen. Figur 2.2 visar hur sidan ser ut för kategorin *Collections* med AVL-träd som vald struktur.



(a) Sidan för visualiseringar inom kategorin *Collections*, här med AVL-träd som den valda datastrukturen. I fönstret i mitten syns de objekt som bygger upp visualiseringen, samt två informationstexter i övre och nedre vänstra hörnet.



(b) Den undre knappraden med de generella kontrollerna för att stega igenom animationen, samt välja animationshastighet och storlek på objekten som ritas ut. En extra kryssruta för att bestämma om tomma barn ska visas eller inte har lagts till på sidorna för trädvisualiseringar.



(c) Rullgardinsmenyn för att välja datastruktur eller algoritm inom den valda kategorin tillsammans med den övre knappraden med de specifika kontrollerna, här för ett AVL-träd. Knappen ”Clear” är gemensam för alla visualiseringar, men har ändå placerats bland de specifika kontrollerna och är därmed ett undantag.

Figur 2.2: Figurerna visar hur sidan med visualisering av AVL-träd ser ut i den version av biblioteket som projektet har utgått ifrån. **(a)** ger en överblick av hela sidan, **(b)** visar den undre knappraden med de generella kontrollerna och **(c)** visar den övre knappraden med de specifika kontrollerna för AVL-träd, tillsammans med rullgardinsmenyn som används för att välja datastruktur eller algoritm.

2.1.3 Uppspelningslägen för visualiseringarna

I applikationen finns tre möjliga uppspelningslägen för visualiseringarna. Det första läget är att de utförs stegvis, alltså att applikationen stannar vid varje steg i visualiseringen tills användaren väljer att stega vidare till nästa. Det andra läget ger möjligheten att köra visualiseringen utan pauser. Det innebär att användaren kan starta visualiseringen, och sedan låta applikationen automatiskt påbörja nästa steg när den blir färdig med det nuvarande steget. Till sist kan användaren snabbspola visualiseringen, alltså genomföra alla steg utan att visa animationerna, så att slutresultatet presenteras direkt. Möjligheten att stega och snabbspola både framåt och bakåt syns i figur 2.2b ovan, och funktionaliteten beskrivs vidare i avsnitt 2.2.3.

2.2 Bakomliggande kodstruktur

Biblioteket är uppbyggt av ett antal viktiga komponenter. Den mest centrala komponenten är klassen `Engine` som är grunden och motorn för alla animationer, och den innehåller flera centrala metoder. En av dem är `execute` som ansvarar för att initiera varje ny operation som ska visualiseras, exempelvis lägga till eller söka upp värden i ett AVL-träd. Metoden sparar dessutom operationen i en historik. Därefter anropar `execute` metoden `runActionsLoop`, en annan viktig metod i `Engine`. Den utför alla operationer i historiken i ordning, från den första till den sista. Visualiseringen av alla operationer är uppdelade i flera steg, och `pause`, som också finns i `Engine`, är den metod som används för att definiera de olika stegen. Hur de olika delarna hänger ihop och fungerar samt några viktiga tekniker som används förklaras i följande avsnitt.

2.2.1 Huvudsakliga tekniker

Biblioteket bygger huvudsakligen på två tekniker, varav den första är SVG.[3] Det är ett språk som används för att beskriva vektorgrafik i två dimensioner och lätt kan integreras med HTML. Vektorgrafiken gör att en hög kvalitet kan upprätthållas för skärmar av alla olika storlekar. Detta gör att SVG lämpar sig för ett webbaserat animeringsprojekt likt detta. I detta projekt används biblioteket SVG.js, som underlättar implementeringen av några av de vanligaste funktionerna i SVG, exempelvis att rita ut olika former och animera förflyttningar [8].

Den andra tekniken projektet bygger på är asynkrona funktioner. Dessa funktioner möjliggör att olika delar av programmet kan exekveras mer eller mindre parallellt. De asynkrona funktionerna returnerar något som kallas *Promise*, löfte, vilket är ett objekt som innehåller information om tillståndet för det asynkrona anropet. Löftet kan avslutas i två olika tillstånd, *fulfilled* (uppfyllt) eller *rejected* (avvisat). En funktion som anropar en asynkron funktion kan välja att pausa exekveringen tills löftet har blivit avslutat, vilket sker då någon del av programmet anropar en funktion som antingen avvisar eller uppfyller löftet. Denna möjlighet utnyttjas i

[3]Scalable Vector Graphics: https://developer.mozilla.org/en-US/docs/Web/SVG

biblioteket när användaren ska stega igenom en animering steg för steg, något som beskrivs mer i detalj i avsnitt 2.2.3.

2.2.2 Utformning av visualiseringar

Alla visualiseringar representeras i koden i form av var sin asynkron funktion. Funktionen definierar alla animationer och uträkningar som behövs för att presentera datastrukturen eller algoritmen, och visualiseringarna är uppdelade i flera steg för att användaren lättare ska förstå vad som händer. För att skapa ett steg i en visualisering utformas metoden som representerar visualiseringen på följande sätt; först genomförs alla beräkningar och animationer som hör till det aktuella steget, och sedan anropas metoden `pause`. Syftet med `pause` beskrivs i avsnitt 2.2.3. Hela visualiseringen byggs således ihop av flera steg genom att upprepa olika beräkningar och animationer följt av anrop till `pause`, ända tills visualiseringen är klar.

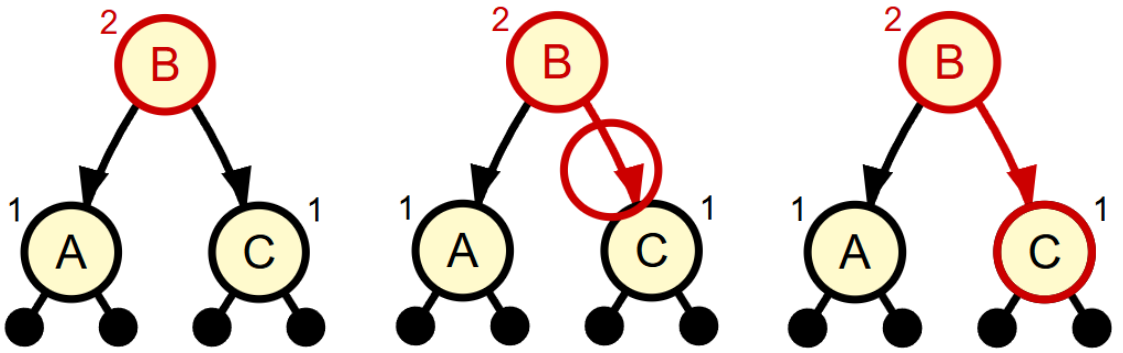
För att exemplifiera detta beskrivs här hur ett steg utformas vid sökning efter ett värde i ett AVL-träd. Sökningen börjar i den nod som är roten av trädet. Förutsatt att värdet i roten inte är det värde som eftersöks är nästa steg i visualiseringen att fortsätta söka antingen i det vänstra eller högra barnet. För att göra detta krävs ett antal beräkningar, dels att bestämma om det eftersökta värdet är större eller mindre än rotens värde, dels att hämta koordinaterna för det barn som ska undersökas. Utöver beräkningarna animeras steget också genom att markera den nuvarande noden samt kopplingen till barnet som ska undersökas, och dessutom flyttas en cirkel som indikerar vilken väg algoritmen tar för att söka igenom trädet. Alla dessa beräkningar och animationer utgör ett steg i visualiseringen, och följs därefter av ett anrop till `pause` för att markera gränsen innan nästa steg. Pseudokoden för detta steg ses i figur 2.3 nedan, och animeringssekvensen ses i figur 2.4.

```
def find_value(val, node):
    ...
    pause() // Previous step ends here

    highlight(node)
    child = left_child if val < node.val else right_child
    child_coords = get_coordinates(child)
    highlight(child)
    highlight(node_to_child_connection)
    highlight_circle.move_to(child_coords)
    pause() // Current step ends here

    ... // Following steps
```

Figur 2.3: Pseudokod för att definiera ett av stegen i visualiseringen av att hitta ett värde i ett AVL-träd. Steget som beskrivs är att algoritmen ska fortsätta leta efter ett värde `val` i ett av barnen till `node`.



(a) Visualiseringen innan animationen påbörjas. Värdet som eftersöks är ”C”.

(b) Mitt under animationen. Kopplingen till högerbarnet markeras med rött, och en röd cirkel flyttas från roten till barnet för att visa hur algoritmen vandrar genom trädet.

(c) Efter animationen är färdig. Den förra noden, alltså roten, den nya noden som undersöks samt vägen mellan dem markeras med rött.

Figur 2.4: Figurerna visar hur ett animationssteg i visualiseringen av att hitta ett värde i ett AVL-träd ser ut. Steget som visas är att algoritmen ska fortsätta leta efter värdet ”C” i ett av barnen till trädets rot.

2.2.3 Implementering av uppspelningslägen

För att skapa de olika uppspelningslägena utnyttjas metoden `pause`, så att visualiseringarna kan delas upp i flera steg. Metoden bygger på konceptet av asynkrona funktioner för att stödja de olika uppspelningslägena. Som nämndes i avsnittet om asynkrona funktioner (se 2.2.1) möjliggör sådana funktioner att exekveringen av en funktion pausas tills löftet som returneras av den asynkrona funktionen har avslutats, och detta används i programmet vid utformningen av `pause`. Metoden returnerar alltid ett löfte, vilket gör att den yttre metoden som definierar visualiseringen kan pausa exekveringen tills löftet avslutats. Det innebär att alla beräkningar och animationer som utgör ett steg i visualiseringen kan utföras innan exekveringen pausas vid anropet till `pause`. När användaren klickar på knappen för att stega framåt anropas funktionen som uppfyller löftet, och på så vis kan exekveringen av visualiseringen fortsätta innan den återigen stannar på liknande vis vid det påföljande stegets anrop till `pause`-metoden. Om användaren istället valt att automatiskt stega igenom hela visualiseringen ser `pause` till att löftet uppfylls så fort hela det aktuella steget är färdigt, utan att användaren behöver göra något.

För att tillåta användaren att stega bakåt utnyttjas istället det andra avslutande tillståndet för ett löfte, att det blir avvisat, samt det faktum att det finns möjlighet att skicka tillbaka information till funktionen som väntar på löftet. I de fall då löftet som returneras från `pause`-metoden avvisas betyder det att användaren valt att antingen stega bakåt ett steg, eller snabbspola till början av den nuvarande operationen. Informationen som skickas med löftet består av vilket steg i operationen applikationen ska hoppa tillbaka till, som alltså antingen är ett steg innan det

nuvarande, eller det första steget i operationen. Att stega bakåt i en visualisering är dock inte trivialt, då det exempelvis kräver att man kan beräkna den föregående positionen för alla objekt som ingår i visualiseringen. För att slippa detta implementeras bakåtstegning istället genom att börja om från början och stega fram till rätt steg. Det innebär alltså att när applikationen stegar bakåt så återställs först hela visualiseringen, och därefter utförs alla stegen en gång till från början, ända fram tills det steg användaren ville stega bak till. För att kunna göra det krävs att alla operationer sparas i en historik, som beskrivs närmre i nästa avsnitt.

2.2.4 Historikens uppbyggnad

Historiken består av en lista av objekt, där varje objekt motsvarar en operation som användaren utfört. Objekten innehåller en metod, eventuella argument till metoden och en räknare i form av ett heltal. Metoden i objekten är den metod som definierar visualiseringen av operationen, så som den beskrivs i 2.2.2. Exempel på sådana operationer är insättning eller borttagning av ett eller flera värden i ett AVL-träd. Eftersom visualiseringarna av operationerna är uppdelade i flera steg används räknaren i objektet för att hålla koll på hur många steg som utförts i den aktuella visualiseringen. För de operationer som inte påbörjats är räknaren noll, och för operationer som redan är klara är räknaren samma som antalet steg i operationen. På så vis kan applikationen veta hur långt den kommit igenom alla operationer.

3

Metod

Under genomförandet av projektet användes ett antal program och processer för att stödja utvecklingen. Som nämnts tidigare var projektet en vidareutveckling av ett befintligt bibliotek, och det hade relativt god möjlighet till separation av arbetet som skulle utföras. Därav skapades tre grupper om två personer i varje, där varje grupp huvudsakligen skulle ägna sig åt visualiseringar inom var sitt område. Grupp 1 arbetade med att utveckla de befintliga sorteringsalgoritmerna och lägga till några nya, samt förbättra användargränssnittet. Grupp 2 arbetade med grafer och grafalgoritmer, och grupp 3 med grundläggande datastrukturer som länkade listor och hashtabeller. För att underlätta både versionshantering och parallellt arbete användes Git [9] och GitHub [10]. GitHub var ett naturligt val då det var samma system som användes för det ursprungliga biblioteket. Dessutom användes ramverket Cypress [11] för implementering av automatiserad testning, och Webpack [12] för att underlätta utvecklingen.

För att utvärdera hur väl biblioteket underlättar för studenters inlärning utfördes ett antal användartester. Testerna bestod under projektets gång huvudsakligen av att den ansvariga utvecklaren, övriga gruppmedlemmar och handledaren testade de olika delarna av biblioteket. I projektets slutfas genomfördes även några enstaka utvärderande tester med utomstående personer. Testpersonerna bestod av studenter inom ett tekniskt program, varav vissa hade tagit minst en kurs i datastrukturer och algoritmer, och andra hade inte tagit någon kurs inom ämnet. Det var två typer av utvärderande tester. Den första typen fokuserade på hur väl visualiseringarna fungerade och om de underlättade förståelsen av det som visualiserades, och varje testperson fick testa en visualisering. Den andra typen undersökte hur det uppdaterade användargränssnittet upplevdes i jämförelse med det tidigare gränssnittet, och i dessa tester fick varje person utföra ett antal olika uppgifter kopplade till gränssnittet. Båda testtyperna avslutades med att testpersonen svarade på ett antal påståenden med hjälp av en femgradig Likertskala. Utöver påståendena gavs också möjlighet att lämna kommentarer i fritext. Påståendena som användes, samt svar, finns i appendix B.

4

Implementering

Under projektets gång utvidgades hemsidans användargränssnitt och det gjordes flera visualiseringar av diverse datastrukturer och algoritmer. För detta krävdes det implementering av flera delar som tillsammans kan utgöra pedagogiska visualiseringar och en användarvänlig webbapplikation. I detta kapitel beskrivs implementeringsprocessen för dessa delar, samt motiveringar för vissa val som står till grund för den slutprodukt som beskrivs i Appendix A.

4.1 Kö och stack

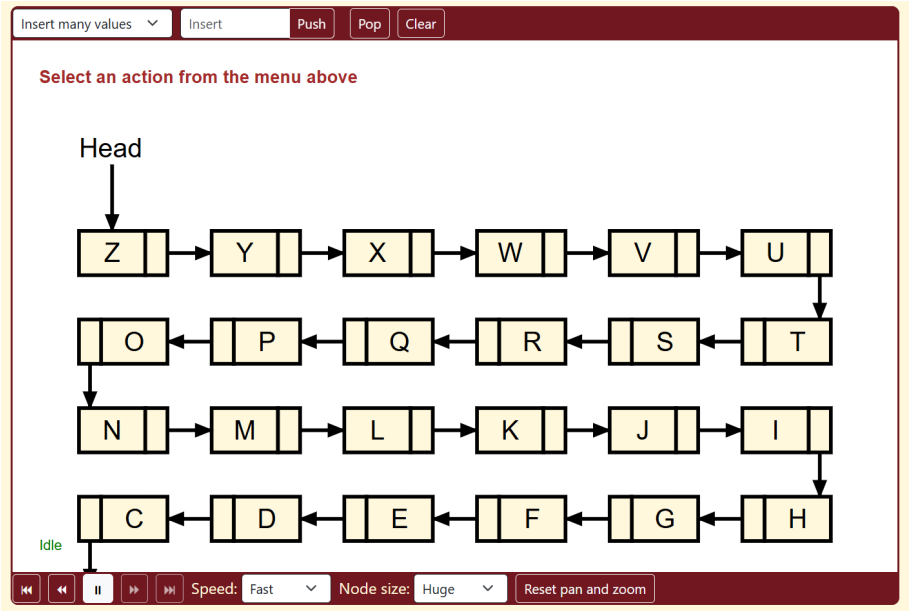
Köer och *stackar* är datastrukturer som håller koll på vilken ordning olika element ska behandlas i. En kö följer principen först in, först ut, där det äldsta elementet behandlas först. En stack följer sist in, först ut principen, där det senaste elementet behandlas först. Både köer och stackar kan skapas med hjälp av olika underliggande strukturer, exempelvis länkade listor och dynamiska arrayer. En länkad lista är en rekursiv datastruktur som består av ett antal noder. Varje nod innehåller ett värde samt en referens till nästa nod. Denna referens kan även vara tom, vilket indikerar att noden innehåller det sista värdet i listan, och den kallas också för tail. En dynamiskt array beter sig som en vanlig array, förutom att dess storlek justeras beroende på hur många värden som finns i arrayen. Detta förhindrar att onödigt mycket minne reserveras för delar av arrayen som inte används för tillfället.

Implementeringen av kö och stack delades upp i tre delar. I och med att kö och stack förutsätter befintliga strukturer av länkad lista och dynamisk array behövde dessa skapas först. Därefter implementerades kö och stack med deras grundläggande funktioner, såsom `pop`, `push`, `enqueue` och `dequeue`.

4.1.1 Förbättring av länkad lista

Den befintliga implementeringen av länkad lista var redan relativt välgjord, men hade bristande logik i vissa funktioner. I funktionen `insertBack`, som lägger till en nod i slutet av listan, saknades en animation som visar hur algoritmen hittar sista noden. Funktionen `findTail` lades därför till för att animera processen och visa hur lång tid det tar, vilket ger en bättre inblick i tidskomplexiteten för användaren.

I koden fanns en begränsning på den maximala längden som listan kunde anta bero-



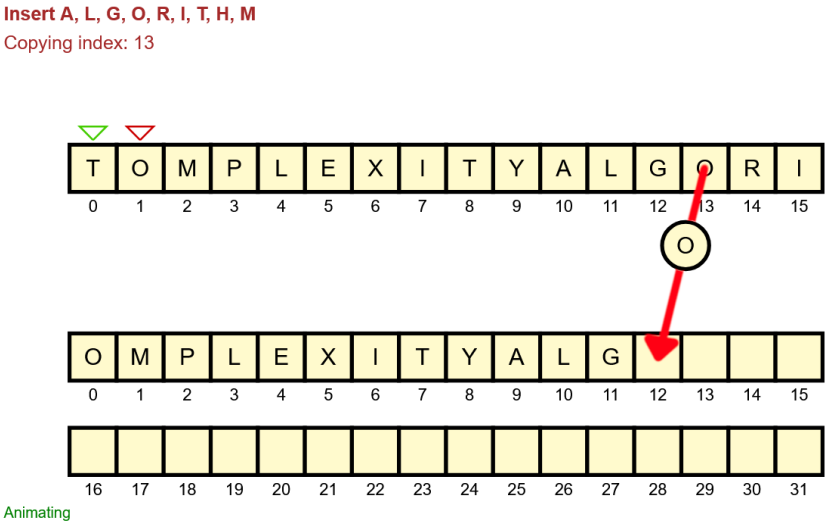
Figur 4.1: En stack representerad av en länkad lista som går utanför ritytan. Överst syns knapparna för `push` och `pop`.

ende på nodstorleken användaren valt. Detta begränsade listan så att den inte gick utanför användarens vy. Det visade sig dock vara problematiskt när användaren skapade en lista med en viss nodstorlek och sedan ökade nodstorleken. Då kunde listan överstiga den maximala längden för den nya nodstorleken, vilket ledde till att noder försvann eller att programmet försökte återskapa fel lista om ytterligare operationer utfördes på den länkade listan. Av dessa skäl valdes det att ta bort begränsningen. Det innebär att man kan skapa en större lista än vad som får plats inom vyn, men detta kan enkelt åtgärdas genom att användaren minskar nodstorleken, eller med hjälp av panorering och zoomning som beskrivs i avsnitt 4.5.1. En överfull länkad lista kan ses i Figur 4.1.

4.1.2 Skapande av dynamisk array

Det saknades en representation av *dynamisk array* i det befintliga biblioteket, och för att fortsätta med köer och stackar behövde detta implementeras. Det fanns dock redan en representation av en array i form av `DSArray`, som användes för sorteringsalgoritmer. `DSArray` hade redan funktioner och visualiseringar som liknar en dynamisk array, såsom pekare som håller koll på start och slut av strukturen, animationer för att lägga till och ta bort värden, samt visuella rutor för varje index. En dynamisk array tillämpar en pekare för start av en kö eller stack, pekaren kallas för head, och en för slutet för köer som heter tail. Genom att utgå från denna struktur underlättades skapandet av en dynamisk array.

Funktionen som saknades i `DSArray` var `resize`, vilket behövde implementeras. Eftersom `DSArray` redan kan ändra storlek behövde den dynamiska arrayen endast ett sätt att visualisera detta. När en dynamisk array når sin maximala storlek måste den expanderas. Detta kan inte göras på plats, eftersom minnet inte garanterar ledigt



Figur 4.2: En *dynamisk array* när `resize` körs. Värdena kopieras från den övre, gamla arrayen till den nedre, som är den nya, större arrayen. ”O” animeras längst röda pilen.

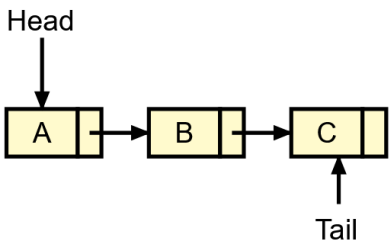
utrymme direkt efter arrayen. Därför måste en ny array skapas varje gång `resize` körs, vilket kan ses i Figur 4.2.

När arrayen blir full och ett nytt värde läggs till anropas `resize` för att utöka kapaciteten. På motsvarande sätt krymps arrayen när element tas bort och den blir till stor del tom. Gränsen för när arrayen krymper kan variera, men i denna implementering halveras storleken när mindre än 25% av arrayen är fylld. Detta gör att det frigörs minne så att den dynamiska arrayen inte tar upp onödigt mycket utrymme. När `resize` körs skapas en ny array med antingen dubbla eller halva storleken beroende på om en förstoring eller en förminskning sker. Alla värden kopieras över från start vid headpekaren, den gamla arrayen raderas och den nya tar dess plats. Animationen används för att visualisera tidskomplexiteten, eftersom `resize` tar betydligt längre tid än övriga operationer.

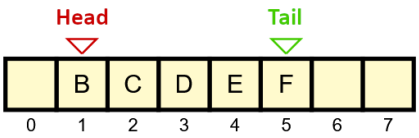
4.1.3 Skapa logik för kö och stack

Slutligen implementerades logiken för *köer* och *stackar*. Dessa är mindre komplexa strukturer än andra i biblioteket. Det beror på att de inte behöver några funktioner som aktivt visar hur ett värde hittas då köer och stackar inte kan radera valfria värden. Knapparna som kontrollerar algoritmerna behövdes därför döpas om: knappen `find` togs bort helt, och `insert` och `delete` döptes om till `enqueue` och `dequeue` för köer, samt `push` och `pop` för stackar. Textfältet kopplat till `delete`-knappen togs också bort, eftersom radering endast sker vid head. Knapparnas utseende kan ses överst i Figur 4.1.

Eftersom både köer och stackar kräver pekare till head, och köer även till tail, så behövdes en visuell representation för dessa pekare i både länkade listor och dynamiska arrayer. Klassen för länkade listor saknade en representation av pekare



(a) *Länkad lista* med head och tail pekare.



(b) *Dynamisk Array* med head och tail pekare.

Figur 4.3: Pekare för de två representationerna av en kö.

till head och tail, så dessa lades till i form av osynliga noder som fungerar som startpunkter för pilarna som pekar på head och tail. För dynamiska arrayer fanns redan funktionen `addArrow`, där en röd och grön pil representerar head- och tail-pekaren. Representationen för pekarna i både länkade listor och dynamiska arrayer kan ses i Figur 4.3

Logiken för stackar implementerades genom att koppla rätt funktion till rätt knapp i gränssnittet. Vid `push` körs funktionen `insertFront` för länkade listor, där den nya noden blir den nod som head-pekaren pekar på. För dynamisk array läggs värdet på positionen efter head-pekaren, och pekaren flyttas sedan till det nya värdet. `Delete` sker också vid head: för länkade listor raderas noden vid head och pekaren flyttas till nästa nod. Dynamisk array tar bort värdet vid pekaren och flyttar pekaren ett steg bak.

Logiken för köer implementerades på liknande sätt, men med en extra tail-pekare. `enqueue` sker efter tail: en ny nod läggs till efter noden som tail pekar på, och tail flyttas till den nya noden. Dynamisk array gör detta på ett liknande sätt som stackar. `Dequeue` sker vid head, där länkade listor fungerar som i stackar, medan dynamisk array flyttar pekaren ett steg fram.

4.2 Hashtabeller

En *hashtabell* är en datastruktur som lagrar värden och kan utföra de flesta av sina operationer såsom `insert`, `delete` och `find` på konstant genomsnittlig tid. Hashtabeller använder en hashfunktion som tilldelar varje värde en specifik position i hashtabellen. En sökning efter ett värde behöver endast ske på den givna positionen, istället för att söka igenom samtliga värden som finns i hashtabellen.

Det finns två huvudkategorier av hashtabeller. *Open adressing*, som lagrar värderna direkt i en array, och *seperate chaining*, där varje index innehåller en referens till

en annan datastruktur som lagrar värderna. Dessa datastrukturer kan vara länkade listor, dynamiska arrayer eller någon annan struktur som kan lagra värden. Flera värden i hashtabellen kan bli tilldelade samma position, och när ett värde ska placeras på en position som redan är upptagen sker en kollision. Det finns många olika kollisionshanteringsstrategier för både *open adressing* och *seperate chaining*. I detta arbetet implementerades *linear probing* för *open adressing*, och *seperate chaining* använder länkade listor för att kunna lagra flera värden på samma position, som då även kallas för en hink.

Open adressing behandlar endast en array, vilken är endimensionell, medan *seperate chaining* behöver en rad med hinkar, där varje hink pekar på en länkad lista som i sin tur kan innehålla flera värden. Därför representeras *open adressing* horisontellt, medan hinkarna i *seperate chaining* representeras vertikalt.

Hashtabeller implementerades som en generell och återanvändbar komponent genom att representeras som en SVG-grupp, där flera SVG-element grupperas till ett objekt. Detta möjliggjorde att samma grundstruktur kan återanvändas i olika visualiseringar vid behov. Implementeringen byggde vidare på den befintliga `DynamicArray`-klassen, som utökades för att stödja operationer specifika för hashtabeller. Genom denna utökning kunde två olika strategier för kollisionshantering implementeras inom samma ramverk: *linear probing* och *separate chaining* med länkade listor.

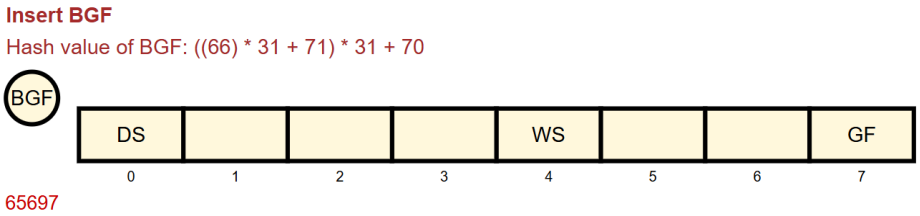
4.2.1 Hashfunktioner

Hashtabeller tillämpar alltid en hashfunktion, vilken exekveras på värdet som ska hanteras av hashtabellen, vid till exempel radering, insättning eller sökning. Hashfunktionen tilldelar värdet till en hink för *seperate chaining*, eller ett index för *open adressing*. Detta sker genom att omvandla texten till ett numeriskt värde genom en algoritm. Funktionen `hashString` implementerades först, med tre olika hashfunktioner: Javas standardhashfunktion, `FirstCharacter` och `SumOfCharacters`. Användaren väljer vilken av de tre hashfunktionerna som ska användas genom en rullgardinsmeny.

Javas standardhashfunktion översätter först varje tecken till ett numeriskt värde baserat på dess värde i ASCII[1], exempelvis är A = 65, B = 66 och så vidare. Javas hashfunktion börjar på noll och för varje tecken multipliceras det nuvarande värdet med 31 och sedan adderas tecknets värde. Till exempel om texten BA hashas med Javas hashfunktion resulterar BA i 66 * 31 + 65 = 2111 (B * 31 + A), där 2111 är det slutliga hashvärdet. De andra hashfunktionerna översätter också tecken till ett numeriskt värde genom att använda ASCII: `SumOfCharacters` summerar värdet för alla tecken, medan `FirstCharacter` endast använder värdet av den första.

Med hjälp av hashfunktionen kan `insert`, `delete` och `find` få en startpunkt genom att ta hashvärdet modulo tabellens längd. För att exemplifiera, när `insert` körs med värdet ”A” på en hashtabell som har längden 8, med `FirstCharacter` som hashfunktion, så hamnar ”A” på index 1. Värdet ”A” har numeriskt värde 65 enligt

[1] American Standard Code for Information Interchange



Figur 4.4: Uträkning av hashvärde för strängen ”BGF” enligt Javas standardhashfunktion vid insättning i en hastabell med linear probing.

ASCII, vilket sedan moduleras med tabellens längd 8 viket ger rest som är 1. Under hashvärdets uträkning visas ett meddelande uttryckt i matematiska termer för att ge en överblick över hur hashvärdet räknas ut. Detta gör att användaren har någonting att förhålla sig till istället för att hashvärdet bara presenteras utan något stöd. Ett exempel på ett sådant uttryck kan ses i Figur 4.4.

4.2.2 Open Adressing

Hashtabeller med *open adressing* liknar mest strukturen av en dynamisk array. Klassen `HashTable` utgick därför från `DynamicArray`. Hashfunktionen baseras oftast på strängar som är längre än ett tecken. För att tydligt visa skillnader mellan olika hashfunktioner behövdes bredare indexrutor än i dynamiska arrayer så att strängar av fler karaktärer kan få plats, och därför dubblerades bredden på rutorna.

Funktioner som `insert`, `delete` och `find` fanns redan implementerade, men behövde anpassas för hashtabeller. Alla dessa funktioner förutsätter funktionen `hashString` (som beskrivs i 4.2.1), som returnerar hashvärdet för strängen som ska sättas in, raderas eller sökas efter. En kollision uppstår när ett värde ska sättas in på ett index som inte är tomt när `insert` körs. För `delete` eller `find` sker en kollision om värdet som hittas på det aktuella indexet inte är rätt värde eller om indexet innehåller en gravsten. Gravstenar är viktiga etiketter som visar att indexet innehöll ett tidigare värde som har blivit raderat. Utan gravstenar riskerar `find` och `delete` att inte kunna hitta värdet. Vid kollision när *linear probing* används, flyttar funktionen till indexet som är direkt efter och upprepar denna process tills en kollision inte längre sker. Om detta sker på sista indexet, så anses 0 vara nästa index. Funktionen `find` letar efter ett värde. Om den stöter på ett tomt index som inte är en gravsten innan värdet hittas kan den konstatera att värdet inte finns i tabellen. Funktionen `delete` fungerar som `find`, men ersätter det hittade värdet med en gravsten.

Hashtabellen håller koll på hur full den är med variabeln `loadFactor`, som representerar antalet fyllda index exklusive gravstenar. När `loadFactor` överstiger 75% körs funktionen `resize`, där tabellen fördubblas. När den understiger 25% halveras tabellen. Detta görs tidigare än i dynamiska arrayer eftersom en nästan full *linear probing* hashtabell blir mycket långsam. Vid `resize` beräknas alla hashvärden om till den nya tabellen, medan gravstenar ignoreras. Detta är nödvändigt eftersom tabellens längd ändras, vilket påverkar vilken index de olika värderna hamnar på.

4.2.3 Seperate Chaining

För att implementera *separate chaining* utökades `HashTable`-klassen med stöd för en vertikal layout, där varje index representerar en hink som kan innehålla en länkad lista. Till skillnad från *open adressing*, där alla element lagras direkt i arrayen, lagras här istället referenser till liststrukturer i varje hink.

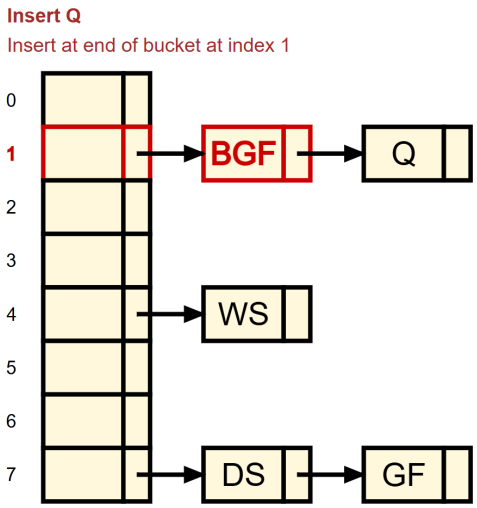
Som nämnts tidigare representeras *separate chaining*-hashtabeller vertikalt. Anledningnen till att göra hashtabeller vertikala ligger i hur den vanligtvis representeras i olika medier, men främst kurslitteraturen [13]. Genom att följa samma visuella struktur som används i undervisningsmaterial och etablerade illustrationer blir implementeringen mer intuitiv och lättare att relatera till för den som läser eller använder programmet.

För visualiseringen krävdes ett alternativt internt koordinatsystem för `HashTable`-klassen. Varje hink positioneras vertikalt baserat på sitt index, medan noder i den tillhörande *länkade listan* växer horisontellt (se Figur 4.5). Varje hink implementerades med en osynlig nod som fungerar som startpunkt för den *länkade listan*. När ett nytt element adderas pekar den osynliga noden på den första riktiga noden i listan. Detta förenklar hanteringen av specialfall vid insättning och borttagning, som när en pekare ska peka från en osynlig nod till den första noden i en lista. För att stödja denna struktur introducerades funktionerna `addLinkedNode` och `removeLinkedNode`.

Funktionen `addLinkedNode` ansvarar för att skapa en ny nod och lägga till den i rätt hink. Vid insättning beräknas först index via `hashString` modulo tabellens storlek. Om hinken är tom skapas en ny lista med noden som första element. Om hinken redan innehåller noder traverseras listan till slutet, där den nya noden länkas in.

Funktionen `removeLinkedNode` används vid borttagning och traverserar listan i aktuell hink för att hitta noden som matchar värdet. När noden hittas uppdateras pekarna så att den tas bort ur listan, inklusive specialfallet då första noden i listan tas bort. Noderna till höger om den borttagna noden justeras sedan för att behålla en konkret och sammanhållen struktur i form av en länkad lista.

Funktionerna `insert`, `find` och `delete` anpassades till denna struktur genom att samtliga operationer först beräknar korrekt hink och därefter utför operationer på den tillhörande länkade listan. Till skillnad från *open adressing* krävdes ingen hantering av gravstenar, eftersom borttagning inte påverkar andra element i tabellen. Hashfunktionen återanvändes dessutom oförändrad från implementeringen av *open adressing*, vilket möjliggjorde att samma gränssnitt kunde användas oberoende av kollisionshanteringteknik.



Figur 4.5: En *Separate Chaining* hashtabel av standardstorlek 8 som lägger till värdet ”Q”

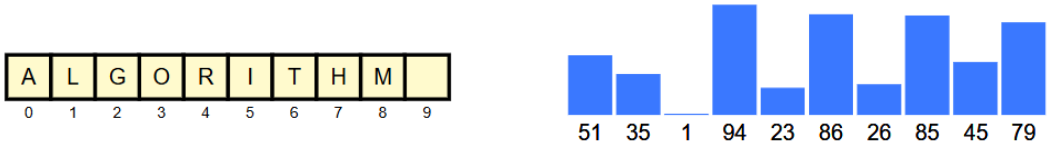
4.3 Sorteringsalgoritmer

En viktig del i kursen som projektet utgår ifrån är olika typer av sorteringsalgoritmer, och de mest centrala algoritmerna är *insertion sort*, *selection sort*, *mergesort* och *quicksort* [1]. Dessa fyra fanns redan implementerade i det befintliga biblioteket från det tidigare kandidatarbetet, men de har utvecklats vidare under projektet. Utöver de centrala sorteringsalgoritmerna har också *bubble sort*, *radix sort* och *heapsort* lagts till. Implementeringen av sorteringsalgoritmerna beskrivs i avsnitten nedan.

4.3.1 Förändringar i utseende

En större förändring som gjordes var representationen av de olika värdena i samlingen som ska sorteras. Ursprungligen representerades värdena som en cell i en array, vilket syns i Figur 4.6a nedan. Detta gjorde att man lätt kunde använda både bokstäver och tal som värden, vilket är flexibelt. Dock var det inte alltid lätt för användaren att uppfatta resultatet av en jämförelse, särskilt inte när värdena består av bokstäver eller någon av de högre animationshastigheterna används. Att snabbt kunna identifiera om exempelvis ’R’ kommer före ’T’ i alfabetisk ordning kan vara svårt, och det flyttar fokus från att förstå hur algoritmen fungerar till att lära sig hur jämförelserna går till. Med inspiration från Galles bibliotek [4] gjordes därför representationen av värdena om till att bestå av staplar med höjder som är proportionerliga mot det värde varje stapel representerar. När staplarna är tillräckligt breda visas även det numeriska värdet med text under stapeln för extra tydlighet. Den nya representationen illustreras i Figur 4.6b.

En begränsning av denna representation är att endast numeriska värden tillåts, då det inte är trivialt att exempelvis omvandla en sträng av en eller flera bokstäver till en representativ höjd på stapeln. Dock ansågs denna begränsning vara av liten betydelse, då huvudsyftet är att visa det grundläggande konceptet för algoritmerna.



(a) Befintliga representationen: en array med celler innehållande värden. I slutet finns en överflödig, tom cell.

(b) Nya representationen: en rad staplar med motsvarande numeriska värde.

Figur 4.6: Figurerna visar **(a)** den befintliga och **(b)** den nya representationen av värdena som ska sorteras.

Konceptet är detsamma oavsett vilken typ av värden som används, så länge det går att jämföra två värden med varandra.

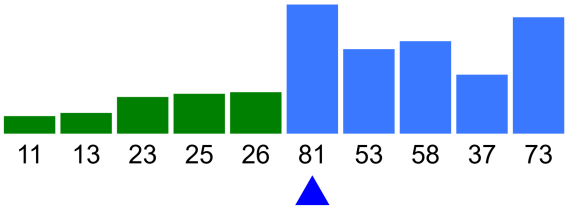
4.3.2 Förändringar vid val av värden

I de befintliga implementeringarna initierades alla sorteringsalgoritmer med en tom samling, och användaren behövde därför mata in värden innan algoritmerna kunde startas. Dessutom animerades inmatningen genom att ett värde i taget flyttades till sin rätta position. Eftersom syftet med denna del av biblioteket är att visa hur sorteringsalgoritmerna fungerar, var animationen för inmatning överflödig och togs bort. Implementeringen ändrades också så att samlingen initieras med ett antal startvärden, och möjligheten för användaren att byta mellan olika stora samlingar av slumpmässiga värden tillfördes. En annan del som rättades till var att ta bort den extra, tomma positionen, vilken kan ses i Figur 4.6a, som den ursprungliga representationen alltid initierades med.

4.3.3 Insertion, selection och bubble sort

Insertion, *selection* och *bubble sort* fungerar alla tre på liknande sätt. *Insertion sort* itererar över alla värden i samlingen ett i taget. Under en iteration flyttas det aktuella värdet mot samlingens början så länge det föregående värdet är större. *Selection sort* itererar istället igenom samlingen position för position. I varje iteration väljs det minsta av de kvarvarande värdena, och det valda värdet placeras sedan på den aktuella positionen. Både *insertion* och *selection sort* bygger på så vis en växande sorterad del i början av samlingen. *Bubble sort* bygger istället en växande sorterad del i slutet av samlingen. I samtliga iterationer stegar algoritmen igenom den osorterade delen av samlingen värde för värde, och om det aktuella värdet är större än det föregående så byter de plats. Därefter stegar algoritmen vidare till nästa värde. Detta innebär att det största värdet i den osorterade delen flyttas upp till slutet av samlingen och hamnar på rätt plats i varje iteration.

Både *insertion* och *selection sort* fanns redan implementerade i biblioteket, och utöver förändringen av representationen av värdena som beskrivs i 4.3.1 ovan gjordes endast några mindre justeringar av visualiseringen för dessa algoritmer. För det första introducerades en pekare som indikerar hur många iterationer algoritmen ge-



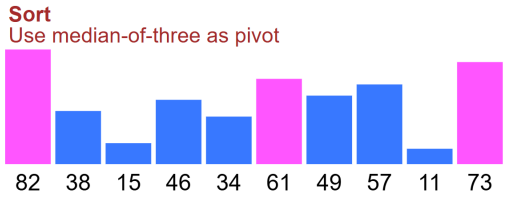
Figur 4.7: Figuren visar pekaren och markeringen som implementerades i *insertion* och *selection sort*. Pekaren indikerar hur många iterationer algoritmen kört, och den gröna markeringen visar den del av samlingen som redan är sorterad.

nomfört, och för det andra markeras nu den del av samlingen som redan är sorterad, vilket kan ses i Figur 4.7. Detta för att användaren lättare ska se hur dessa algoritmer behandlar alla värden i ordning från början till slut, och samtidigt tydliggöra att den del av samlingen som behandlats hittills alltid är sorterad.

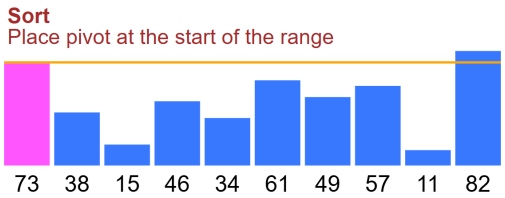
4.3.4 Quicksort

Quicksort skiljer sig från algoritmerna som nämnts ovan eftersom den bygger på konceptet divide-and-conquer, söndra och härska. Det är en form av problemuppdelning, som innebär att algoritmen delar upp samlingen i mindre delar och sorterar dessa för sig, och kombinationen av delarna resulterar i att hela den ursprungliga samlingen blir sorterad. I *quicksort* delas samlingen upp i två delar, kallade partitioner, genom ett partitioneringssteg. Partitioneringen börjar med att ett värde i samlingen väljs som pivot-värde, och därefter flyttas värdena runt i samlingen så att den delas upp i två partitioner, en på var sida om pivot-värdet. I den första partitionen är alla värden mindre än eller lika med pivot-värdet, och i den andra är alla värden större än eller lika med pivot-värdet. Värdena inom de två partitionerna är inte nödvändigtvis sorterade, därför sorteras dessa rekursivt genom att upprepa processen för de två delarna för sig. Till slut består partitionerna av ett enda element, och de är då per automatik sorterade. Kombinationen av alla sorterade partitioner gör att hela den ursprungliga samlingen blir sorterad.

Quicksort fanns också implementerat sedan tidigare, men utöver representationsförändringarna (se 4.3.1) gjordes det även några andra större förändringar. En viktig del i *quicksort* är valet av pivot-värde, då ett dåligt val kan leda till en tidskomplexitet i klassen $\mathcal{O}(n^2)$, istället för $\mathcal{O}(n \log n)$, och därför finns det flera olika strategier för att välja pivot-värde [14]. I den befintliga implementeringen valdes alltid värdet i mitten till pivot-värde, utan att ge användaren någon inblick i hur valet gick till. Eftersom valet av pivot-värde är avgörande för algoritmen infördes möjligheten för användaren att bestämma vilken strategi som ska användas. Ett urval av strategier valdes för att ge användaren möjlighet att förstå problemen med olika val av pivot-värden.Strategierna som implementerades var att välja första, mittersta och sista värdet, medianvärdet av dessa tre, eller ett slumpmässigt värde. Dessutom förklaras valet av pivot-värdet genom att kandidaterna markeras tillsammans med en förklarande text, vilket kan ses i Figur 4.8a.



(a) Val av pivot-värde i *quicksort*, där strategin är att ta medianen av första, mittersta och sista värdet. Kandidaterna markeras, och i övre vänstra hörnet finns en förklarande text.



(b) Pivot-värdet placeras i början av samlingen och en linje som markerar höjden på pivot-värdets stapel ritas ut över hela längden på samlingen.

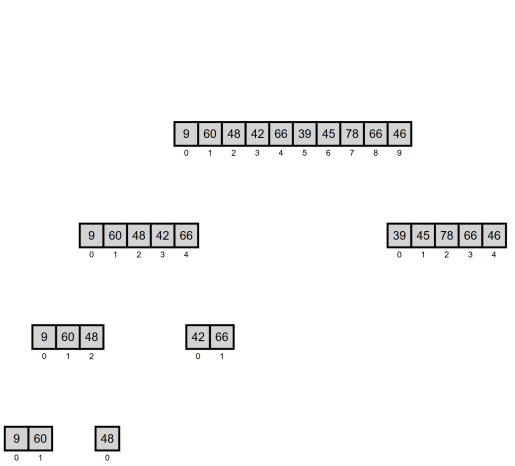
Figur 4.8: Figurerna visar **(a)** exempel på hur val av pivot-värde visualiseras och **(b)** linjen som visar höjden på pivot-värdets stapel. Texten i figurerna har förstorats för att vara mer läsbar.

För att ytterligare underlätta för användaren att förstå resultatet av olika jämförelser, utöver att representera värdena med staplar, lades en horisontell linje till som sträcker sig över hela bredden på samlingen. Linjen visar höjden på stapeln för pivot-värdet, vilket gör det lätt att se om ett värde är större eller mindre än pivot-värdet. Detta kan ses i Figur 4.8b. Linjen tydliggör också uppdelningen mellan de värden som är större respektive mindre än pivot-värdet allteftersom algoritmen kör.

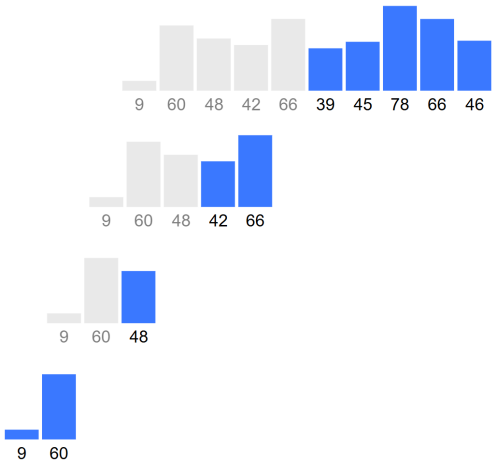
4.3.5 Mergesort

Mergesort bygger precis som *quicksort* på principen söndra och härska. Till skillnad från *quicksort* utför *mergesort* ingen sortering i uppdelningsfasen, utan den delar alltid samlingen i två lika stora delar. Efter att de två delarna sorterats fogas de samman igen, vilket är det viktigaste steget i *mergesort*. I sammanfogningen jämförs det första värdet från respektive del med varandra, och det minsta av dem flyttas till den första lediga positionen i den nya, sammanfogade samlingen. Samma procedur upprepas med resterande värden, ända tills det inte finns några värden kvar i någon av delarna. Sammanfogningen bygger på att delarna är sorterade, och de blir det genom att de sorteras rekursivt med samma metod, ända tills delarna består av ett enda värde, då de är naturligt sorterade.

Även *mergesort* fanns implementerad sedan tidigare, och krävde endast några mindre justeringar. Till skillnad från *quicksort* är *mergesort* svår att göra in-place, det vill säga utan att behöva allokera extra minne. För att visa att extra minne behöver allokeras i de olika stegen och tydliggöra den rekursiva uppbyggnaden visualiseras *mergesort* på ett annorlunda sätt jämfört med de sorteringsalgoritmer som beskrivits ovan. Vid varje rekursivt anrop skapas en kopia av den del av samlingen som ska sorteras, och kopian flyttas ner och bildar en ny nivå av rekursiva anrop. En variant av denna rekursiva struktur fanns sedan tidigare i biblioteket, dock ändrades det vilket tillfälle under algoritmen som kopiorna skapas. I den befintliga implementeringen skapades kopior för båda delarna direkt, innan algoritmen rekursivt började behandla den första delen. Detta kunde orsaka förvirring, eftersom det verkade som



(a) Den tidigare visualiseringen av *mergesort*. De rekursiva anropen på varje nivå påbörjas samtidigt, innan den första anropet är färdigt.



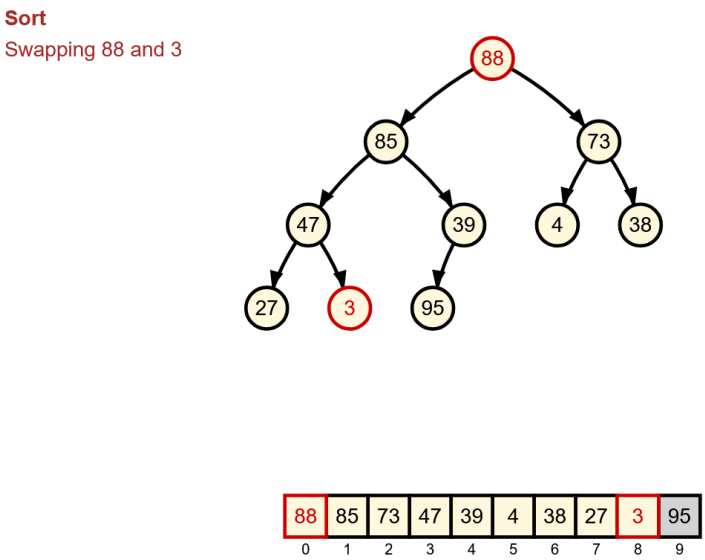
(b) Den nya visualiseringen av *mergesort*. Det andra rekursiva anropet på varje nivå påbörjas inte innan den första anropet är färdigt.

Figur 4.9: Figurerna visar hur den rekursiva uppbyggnaden och behovet av extra minnesallokering vid varje nytt anrop i *mergesort* visualiseras i **(a)** den tidigare och **(b)** den nya implementeringen.

att algoritmen påbörjade båda rekursiva anropen, men sedan avbröt det andra tills det första var färdigt. Istället gjordes justeringen att kopian för det andra rekursiva anropet skapas först efter att den första är helt färdigt. Visualiseringen av de rekursiva anropen och skillnaden mellan den föregående och den nya implementeringen ses i Figur 4.9.

4.3.6 Heapsort

Heapsort skiljer sig från algoritmerna som nämnts ovan eftersom den bygger på användandet av en specifik datastruktur som kallas för en binär heap. Istället för att dela upp samlingen i mindre partitioner, betraktar algoritmen samlingen som ett komplett binärt träd. Algoritmen fungerar i två huvudsakliga steg, uppbyggnaden av heapen och den faktiska sorteringen genom uttag av element. Processen inleds med att samlingen omorganiseras till en så kallad max-heap. I en max-heap gäller regeln att varje föräldranod måste ha ett värde som är större än eller lika med sina barnnoder. Detta garanterar att det största värdet i hela samlingen alltid hamnar längst upp, i trädets rot. När heapen är färdigställd, påbörjas sorteringen genom att rotens värde (det största) byter plats med det sista elementet i samlingen. Eftersom det största värdet nu ligger på sin rätta plats i slutet av samlingen, betraktas det som sorterat och exkluderas från framtida operationer. Det nya värdet vid roten bryter sannolikt mot heap-regeln, och därför körs en process för att ”bubbla ner” värdet till en korrekt position så att resterande del av samlingen återigen bildar en giltig max-heap. Denna process upprepas rekursivt eller iterativt: det största kvarvarande elementet flyttas till den sista osorterade platsen, och heapen återställs. Till slut återstår bara ett element, och hela samlingen har då steg för steg byggts upp till



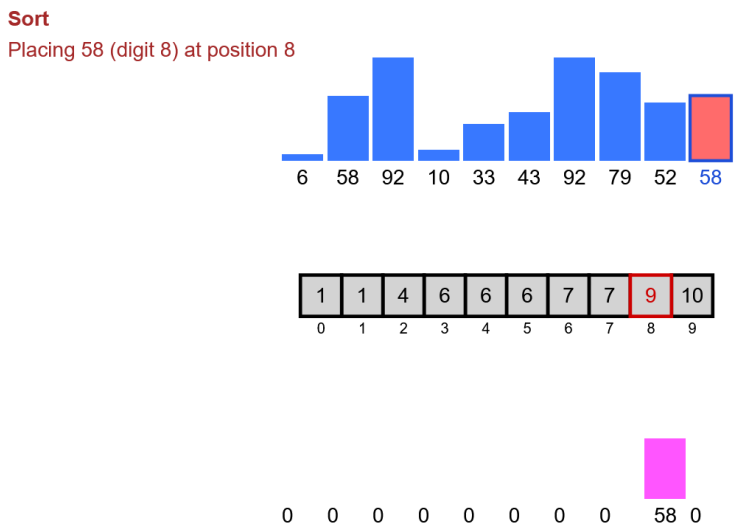
Figur 4.10: Figuren visar när det största värdet för iterationen är hittad och placeras längs bak i arrayen, med det tidigare största värdet utgråat.

att bli helt sorterad. Till skillnad från *quicksort*, som delar upp problemet, utnyttjar *heapsort* strukturen i trädet för att systematiskt identifiera och placera nästa största värde.

Heapsort fanns inte implementerat från det föregående kandidatarbetet vilket gjorde att algoritmen behövde utvecklas från grunden. Återigen togs det inspiration av David Galles bibliotek [4], där *heapsort* visualiseras som en heap i form av ett träd. Logiken för att visualisera en heap som ett träd fanns sedan tidigare i implementeringen av en prioritetskö, och den kunde därför användas som grund för *heapsort*. Genom att utgå från detta så kunde sorteringsalgoritmens logik implementeras. En viktig del i *heapsort* är att när det största värdet är hittat så ska de sättas på den sista platsen i listan. För att förtydliga detta så lades en array till under trädet, så användaren kan se att listan sorteras från arrayens sista värde till dess första. En del av visualiseringen av *heapsort* ses i Figur 4.10.

4.3.7 Radix sort

Radix sort skiljer sig fundamentalt från de tidigare nämnda algoritmerna då det inte är en jämförelsebaserad algoritm. Istället för att jämföra två värden mot varandra för att se vilket som är störst, utnyttjar den talens struktur genom att sortera dem efter deras talsort, det vill sä först efter ental, sedan tiotal och så vidare. Konceptet bygger på att dela upp värdena i kategorier baserat på deras talsorts värde. Processen börjar med att talen sorteras ut i olika hinkar som är numrerade 0-9. Vilken hink talen hamnar i beror på värdet av den minst signifikanta siffran (entalsvärdet) i talet. När alla tal är sorterade i hinkarna baserat på deras ental så placeras de tillbaka ut i arrayen igen, så att den efter första iterationen är sorterad baserat på varje värdes ental (se Figur 4.11). Med första iterationen av denna process så är arrayen sorterad baserat på varje värdes ental. Därefter sker samma process på



Figur 4.11: Värdet 58 blir placera på den näst sist, enligt placeringen vilken hink (fält i arrayen) den är i.

tiotalen och så vidare. Genom att systematiskt sortera talsort för talsort så vandrar värdena successivt mot sin korrekta slutposition. När den sista sorteringsomgången för den högsta positionen är klar, är hela samlingen garanterat sorterad. Detta gör att *radix sort* har en tidskomlpexitet på $\mathcal{O}(n * i)$, där i är antalet talsorter i det största värdet i arrayen. Till skillnad från *quicksort* eller *heapsort*, som manipulerar elementen baserat på deras storlek i förhållande till varandra, förlitar sig *radix sort* helt på talens matematiska representation.

Likt *heapsort* så var inte *radix sort* implementerat i biblioteket sedan tidigare. Ytterligare en gång hämtades det inspiration ifrån David Galles bibliotek [4], där hinkarna representeras som en array som räknar hur många värden som är i de olika hinkarna. Representationen att visualisera värden som staplar är kvar för att visa att algoritmen sorterar olika talsorter åt gången, och inte det faktiska värdet. När iterationen har gått igenom arrayen en gång så kopieras stapeln och placeras på sin nya plats under hinkarrayen. När alla staplar har fått sin nya plats, flyttas de nya staplarna till den ursprungliga platsen och de tidigare staplarna försvinner.

4.3.8 Omkastning av ordning

Tiden det tar för en sorteringsalgoritm att sortera handlar inte bara på storleken på arrayen, utan även ordningen som värdena i arrayen är placerade i från början. För att förtydliga styrkor och svagheter i en algoritm så tar man upp tre fall, bästa, genomsnittliga och värsta fall. En del algoritmer är likgiltiga, det vill säga att de presterar lika bra oavsett ordning på arrayen, medan andra är mycket känsliga. Till exempel har *insertion sort* tidskomplexiteten $\mathcal{O}(n^2)$ i värsta fall, när listan är sorterad i omvänd ordning, medan i bästa fall så är tidskomplexiteten $\mathcal{O}(n)$, när listan redan sorterad.

För att visa olika styrkor och svagheter för sorteringsalgoritmerna så implementera-

(a) Gamla grafnoder

(b) Nya viktade grafnoder

Figur 4.12: Jämförelse mellan grafnoder och viktade grafnoder

des en rullgardinsmeny där användaren kan välja mellan fyra olika metoder för att kasta om ordningen på arrayen. Den första metoden är *Random*, och denna metod kastar slumpmässigt om ordningen på arrayen. Andra metoden är *Reversed*, vilken kastar om så att det första värdet i arrayen kommer sist, det andra kommer näst sist och så vidare. Tredje metoden är *Reversed Sorted*, där listan är sorterad så största värdet är på första platsen i arrayen, det vill säga omvänt sorterad mot hur det brukar vara. Sista metoden är *Almost Sorted*, och denna metod delar upp arrayen i två delar och sorterar de första 70% av antalet värde och sedan slumpar ordningen på de sista 30% av antalet värdena i arrayen.

4.4 Grafalgoritmer

Projektet har implementerat fem olika grafalgoritmer som användaren kan välja mellan, dessa algoritmer är *djupet-först-sökning*, *bredden-först-sökning*, *Dijkstras algoritm*, *Floyd-Warshalls algoritm* och *Prims algoritm*. I denna sektion beskrivs arbetsprocessen för att implementera dessa grafalgoritmer.

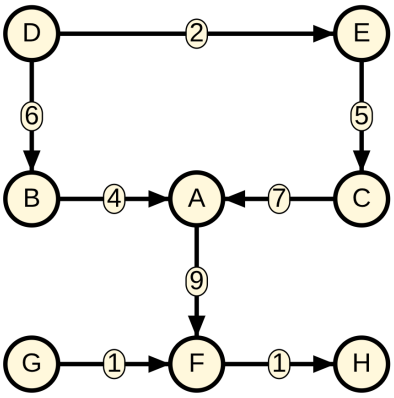
4.4.1 Grundläggande arbete

Olikt många av de andra algoritmer som tas upp kapitel 4, hade kategorin för grafalgoritmer ingen animation implementerad sedan tidigare. Detta betydde att grunden för graferna inte var helt upprättad, det fanns dock existerande material som kunde utvecklas till en grund, till exempel grafnoder och kantobjekt. Dock var dessa inte viktade vilket krävs för flera grafalgoritmer (till exempel *Dijkstras algoritm*, *Floyd-Warshalls algoritm*, *Prims algoritm*, etc.), så därför skapades nya viktade grafnoder och kanter. Skillnaden mellan hur de gamla ser ut jämfört med det nya kan ses i Figur 4.12.

Därefter skapades ett SVG-objekt för en tabell som kan visualisera grafernas kanter, så att användaren lättare kan förstå hur nästa kant väljs. Denna tänktes efterlikna hur vanliga datastrukturer används i algoritmerna, som till exempel hur sättet de sorterar sina element utnyttjas. Den var även avsedd så att det går att visualisera vad algoritmen vet om vid tillfället, till exempel kan det animeras vilka kanter som

From	Weight	To
J	2	D
A	3	B
C	4	B
I	5	H

Figur 4.13: Tabell för implementeringen av *Prims algoritm* (representerar en prioritetskö) som markerar nästa kant som ska utforskas.



Figur 4.14: En acyklisk graf.

läggs till tabellen vid ett visst steg. Det kan också ses hur nästa kant som utforskas väljs med hjälp av en datastrukturs interna sortering. Se Figur 4.13 för hur tabellen ser ut när en ny kant väljs ut i *Prims algoritm*.

Med tabell och både viktade grafnoder och kanter implementerade, var nästa steg att skapa grafer som användaren kan köra algoritmen på. Det övervägdes att låta användaren själv skapa en graf, men som nämns i avsnitt 1.2 ansågs detta för komplicerat att utföra inom projekts tidsram. Istället implementerades förprogrammerade grafer som efterliknar vanliga graftyper. Ett exempel kan ses i Figur 4.14 som representerar en acyklisk graf, det vill säga en graf utan cykler.

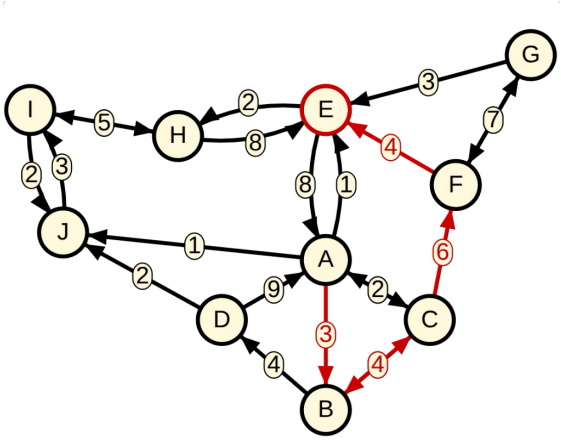
4.4.2 Basklass för visualisering av grafer

Klassen `base-graph` definierar delade beteenden mellan de olika grafalgoritmerna. Denna klass används främst för att undvika att kopiera kod, till exempel innehåller den de förprogrammerade graferna och allmänna funktioner för att manipulera animationerna som används av alla grafalgoritmer. Men den innehåller också ramverk för utvecklingen av nya algoritmer i form av de abstrakta metoder som nämns nedan.

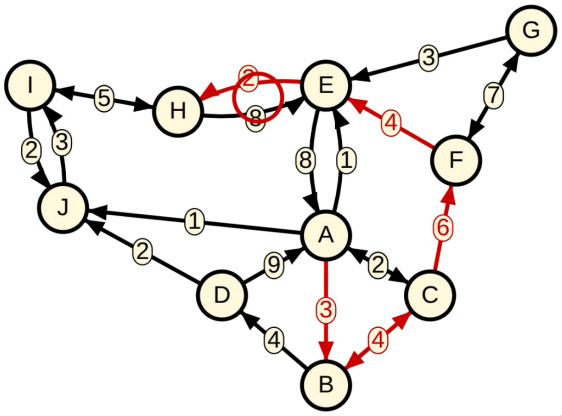
Den abstrakta metoden `updateTable` används för att uppdatera en tabell, denna inkluderas i base-graph för att varje grafalgoritm behöver något sätt att visa varför

nästa kant valdes. Tabellen visar vanligtvis de kanter eller noder som står på tur att väljas härnäst, men varje algoritm kan själv välja sin egen implementering. För att göra det mer tydligt för användare så kan `updateTable` markera en rad i färg (se Figur 4.13), detta används för att göra det tydligt vilken kant eller nod som hanteras just då i algoritmen.

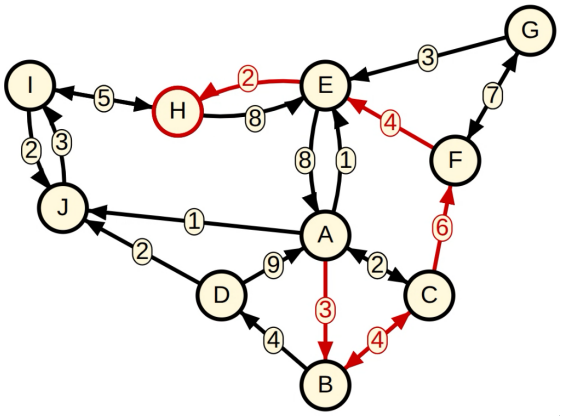
Den abstrakta metoden`nodeTraversalVisualisation` går igenom grafen och visualiserar den på ett sätt som ska hjälpa användare att förstå algoritmen. Ett grundläggande koncept för denna funktion är att börja visualiseringen med att markera en nod med en ring, se Figur 4.15a för ett exempel på markeringen av en nod. Ringens vandring animeras sedan utmed kanten, en del av detta ses i Figur 4.15b, och visualiseringen avslutas sedan i Figur 4.15c där konceptet börjar om från början. Varje subklass kan dock själv definiera hur den ska visualisera sin algoritm på bästa sätt, till exempel använder Prims algoritms en annan typ av visualisering.



(a) Innan visualiseringen börjar. Den nuvarande noden, E, markeras med en röd cirkel.



(b) Halvvägs genom visualiseringen. Den röda cirkeln som markerar den nuvarande noden har flyttats halvvägs till noden H.



(c) Visualiseringen är klar. Den röda cirkeln har flyttats hela vägen och markerar noden H, som nu är den nuvarande noden.

Figur 4.15: Figurerna illustrerar den del av visualiseringen av en djupet-först-sökning då algoritmen vandrar från nod E till nod H.

4.4.3 Djupet-först-sökning

Djupet-först-sökning (DFS) är en grundläggande grafalgoritm som används för att söka igenom en graf. Algoritmen startar i en nod och utforskar rekursivt varje ansluten nod så djupt som möjligt innan den tar sig tillbaka till nästa obesöka nod.

DFS implementeras antingen med en stack eller implicit via rekursion. Vid varje steg markeras den aktuella noden som besökt för att förhindra oändliga cykler, och algoritmen fortsätter därefter till nästa obesökta nod. När en nod inte längre har någon ansluten nod som inte är besökt så går den tillbaka till en nod som har det. När ingen nod är obesökt är algoritmen klar [15]. På detta sätt söker DFS genom hela grafen genom att alltid prioritera att gå så djupt som möjligt innan den utforskar

alternativa vägar.

I DFS klassens implementering av `nodeTraversalVisualisation` valdes att främst förmedla algoritmen genom att illustrera de tre stegen som algoritmen gör i varje iteration.

- Första steget är när algoritmen tar sig till en ny nod så kommer den ihåg att den besökt den noden så den inte försöker göra det igen. Det illustreras genom att animera en röd cirkel som flyttas från tidigare nod till den nya. Samtidigt så markeras kanten i `updateTable` som algoritmen tog.

- Andra steget är att den lägger alla kanter som leder till noder där den inte besökt på en stack. För att visa alla nya kanter som ligger på stacken uppdateras `updateTable` för att visa kanterna.

- Sista steget är att ta bort den översta kanten på stacken och gå vidare till den noden. Kanten markeras med röd färg för att visualisera att algoritmen har besökt den anslutna noden. Dessa markeringar försvinner aldrig, så när algoritmen är klar så kan användare se exakt hur algoritmen tog sig igenom grafen. De tre stegen repeteras tills stacken är tom vilket innebär att hela grafen har genomsökts.

4.4.4 Bredden-först-sökning

Bredden-först-sökning är väldigt lik *djupet-först-sökning*, den enda skillnaden är att algoritmen går igenom kanterna den hittar i en ”first in, first out” metod till skillnad från djupet-först-sökningens ”last in, first out”. Detta möjliggjorde att återanvända visualiseringen av hur algoritmen går från en nod till en annan helt från DFS. Vilket innebar att det bara behövdes implementeras en funktion för att bestämma i vilken ordning kanterna ska besökas och nya meddelanden för att beskriva bredden-först.

4.4.5 Dijkstras algoritm

Dijkstras algoritm är en annat grundläggande grafalgoritm som finns med i kursen projektet utgått ifrån. Algoritmen börjar från en definerad startnod och söker igenom grafen för den optimala vägen till varje annan nod. För att hitta den optimala vägen måste grafen alltså vara viktad. Det används bland annat i GPS system och routningsprotokoll för nätverk.

Dijkstras algoritm implementeras genom att först sätta startnodens avstånd till noll och övriga noders avstånd till oändlighet. Därefter väljer den upprepande gånger den nod som har det minsta kända avståndet och undersöker dess grannar. Om en kortare väg till en granne hittas via den aktuella noden uppdateras avståndet. Denna process fortsätter tills alla noder har fått sina kortaste avstånd bestämda. Ofta sparas också den väg som ledde till varje nod, så att inte bara avståndet utan även själva kortaste vägen går att få tag på.

Algoritmen illustreras genom att iterera över tre steg.

To	Distance
I	8
J	5
H	5
L	4

Figur 4.16: Visualiseringen av `updateTable` i *Dijkstras algoritm* med två kolumner.

- Först tar man bort den översta noden från prioritetskön, alltså den noden som har den kortaste distansen från start noden och går till den. För *Dijkstras algoritm* innehåller `updateTable` två kolumner som visar varje nod samt distansen från startnoden till den noden, med andra ord representerar den prioritetskön. (se Figur 4.16). Därför markeras i detta steget den raden i `updateTable` som blir borttagen, så användare enklare kan följa med var i algoritmen de är.

- I det andra steget undersöks först om noden redan har besökts. Har den tidigare markerats innebär det att den kortaste vägen till noden redan är bestämd, och processen fortsätter då tillbaka till steg ett. Om noden däremot inte har besökts tidigare markeras både noden och kanterna som leder från den aktuella noden till startnoden. När algoritmen är klar kan det enkelt ses att varje nod har besökts och vilken väg som togs dit.

- Efter att en nod har markerats som besökt granskar algoritmen dess grannoder. Om en kortare total distans upptäcks uppdateras värdet och noden placeras på prioritetskön, i annat fall görs ingen ändring. När algoritmen undersöker varje granne markeras respektive nod, vilket gör algoritmens steg mer överskådliga.

4.4.6 Floyd-Warshalls algoritm

Algoritmen *Floyd-Warshall* utförs genom att först notera kostnaden på vägar till omedelbara grannar (bara en kant emellan noderna), där alla andra vägar noteras som oändlig kostnad. Man väljer sedan tre noder som ska bilda en ny väg. Vägen startar i första noden (startnod), andra är ett mellansteg i vägen (mellannod) och tredje är slutet på vägen (slutnod). Den väg som upprättas av de tre noderna jämförs med den gamla noterade vägen (från startnod till slutnod utan mellannod) och byter ut den om den nya vägen har mindre total kostnad. Algoritmen avslutas när varje nod har varit startnod, mellannod och slutnod, vilket ger denna algoritm en kubisk ($O(N^3)$) tidskomplexitet.

	A	B	C	**D**	E	F	G	H
A	0	2	2	∞	∞	∞	∞	∞
B	∞	0	∞	**8**	5	∞	3	∞
C	∞	∞	0	4	∞	∞	∞	∞
D	∞	∞	∞	0	∞	4	∞	∞
E	∞	∞	∞	∞	0	2	∞	∞
F	∞	∞	∞	∞	∞	0	6	2
G	∞	∞	∞	∞	∞	∞	0	∞
H	∞	∞	∞	∞	∞	∞	1	0

Figur 4.17: En del av Floyd-Warshall-söknings algoritmen där tabellen markerar kostnaden av vägen från B till D.

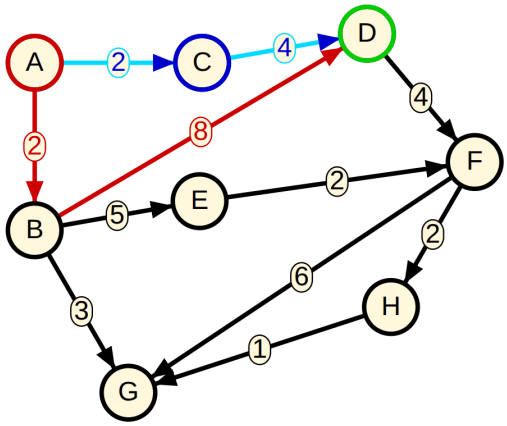
Floyd-Warshall är unik i detta projekt i att den inte använder en datastruktur för att välja nästa kant som skulle utforskas, vilket skapade en underlig situation för tabellen då det blev oklart vilken information den skulle visa och hur informationen skulle formateras. Tabellen omformaterades då till en matris som visar kostnaden på vägen mellan två noder, detta motiverades av att *Floyd-Warshall* ofta använder den kostnaden i de besluten algoritmen gör. När matrisen sen gjordes blev kolumnen till vänster utsedd att visa startnoden och den översta raden blev utsedd att visa slutnoden, så att användaren lättare förstår vilken kostnad som är relevant. Det skapades också funktionalitet så att texten i matrisens rutor kunde betonas för att klargöra informationens relevans. Figur 4.17 visar hur matrisen ser ut och hur den upplyser en vägs kostnad.

Eftersom *Floyd-Warshall* vid varje beslut jämför kostnaden mellan två vägar utvecklades `nodeTraversalVisualisation` för att kunna differentiera mellan dessa vägarna. Se Figur 4.18 för att få inblick i hur gamla vägen i rött visas jämfört med utmanande vägen i blått. Förutom hur vägarna representeras gjordes även blå och gröna cirklar, så att det kan differentieras mellan startnoden (röd cirkel), mellannoden (blå cirkel) och slutnoden (grön cirkel). Denna förändringen kan ses i Figur 4.18 där A är startnoden, C är mellannoden och D är slutnoden.

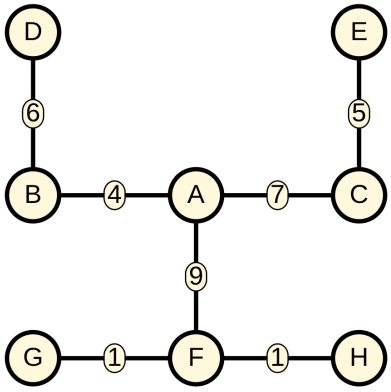
4.4.7 Prims algoritm

Prims algoritm är en algoritm som upprättar ett minimalt uppspännande träd. Detta görs genom att gå igenom varje kant och välja den minst viktade kanten möjligt tills varje nod har blivit besökt. En normal implementering för detta är att använde en prioritetskö. Då kan algoritmen starta i en slumpmässig nod, notera dess kanter och gå igenom grafen med de minsta viktade kanterna som prioritetskön sorterat, där nya kanter noteras när en nod läggs till i det uppspända trädet.

Eftersom *Prims algoritm* upprättar ett uppspännande träd betyder det att den bara funkar på oriktade grafer, de dåvarande implementerade graferna var enbart riktade. Det fanns då möjlighet att antingen skapa nya oriktade grafer eller göra om de nuvarande. Att skapa nya grafer betydde att *Prims algoritm* bara skulle ha



Figur 4.18: En del av *Floyd-Warshall-sökning* där en gammal väg (röd) jämförs med en utmanande väg (blå).

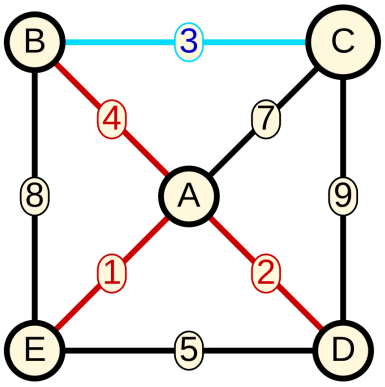


Figur 4.19: En oriktad version av den acykliska grafen på hemsidan.

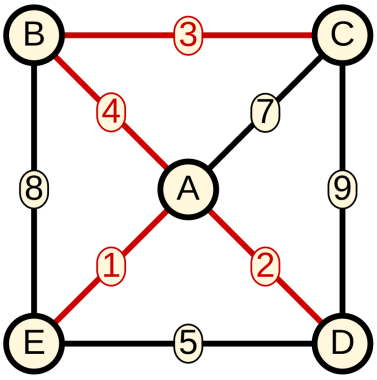
ett fåtal grafer att välja mellan, medan andra implementerade grafalgoritmer kunde välja alla. Att förändra graferna ansågs då vara det bättre alternativet.

För att sedan representera oriktade grafer bestämdes det att ta bort pilspetsarna från kanterna skulle räcka, då detta är det vanliga sättet som oriktade grafer representeras. Ett exempel på en implementerad oriktad graf finns i Figur 4.19 och se Figur 4.14 för att se en riktad version av grafen. I exemplen kan man också se en skillnad på mängden kanter på graferna (Figur 4.19 har en kant mindre), detta är för att erhålla den oriktade grafens egenskap som acyklisk och har utförts med samma motivation på andra implementerade oriktade grafer.

Prims algoritm är en algoritm som bygger upp ett uppspännande träd med minimal total vikt, vilket innebär att algoritmens fokus är att alla noder har en väg till varandra. Detta står till skillnad från de andra implementerade algoritmerna som tar hänsyn till vikt (det vill säga Dijkstra och Floyd-Warshall), dessa börjar i en startnod och söker en optimal väg till de andra noderna i grafen. På grund av denna skillnaden ändrades progressionsanimeringen från den vanliga `nodeTraversalVisualisation` visualisering som beskrivs i avsnitt 4.4.2, till att visualisera progression genom att markera nya kanter i blått och de gamla i rött. Med detta tillsattes också funktionen



(a) Ny kant blir vald till trädet.



(b) Vald kant är nu med i trädet.

Figur 4.20: Progressions visualisering för Prims.

att besökta noder krymper för att visa att de är ointressanta och redan ingår i trädet. Visualiseringen av progressionen kan ses i Figur 4.20, där Figur 4.20a visar när en ny kant läggs till och Figur 4.20b visar hur en kant i trädet normalt ser ut.

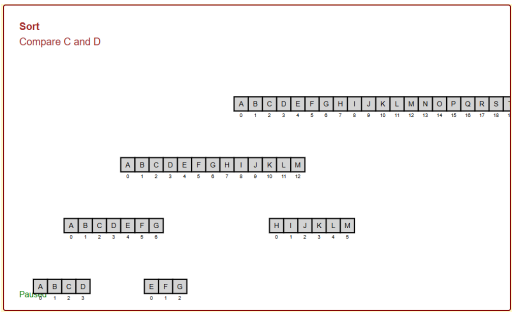
4.5 Övriga förbättringar

Utöver utvecklingen av de olika visualiseringarna genomfördes även övriga förbättringar av biblioteket för att öka användarvänligheten. Dels gjordes ett antal justeringar av visualiseringarnas underliggande funktionalitet, både för att underlätta användningen av biblioteket och åtgärda specifika programfel. Dels uppdaterades bibliotekets gränssnitt för att bli mer attraktivt och intuitivt. I följande avsnitt beskrivs dessa förbättringar.

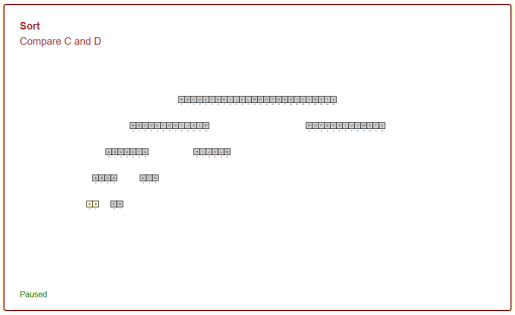
4.5.1 Panorering och zoomning

Ett problem med de befintliga visualiseringarna var hanteringen av situationen då alla delar av en visualisering inte fick plats inom användarens vy. Detta kunde exempelvis uppstå vid sortering med *mergesort*, eftersom de rekursiva anropen expanderade nedåt och till slut försvann utanför vyn. För att användaren inte skulle missa delar av visualiseringarna fanns därför möjligheten att välja storlek på objekten som ritades ut mellan fem fördefinierade storlekar, och på så vis kunde fler delar av visualiseringen få plats inom vyn, något som illustreras i Figur 4.21 ovan. Denna lösning innebar dock några problem; för det första tog utrymmet på ritytan så småningom slut även när den minsta storleken valts. För det andra blev objektens tillhörande text mer och mer svårläst i takt med att storleken minskades. Dessutom framtvingades en anpassning efter de fördefinierade storlekarna, oavsett om det var lämpligt för användarens situation eller inte.

För att åtgärda dessa problem skapades därför möjligheten till panorering, alltså att användaren kan flytta runt ritytan genom att klicka och dra med muspekaren, samt



(a) Pågående *mergesort* med mittersta objektstorleken. Delar av visualiseringen ligger utanför vyn.



(b) Pågående *mergesort* med minsta objektstorleken. Hela visualiseringen syns, men den är svårläst.

Figur 4.21: Figurerna visar samma tidpunkt under en pågående visualisering av *mergesort* med **(a)** den mittersta och **(b)** den minsta objektstorleken.

zoomning, så att ritytan kan förstoras och förminskas. Denna lösning ger ett antal fördelar. Dels finns ingen begränsning på hur mycket ritytan kan förminskas, vilket innebär att godtyckligt stora visualiseringar kan få plats i användarens vy. Dels kan användaren enkelt förstora ritytan under en pågående visualisering för att kunna fokusera på en specifik del eller läsa objektens tillhörande text, och sedan förminska den igen för att få en överblick av hela visualiseringen. Dessutom kan storlek och position anpassas efter vad som är lämpligt i varje stund, och användaren tvingas inte rätta sig efter fördefinierade storlekar.

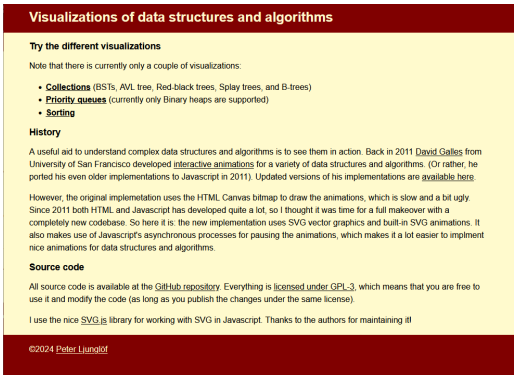
Införandet av panorering var något som även kunde användas för att förbättra implementeringen av *mergesort*. På grund av de rekursiva anropen i *mergesort* förflyttas den aktiva delen av visualiseringen snabbt, särskilt vid högre animationshastigheter, vilket kräver att användaren aktivt flyttar ritytan för att följa efter visualiseringen. För att tillåta att användaren istället kan fokusera på hur algoritmen fungerar introducerades ett alternativ där ritytan automatiskt flyttas, så att den aktiva delen av sorteringen alltid är inom användarens vy.

4.5.2 Åtgärdande av programfel

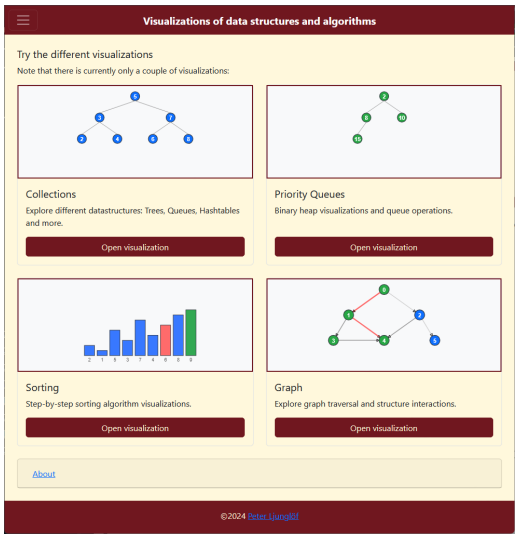
I det befintliga biblioteket fanns ett antal programfel som identifierats, och utifrån deras komplexitet har vissa av dem åtgärdats. Det första felet innebar att positionen för vissa objekt i visualiseringen blev inkorrekt om användaren valde att snabbspola samtidigt som en animation pågick. Detta problem uppstod eftersom positionen för ett objekt i vissa fall beror på positionen av ett annat objekt. Om det andra objektet inte hade sin rätta position när snabbspolningen påbörjades kunde det därför leda till positionsfel. Detta kunde undvikas genom att spara alla animationer på ett och samma ställe, och sedan slutföra alla pågående animationer momentant innan snabbspolningen utfördes.

4.5.3 Användargränssnitt

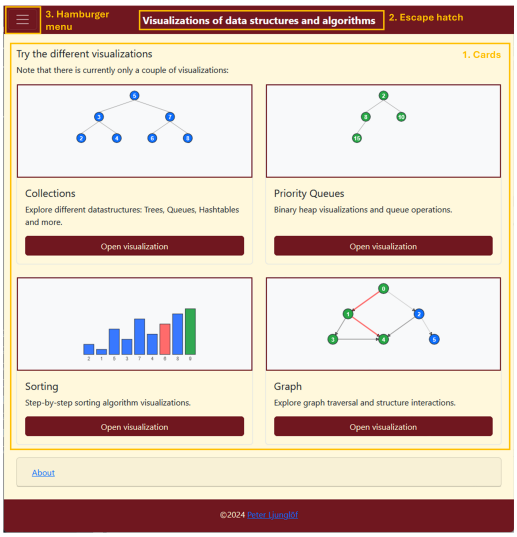
Under projektets gång framkom ett behov av att uppdatera användargränssnittet. Genom att göra knappar och interaktionselement mer intuitiva minskas användarens osäkerhet. Det minskar användarens kognitiva belastning och förbättrar användarens intryck av applikationen. Färgvalet på applikationen behövde ses över så att applikationen kan anpassas för användare med färgblindhet.



(a) Den ursprungliga versionen av webb-applikationen.



(b) Nya versionen, med centrerat innehåll.



(c) De olika designmönstrena inringade 1. Cards, 2. Escape hatch och 3. Hamburger menu icon.

Figur 4.22: Figurerna visar **(a)** den ursprungliga designen av startsidan, **(b)** visar den nuvarande designen av startsidan. Medan **(c)** den nya representationen av webb-applikationen med designmönstrena inringade.

På webapplikationens startsida så implementerades designmönstret Center Stage. Då centreras sidans huvudinnehåll på skärmen och lämnar marginaler tomma, vilket skapar ett fokus på det visuella flödet [16]. Som det föregår i Figur 4.22a så är det svårt att förstå vad det huvudsakliga innehållet är, medan i Figur 4.22b så faller ögonen naturligt mot sidans innehåll. Istället för länkar till de olika visualiseringarna så lades de in som Cards (vilket man kan se inringad i Figur 4.22c), då de håller den specifika informationen för visualiseringarna separerad från det andra och det är ett återkommande designmönster i webbapplikationer [16]. Slutligen lades ett gemensamt sidhuvud till, med en Escape Hatch (vilket man kan se inringad i Figur 4.22c), [16] vilket gör att applikationen navigerar till startsidan om användaren trycker på sidans rubrik. Det lades också till en hamburgar menu ikon på sidhuvudets vänstra sida (vilket man kan se inringad i Figur 4.22c) (en ikon med tre sträck ovanför sig varandra), för att ge möjligheten att navigera mellan alla sidor på applikationen utan att gå tillbaka till startsidan [16].

(a) Den nya versionen med "Colorblind friendlytemat.

(b) Den nya versionen med Dark modetemat.

Figur 4.23: Den nya implementeringen av canvas designen.

På de olika visualiseringssidorna så centrerades också innehållet, och utöver detta så placerades knappar och rullgardinsmenyer närmare canvasen, då användaren associerar närheten mellan elementen att de hör ihop [16]. Se Figur 4.23a. Det lades även till möjligheten att ändra färgschema på applikationen, så att den kan anpassas för färgblindhet. Färgerna som valdes för den är från okabe-ito färgskalan, då den är framtagen för att vara färgblindsvänlig [17]. Det lades också till möjligheten att ha nattläge, vilket man kan se i Figur 4.23b, där hela sidan är mörklagd.

4.6 Automatiserad testning

För att lättare upptäcka fel på ett systematiskt sätt implementerades ett antal automatiserade tester med hjälp av ramverket Cypress [11]. Testerna är olika end-to-end

tester, det vill säga tester som inkluderar alla nivåer av biblioteket och imiterar olika scenarion en riktig användare kan tänkas utföra. Testerna som implementerades behandlar huvudsakligen användarinteraktioner, exempelvis att olika länkar leder rätt, att det går byta till en specifik algoritm eller att användaren kan välja animationshastighet. Utöver dessa implementerades ett antal tester, i stort sett endast för binära sökträd, som kontrollerar att olika funktioner som inmatning, borttagning och sökning gör vad de ska.

5

Resultat

I detta kapitel presenteras resultaten av projektet, både i form av den vidareutvecklade implementationen av visualiseringsbiblioteket och resultaten från användartesterna. Först beskrivs de funktioner, datastrukturer och algoritmer som implementerades samt de förbättringar som gjordes av användargränssnittet. Därefter redovisas resultaten från testningen av biblioteket, där användarnas upplevelser och synpunkter presenteras för att analysera hur väl visualiseringarna stödjer förståelsen av algoritmer och datastrukturer.

En mer detaljerad genomgång av hur biblioteket ser ut och används återfinns under appendix A. Källkoden för projekten finns tillgänglig via Github[1], och förändringarna kommer även integreras in i det ursprungliga projektet av Peter Ljunglöf som också finns tillgängligt i Github[2]. Dessutom finns möjlighet testa den senaste stabila versionen av Ljunglöfs visualiseringar genom hemsidan[3].

5.1 Biblioteket

Projektet resulterade i en omfattande vidareutveckling av det befintliga visualiseringsbiblioteket, där både datastrukturer, algoritmer och användargränssnitt förbättrades och utökades, men den grundläggande strukturen av biblioteket är densamma. För att underlätta visualisering av större strukturer implementerades funktionalitet för att manövrera canvasen genom att dra muspekaren över visualiseringen, samt stöd för att zooma in och ut.

Bland de befintliga strukturerna förbättrades *länkade listor* med visualisering av pekare och en tydligare stegvis traversering genom strukturen, vilket gör det enklare att följa relationerna mellan noder och förstå hur operationer utförs stegvis (se avs. 4.1.1).

Utöver detta utökades biblioteket med *köer* och *stackar*, där båda kan representeras både med *länkade listor* och *dynamiska arrayer* (se avs. 4.1). Dynamiska arrayer introducerades också som en egen struktur med stöd för dynamisk förändring av storlek under körning. Även hashtabeller implementerades, med stöd för både

[1] https://github.com/sigfridsson-aron/dsvis-datx11-26
[2] https://github.com/ChalmersGU-data-structure-courses/dsvis
[3] https://chalmersgu-data-structure-courses.github.io/dsvis/

open addressing och *separate chaining* som kollisionshanteringstekniker (se avs. 4.2). Detta möjliggjorde visualisering av skillnaderna mellan de olika metoderna och hur hashfunktioner påverkar lagring och sökning.

Sorteringsalgoritmerna vidareutvecklades genom förbättrade visualiseringar och tydligare representation av data. Värden representeras nu med staplar istället för enbart textbaserade arrayceller, vilket gör jämförelser och förflyttningar lättare att uppfatta visuellt. Visualiseringarna kompletterades även med pekare och markeringar som visar algoritmernas progression och vilka delar av datastrukturen som behandlas. Dessutom ger *Quicksort* nu användaren möjligheten att själv välja pivot-värde vid sortering, vilket har stor betydelse för hur algoritmen beter sig. Utöver de tidigare implementerade algoritmerna tillkom även *bubble sort*, *heap sort* och *radix sort* (se avs. 4.3.3, 4.3.6 och 4.3.7). Detta gav biblioteket ett bredare stöd för olika sorteringsmetoder och gjorde det möjligt att jämföra algoritmernas fördelar samt nackdelar.

Dessutom har en ny visualiseringskategori för grafalgoritmer skapats. Inom denna kategori implementerades visualiseringar för *bredden-först-sökning*, *djupet-först-sökning*, *Floyd–Warshall-algoritmen*, samt *Prim's* och *Dijkstra's algoritm* (se avs. 4.4). Algoritmerna kan användas på ett antal olika grafer med olika strukturer för att illustrera respektive algoritms styrkor och svagheter. Detta utökade biblioteket från att främst fokusera på linjära och hierarkiska datastrukturer till att även inkludera grafbaserade problem.

Även det grafiska gränssnittet förbättrades. Systemet utökades med stöd för mörkt läge och färgblindhetsanpassade färgteman för att förbättra tillgängligheten (se avs. 4.5.3). Dessutom infördes en ny struktur för gränssnittet som gjorde navigationen mellan datastrukturer och algoritmer tydligare och mer lättanvänd.

5.2 Testresultat

Testningen av biblioteket utfördes med två olika typer av användartester, varav den första typen fokuserade på de implementerade visualiseringarna, medan den andra utvärderade det nya gränssnittet. Testerna visar överlag mycket positiva resultat (Se Appendix B).

5.2.1 Visualisering

Majoriteten av testpersonerna som deltog i den första typen av test upplevde att visualiseringarna hjälpte dem att förstå hur algoritmerna och datastrukturerna fungerar. Av de elva deltagarna svarade åtta att de helt höll med´och en håller delvis med om att visualiseringarna förbättrade förståelsen, medan endast två ställde sig neutrala . Ingen uttryckte negativ inställning till påståendet. Även tydligheten i visualiseringarna fick god respons, där sex deltagare helt höll med om att visualiseringarna var tillräckligt tydliga och fyra delvis höll med.

Resultaten visar också att den stegvisa uppdelningen fungerade väl. Åtta deltagare ansåg att stegen var lagom stora och ytterligare två höll delvis med. Den förkla-

rande texten till stegen uppskattades också av majoriteten, även om vissa deltagare efterfrågade tydligare instruktioner och bättre förklaringar kring vad algoritmerna försökte uppnå. Däremot tyckte vissa användare att pausen mellan de olika stegen var för kort, även på den långsammaste inställningen.

Kommentarerna från användarna pekar på flera styrkor hos biblioteket. Många lyfte fram att stapelvisualiseringarna gjorde det enklare att förstå sorteringsalgoritmernas beteende jämfört med tidigare textbaserade representationer. Markeringar av element som jämförs samt animationer som följde operationerna bidrog till att göra algoritmernas processer tydligare. Flera användare uppskattade även möjligheten att experimentera med olika parametrar, exempelvis val av pivot i *Quicksort*, vilket gav en djupare förståelse för algoritmernas egenskaper.

Samtidigt framkom vissa förbättringsområden. Flera deltagare påpekade att vissa gränssnittselement var otydliga, exempelvis knappar utan beskrivningar eller instruktioner som hänvisade till menyer utan att specificera vilka menyer som avsågs. I grafvisualiseringarna efterfrågades fler oriktade grafer och tydligare visualisering av redan beräknade avstånd i algoritmer som *Dijkstra's algoritm*. Även hastighetsinställningarna fick kritik då animationer ibland gick för snabbt för att kunna följas, även på långsammare lägen. Några användare upplevde dessutom problem med zoomnivåer och att inställningar återställdes mellan körningar, vilket försämrade användarupplevelsen.

Jämförelsen med den tidigare versionen av biblioteket visar tydligt att den nya implementationen upplevdes som mer pedagogisk och lättare att använda. Flera deltagare beskrev den tidigare versionen som otydlig, särskilt eftersom den främst använde bokstäver och enklare markeringar istället för staplar och animationer. Den nya versionens stöd för att följa visualiseringarna genom kamerarörelser och musstyrning uppskattades också starkt. Samtidigt framhöll vissa testpersoner att den äldre versionen hade vissa fördelar, exempelvis att sub-arrayer låg kvar efter sortering vilket gjorde det enklare att gå tillbaka och analysera tidigare steg.

5.2.2 Gränssnitt

Användartesterna på gränssnittet visade övervägande positivt där 10 av 12 håller helt med att den övergripande upplevelsen av den nya versionen är en förbättring, medan de 2 andra håller delvis med. Även i detta avsnitt var det ingen som ställde sig negativ till ändringarna på gränssnittet. När det kommer till navigeringen på webbapplikationen så svarade 75% att de håller helt med att det är lätt att navigera runt. Medan 25% säger att de delvis har blivit lätt att navigera runt. Däremot när man jämförde med den tidigare versionen så håller 7 svar helt med att navigeringen blivit lättare, av de 5 svaren som är kvar så håller 4 av 12 delvis med, medan 1 ställer sig neutralt. Det sista som utvärderades var färgschemat, där 11 av 12 håller helt med att det är bra med möjligheten att kunna välja färgschema. Däremot är inte åsikterna om de tillgängliga färgscheman lika positiva, där 75% håller helt med att de är bra, medan 25% enbart håller med till viss del.

6

Diskussion

I detta kapitel diskuteras potentiella förbättringar, utvecklingsmöjligheter för biblioteket och svårigheter under arbetet. Först behandlas svaren från användartesterna och om resultaten var väntade, samt vad som kan förbättras utifrån dem. Förbättringsområden diskuteras också utifrån saker som upptäckts under utvecklingen, och innefattar exempelvis problem med vilken webbläsare som används och möjliga utökningar av de implementerade datastrukturerna och algoritmerna. Dessutom behandlas framtida problemställningar såsom utvärdering av biblioteket och kvalitetssäkring av visualiseringarna. Slutligen tas det upp om användningen av AI i rapporten och projektet.

6.1 Testning och utvärdering

För att kunna fastställa hur väl de implementerade förändringarna underlättar för studenter hade rigorösa, mer systematiska tester behövts genomföras. På grund av tidsbrist var detta inte möjligt, utan endast ett fåtal användartester kunde utföras. Trots detta har testerna fortfarande varit givande i att peka ut vissa styrkor, samt begränsningar som biblioteket har.

Resultaten indikerar att visualiseringarna upplevdes som pedagogiska och hjälpsamma för användaren att bättre förstå olika algoritmer. Särskilt uppskattades det nya utseendet och funktionerna som implementerades för sorteringsalgoritmerna (se avs. 5.2). Detta tyder på att valet att använda mer visuella representationer och tydligare animationer bidrog positivt till användarupplevelsen. Flera användare ansåg att den hjälpande texten generellt gjorde det lättare att följa algoritmernas progression. Däremot påpekade vissa användare att även på den långsammaste inställningen så var pauserna för korta och lät inte användaren läsa texten i tid. Detta tyder på att det behövs mer utveckling av funktionen för att välja animationshastigheter.

Testresultaten visar även att interaktiviteten i biblioteket uppskattades av användarna. Funktioner såsom zoomning, panorering, möjligheten att slumpa datasamlingar och att välja pivotstrategi i *Quicksort* upplevdes bidra till ökad förståelse. Detta ligger i linje med tidigare forskning som visar att interaktivitet och aktivt deltagande kan förbättra lärandet inom algoritmvisualisering [18]. Samtidigt visade testerna att interaktivitet också innebär fler utvecklingsområden, då en ökad mängd inställningar och kontroller även ökar kraven på att gränssnittet är intuitivt och

tydligt.

Resultaten från gränssnittets användartester var överlag mycket positiva, vilket tyder på att det var en nödvändig ändring. Likaså gäller med möjligheten att kunna ändra färgtema och sidans navigering. Från testresultaten går det se att förändringarna ledde till en markant förbättring av användarupplevelsen. Däremot så borde processen för framtagningen av användargränssnittet varit bättre, då i visualiseringens användartester så poängteras en del om placeringen av element och tydligheten i användargränssnittet. Processen av användargränssnittet var drabbad av att omarbetningen av användargränssnittet påbörjades för sent och var inte planerad.

Även grafvisualiseringarna visade på vissa begränsningar. Testpersoner efterfrågade fler oriktade grafer och tydligare visualisering av redan beräknade avstånd i algoritmer som *Dijkstra's algoritm*. Flera användare upplevde också att dubbla pilar i oriktade grafer skapade onödigt visuellt brus. Detta tyder på att grafvisualiseringar ställer särskilt höga krav på tydlighet, eftersom komplexiteten snabbt ökar när många noder och kanter visas samtidigt. Detta var en avgörande anledning till varför grafer endast implementerades med förbestämda strukturer, utan möjlighet för användaren att skapa egna grafer.

Utöver att det behöver göras fler användartester finns även stora möjligheter att utveckla de automatiserade testerna. På grund av att samtliga delar av biblioteket är starkt knutna till hur visualiseringarna presenteras, det vill säga olika delar av ett HTML-dokument, fanns inte tillräckligt med kunskap för att testa biblioteket på något annat sätt än genom end-to-end tester, vilket nämndes i avsnitt 4.6. En begränsning med dessa tester är att det är svårt att kontrollera grundläggande egenskaper för olika delar av biblioteket, eftersom det bara finns möjlighet att testa de delar som presenteras för användaren. För att kontrollera att en nod i ett binärt sökträd är ett högerbarn till en annan nod måste deras positioner jämföras, då det inte finns någon koppling mellan dem förutom en utritad anslutning som ska peka från den ena noden till den andra. De exakta positionerna ändras dock allteftersom noder tas bort eller läggs till i trädet, vilket försvårar processen.

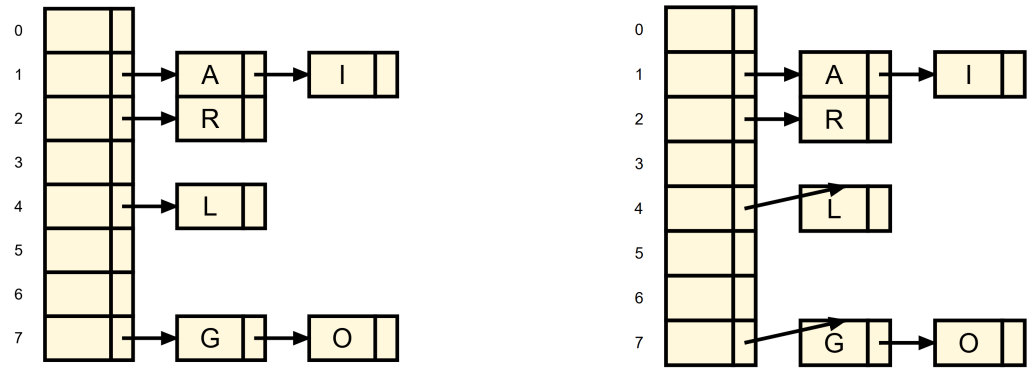
6.2 Förbättringsområden

I denna avsnitt presenteras olika förbättringsområden. Dessa är områden som prioriterades bort på grund av tidsbrist, områden som är naturliga nästa steg efter det som implementerats inom detta projekt, eller områden som är utanför gruppens kompetensnivå.

6.2.1 Problem med Webbläsare

Efter att programmet testades i olika webbläsare upptäcktes det problem som endast uppstod vid exekvering i Google Chrome eller Microsoft Edge. När en separate chaining hashtabell når sin load factor och genomför en resize-operation pekar vissa pekare korrekt, medan vissa andra pekar ovanpå nästa element (se figur 6.1). På grund av tidsbrist har problemet inte kunnat undersökas till fullo och kvarstår,

därmed kommer webbläsarkompatibilitet vara en viktig aspekt att beakta vid vidareutveckling av projektet samt under användartester.



(a) Firefox **(b)** Chrome

Figur 6.1: En separate chaining hashtabell efter en resize-operation som det ser ut i olika webbläsare.

6.2.2 Utöka hashfunktioner

En hashfunktion måste vara deterministisk, vilket innebär att vid samma invärde ska samma utvärde alltid produceras. En bra hashfunktion sprider dessutom sina värden så uniformt som möjligt. Med andra ord bör liknande värden hamna på helt olika platser i hashtabellen [13]. I den nuvarande implementationen finns endast tre hashfunktioner, valda för att de är relevanta och enkla att implementera. I praktiken ser hashfunktioner ofta annorlunda ut beroende på hur datan är strukturerad. Till exempel kan en hashfunktion som används för en tabell skilja sig avsevärt från en som hanterar enbart heltalsvärden. Det är också värt att notera att många hashfunktioner som används i riktiga system är betydligt mer komplexa, svåra att visualisera eller inte lämpade för vår typ av data, som oftast består av endast några få bokstäver.

Å andra sidan uppfyller ingen av de tre hashfunktioner kriterierna för en riktigt bra hashfunktion. Till och med Javas hashfunktion, som är ”den bästa” av de tillgängliga i biblioteket, misslyckas med att sprida datan uniformt. Till exempel hashas värdena ”A”, ”B” och ”C” till 65, 66 och 67 respektive, vilket gör att de hamnar nära varandra. Samma problem uppstår även för längre strängar som ”AA”, ”AB” och ”AC”. En ny hashfunktion som kan sprida korta värden mer uniformt, samtidigt som den är enkel att implementera och visualisera, skulle göra hashtabellsstrukturen i biblioteket betydligt starkare.

6.2.3 Utöka hashtabeller

Under arbetets gång har hashtabeller med linear probing och separate chaining utvecklats, vilket är de vanligaste representationerna. Det finns dock många fler representationer av hash-tabeller som använder olika hashingscheman, såsom *quadratic probing*, *cuckoo hashing* och *double hashing*. Hashtabeller som använder *quadratic*

probing kräver inga större förändringar utöver hur värden söks efter eller läggs till i tabellen, medan andra, som *cuckoo hashing*, fungerar på helt annorlunda sätt. Att implementera fler typer av hashtabeller är därför ett logiskt nästa steg för framtida arbete.

6.2.4 Sorterad länkad lista

Länkade listor har många tillämpningar. En av de mer grundläggande är en sorterad länkad lista, som inte kräver några storleksjusteringar till skillnad från en sorterad dynamisk array. I det tidigare kandidatarbetet utvecklades en funktion för länkade listor som gör det möjligt att lägga till ett värde på valfritt index i listan. Denna funktion användes dock aldrig i någon av de befintliga representationerna av länkade listor, varken i köer eller stackar. Implementeringen av en sorterad länkad lista är därför ett naturligt sätt att utnyttja denna funktion.

6.2.5 Användarskapade grafer

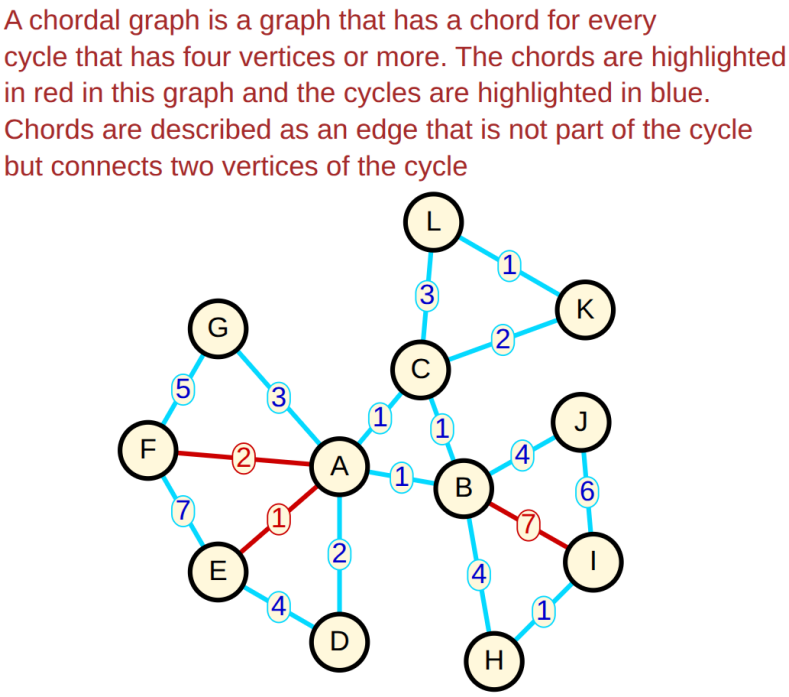
För en användare som vill få en bättre förståelse för en grafalgoritm kan det vara gynnsamt att tillåta den att själv bygga en graf. Genom aktivt lärande blir deltagare mer engagerade, vilket leder till bättre förståelse. [19]. Däremot kan man också argumentera för motsatsen. Om användaren fritt får konstruera godtyckliga grafer finns en risk att algoritmen appliceras på strukturer som den inte är avsedd för, vilket kan leda till dåliga resultat eller feltolkningar av vad den är menad för.

Att tydligare kontrollera vilka typer av grafer som kan användas tillsammans med en viss algoritm, så att de uppfyller algoritmens förutsättningar, är också en framtida förbättring. Till exempel bör algoritmer som förutsätter oriktade grafer eller icke-negativa kantvikter endast demonstreras på sådana. På så sätt minskar risken att användaren utvecklar felaktiga idéer kring när och hur algoritmen är tillämplig. En sådan avgränsning kan därför bidra till en mer korrekt konceptuell förståelse, även om den samtidigt reducerar mängden av egen utforskning.

6.2.6 Utveckla nuvarande grafer

Majoriteten av de nuvarande graferna efterliknar olika graftyper, såsom chordal grafer eller hamiltonska grafer. Dock finns det ingen förklaring över hur dessa grafer uppfyller kriteriet för till exempel en chordal graf. Då kan det anses lämpligt att lägga till något som kan förklara detta.

Ett exempel på hur detta kan göras för chordal är att lägga till en knapp på hemsidan som vid tryck kan uppvisa en text som förklarar vad en chordal graf är. Det kan även upplysas kanter som är relevanta för varför grafen är en chordal graf. I figur 6.2 kan man se ett exempel på hur en sådan förklaring kan gå till.



Figur 6.2: Prototyp på hur förklaring av graftyperna kan gå till.

6.3 Framtida problemställningar

Det finns studier som visar att nyckeln till ett effektivt visualiseringsverktyg för inlärning är att studenterna får möjlighet att interagera med verktyget [18], [19], [20]. En naturlig framtida problemställning blir då dels att utvärdera om interaktiviteten i detta bibliotek är av sådan karaktär att den hjälper användaren att lära sig koncepten, dels att undersöka hur ytterligare möjligheter till att engagera användaren kan se ut. Det skulle exempelvis kunna kunna handla om att implementera ett testläge, där användaren behöver svara på frågor om olika steg och delar av en datastruktur eller algoritm för att kunna gå vidare.

En annan potentiell problemställning är att undersöka hur visualiseringarna kan kvalitetssäkras med hjälp av olika automatiserade tester. Testerna är dels viktiga för att se till att visualiseringarna är korrekta, dels för att underlätta utökning av biblioteket, då det genom regressionstestning kan kontrolleras att olika tillägg och förändringar inte orsakat problem i andra delar av biblioteket. En metod som skulle kunna undersökas är Visual GUI Testing, som använder bildigenkänning för att interagera med och kontrollera det gränssnitt som ska testas [21].

6.4 Användning av AI

Under detta projekt har verktyg baserade på AI (Artificiell intelligens) använts för att underlätta arbetet. Vid utvecklingen av kod användes AI för att hjälpa till att förstå kodbasens uppbyggnad, ge förslag på hur problem kan lösas samt underlätta

vid problemsökning. Vid framställning av denna rapport användes AI-verktyg för att kontrollera grammatik och stavning, hjälpa till med informationssökning och hitta källor. Alla förslag har granskats och justerats efter behov, och vi tar ansvar för det innehåll som presenteras.

Litteratur

[1] Chalmers Tekniska Högskola. ”Kursplan för Datastrukturer och algoritmer.” [Online.], hämtad 29 jan. 2026. URL: https://www.chalmers.se/utbildning/dina-studier/hitta-kurs-och-programplaner/kursplaner/TDA417/.

[2] G. Kogan, H. Chassidim och I. Rabaev, ”The efficacy of animation and visualization in teaching data structures: a case study,” *Educational Technology Research and Development*, årg. 72, nr 4, s. 2349–2372, aug. 2024. DOI: 10.1007/s11423-024-10382-w.

[3] P. Ljunglöf. ”Data Structure Visualizations.” [Online.], hämtad 2 febr. 2026. URL: https://chalmersgu-data-structure-courses.github.io/dsvis/.

[4] D. Gelles. ”Data Structure Visualizations.” [Online.], University of San Fransisco, hämtad 2 febr. 2026. URL: https://www.cs.usfca.edu/~galles/visualization/about.html.

[5] J. Forsberg, I. Hammarlund, D. Holmström, J. Rasheed, A. Sharifi och F. Sundelin, ”Creating an Extendable and Developer-Friendly Framework for Visualizing Data Structures and Algorithms,” kandidatuppsats, Institutionen för data- och informationsteknik, Chalmers tekniska högskola och Göteborgs universitet, Göteborg, Sverige, 2025.

[6] P. Pinki och K. Shekhawat, ”An annotated review on graph drawing and its applications,” *AKCE International Journal of Graphs and Combinatorics*, årg. 20, nr 3, s. 258–281, 2023. DOI: 10.1080/09728600.2023.2218459.

[7] R. Tamassia, *Handbook of graph drawing and visualization*. Providence, Rhode Island, USA: CRC Press, 2013.

[8] Wout Fierens. ”SVG.js.” [Online.], hämtad 7 april 2026. URL: https://svgjs.dev/docs/3.2/.

[9] ”Git.” [Online.], Git, hämtad 26 april 2026. URL: https://git-scm.com.

[10] ”GitHub.” [Online.], GitHub, Inc., hämtad 26 april 2026. URL: https://github.com.

[11] ”Cypress.” [Online.], Cypress.io, Inc., hämtad 26 april 2026. URL: https://www.cypress.io.

[12] ”Webpack Concepts.” [Online.], OpenJS Foundation, hämtad 26 april 2026. URL: https://webpack.js.org/concepts.

[13] A. G. Peter Ljunglöf. ”Data Structures and Algorithms.” Online book, hämtad 30 april 2026. URL: https://chalmersgu-data-structure-courses.github.io/dsabook/html/index.html.

[14] A. Mohammed och M. Othman, "Comparative analysis of some pivot selection schemes for quicksort algorithm," *Information Technology Journal*, årg. 6, nr 3, s. 424–427, 2007. DOI: 10.3923/itj.2007.424.427.

[15] "Depth First Search or DFS for a Graph." [Online.], geeksforgeeks, hämtad 12 april 2026. URL: https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/.

[16] J. Tidwell, C. Brewer och A. Valencia, *Designing Interfaces: Patterns for Effective Interaction Design*, 3rd. Sebastopol, CA: O'Reilly Media, Inc., 2020.

[17] M. Okabe och K. Ito, *Color Universal Design (CUD): How to make figures and presentations that are friendly to colorblind people*, Hämtad: 2026-05-18, 2008. URL: https://jfly.uni-koeln.de/color/.

[18] S. Grissom, M. F. McNally och T. Naps, "Algorithm visualization in CS education: comparing levels of student engagement," i *Proceedings of the 2003 ACM Symposium on Software Visualization*, ser. SoftVis '03, Association for Computing Machinery, 2003, s. 87–94, ISBN: 1581136420. DOI: 10.1145/774833.774846.

[19] S. Freeman m. fl., "Active learning increases student performance in science, engineering, and mathematics," *Proceedings of the National Academy of Sciences*, årg. 111, nr 23, s. 8410–8415, 2014. DOI: 10.1073/pnas.1319030111.

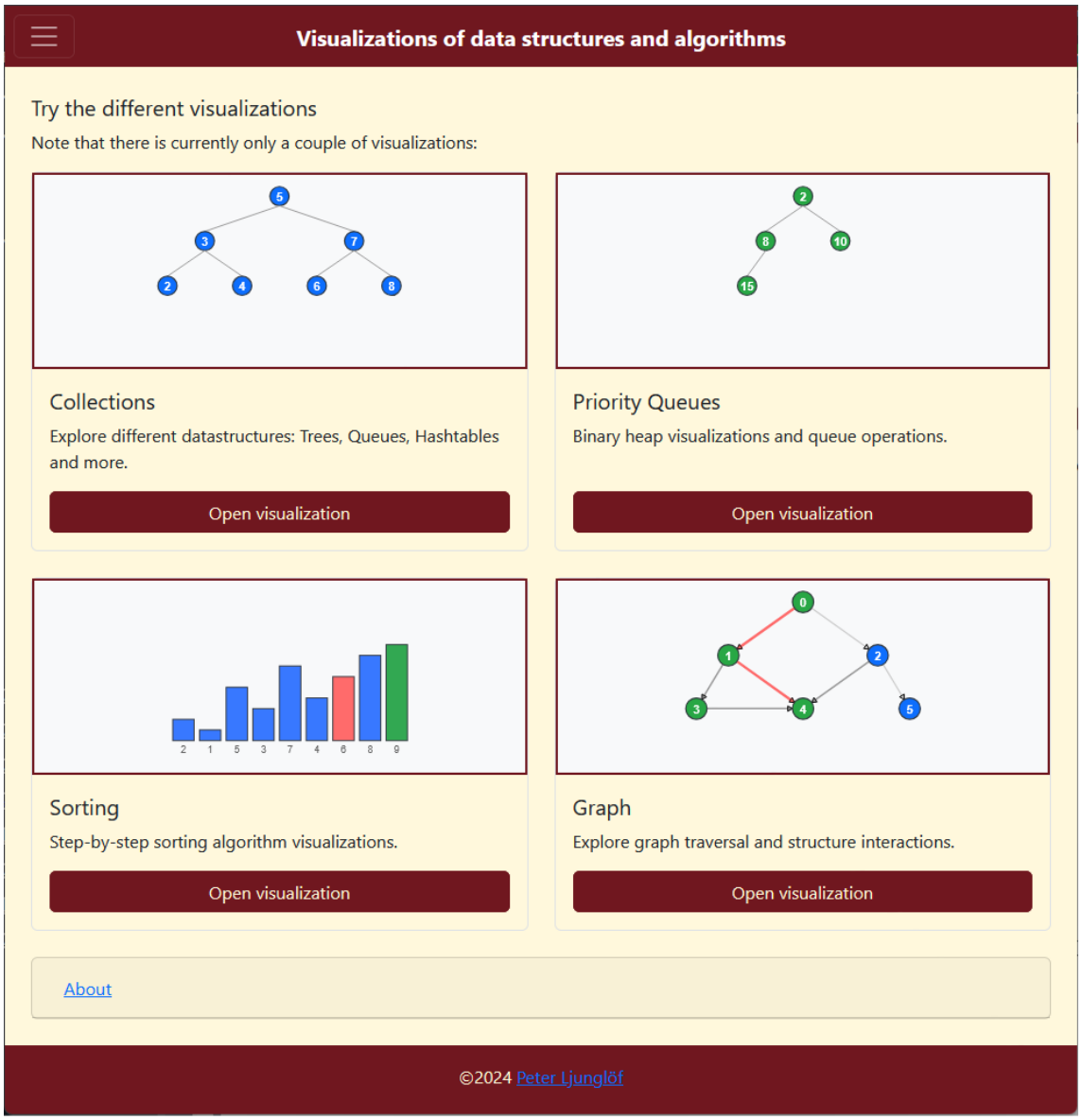
[20] S. Su, E. Zhang, P. Denny och N. Giacaman, "A Game-Based Approach for Teaching Algorithms and Data Structures using Visualizations," i *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '21, Association for Computing Machinery, 2021, s. 1128–1134, ISBN: 9781450380621. DOI: 10.1145/3408877.3432520.

[21] E. Alégroth och R. Feldt, "Industrial Application of Visual GUI Testing: Lessons Learned," i *Continuous Software Engineering*, J. Bosch, utg. Springer International Publishing, 2014, s. 127–140, ISBN: 978-3-319-11283-1. DOI: 10.1007/978-3-319-11283-1_11.

A

User Manual

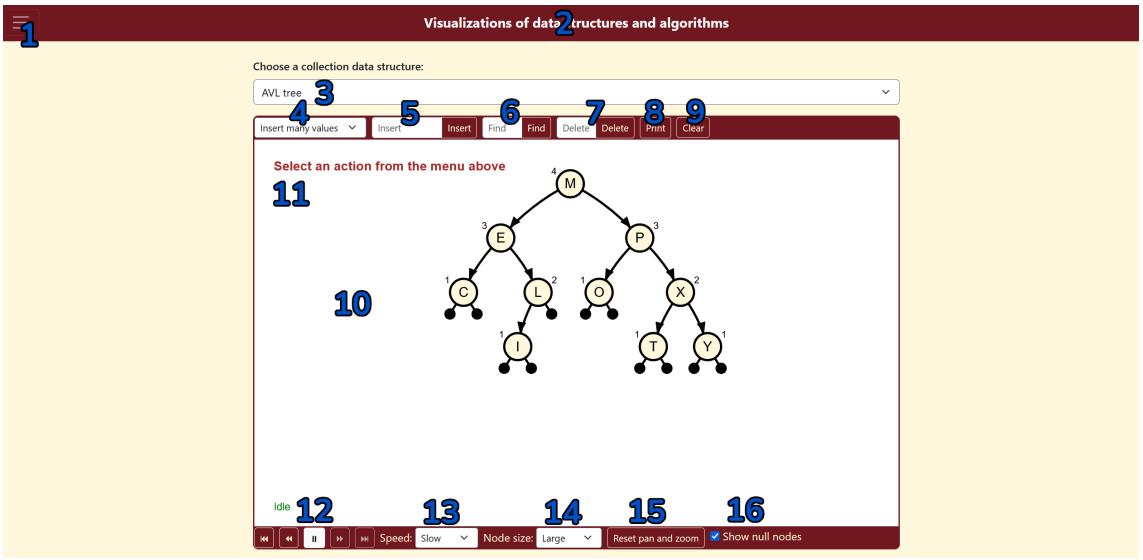
This appendix includes a user manual that describes all the elements and functionalities of the user interface. Since everything on the website is written in english, this user manual will also be in english to appeal to an international audience and be more accessible, and will also serve as a guideline for users when using the website.



Figur A.1: Main page

Figure A.1 shows the main page of the website. It consists of four subpages which can be navigated to by clicking on ”Open visualization” underneath each of the categories. All the data structures and algorithms are categorized and divided amongst these pages to ease the process of finding a specific data structure. Clicking the ”about” text at the bottom of the page will expand the history and additional background about the website.

A.1 Collection and General algorithm controls



Figur A.2: Collection page

Figure A.2 shows the collection page with the AVL tree algorithm selected. The elements on this page are as follows:

1. **Navigation menu**. Clicking this button expands the navigation panel from the left side of the website. There, the user can navigate to other data-structure collections or enable dark mode or color-blindness mode.

2. The title of the website. Clicking this will bring the user back to the main page.

3. **Algorithm selector**. By clicking this dropdown menu, the user can select a data structure to visualize. Upon selection, the canvas below will unlock, and the buttons will become interactable.

4. **Insert multiple values**. Using this dropdown menu, the user can insert many values at once into the chosen data structure. When an option from the dropdown menu is selected, the page will fill the insert field with multiple values. The user can then click the Insert button to insert them.

5. **Insertion field and button**. The user can insert individual or groups of values. When inserting multiple values, the algorithm will separate each values by whitespace. For example, if the user enters ”A B C” into the insertion field. The algorithm will first insert ”A”, then ”B”, then ”C”. In that order.

6. **Find field and button**. Using this field the user can tell the algorithm to search for a specific value. How the algorithm searches for that value depends on the algorithm selected. This field can only search for one value at once, thus whitespace characters are disallowed.

7. **Delete field and button**. Using this field the user can tell the algorithm to delete a specific value, if that value exists in the data structure. Just like the find field, the delete field can only delete one value at a time.

8. **Print button**. It functionality are currently not implemented, and does nothing when clicked.

9. **Clear button**. Clicking this button will prompt a confirmation pop-up to warn the user that this action will reset the canvas and all existing values. Once the confirmation is confirmed. The canvas will be replaced with a blank one.

10. **Canvas**. This is where the data structures and animations will be drawn. This field can also be dragged by holding left-click, or zoomed in or out by holding Control + Mouse Wheel Up/Down for zooming in and out, respectively.

11. **Info messages**. During execution of actions, this field will give a brief explanation of what animation and action is currently being carried out.

12. **Step control buttons**. These buttons control the steps at which the algorithms are run. The **pause button** in the middle controls whether the steps will be proceeded automationally or not. Once clicked during execution of an action on the canvas above, the animations will pause, and the pause button will be replaced by a **play button**. Clicking the play button will resume the algorithm.

The user can also interact with the step buttons to the left and right of the pause/play button. Pressing any of these buttons will either jump to the end of the action or pause the execution.

The left most button is the **undo button**. This button will rewind the data structure to the state before the latest action. This happens instantaneously and without animation.

The button immediately to the left of the pause/play button is the **step-back button**, This button will rewind the algorithm by one step, within the current action. Note that one action usually consists of multiple steps. For example for an insertion of a value in a queue, the algorithm must look at where the head pointer is, move the value into that place, and then increment the head pointer. All of which is one step within that action.

The button immediately to the right of the pause/play button is the **step-forward button**. It functions similarly to the step-back button, but instead steps forward. While the algorithm is paused. The user can use this button to advance through the steps in their own pace.

The right most button is the **fast-forward button**. When this button is pressed. The algorithm will finish all its pending action immediately, skipping any animation and pauses. This is useful when the user would like to insert multiple values at once, and skip the animations.

13. **Speed control**. From this drop-down menu the user can select between four

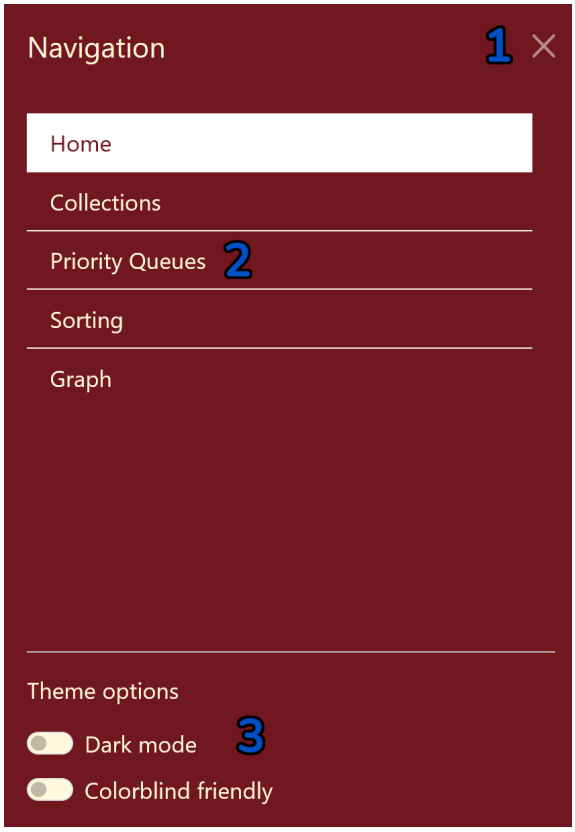
different speeds with which the algorithm runs. This speed effects the length of the pauses between steps within an action, as well as the speed of the animations within each step.

14. **Node size control**. From this drop-down menu the user can select five different sizes for all objects in the canvas. This can be used if the objects gets too big and exceeds the boundaries of the canvas. Alternatively, the user can only use the zooming function of the canvas to increase or decrease the size of the objects.

15. **Reset pan and zoom button**. Clicking this button resets the canvas back to its original position and size before any panning or zooming has taken place.

16. Additional buttons and controls. Depending on which algorithm is selected from the drop-down menu at **3. Algorithm selector**. There may be additional controls on the top or at the bottom of the canvas. In this case, the AVL tree data structure is selected, and there is an additional option to show null nodes or not. Similarly, if a hash table is selected, there will be an additional drop-down menu to select the hash function, and so on.

A.2 Navigation menu



Figur A.3: Navigation menu

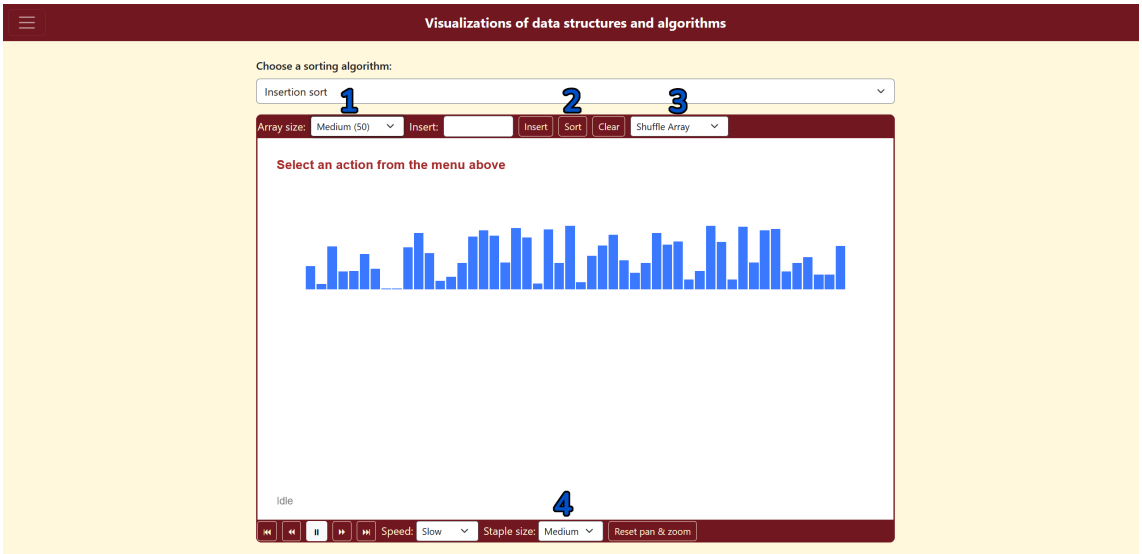
Figure A.3 shows the navigation menu which can be expanded by pressing the icon at the top left of every page. The elements on this page are as follows:

1. **Close navigation menu button**. Clicking this button minimizes this menu.

2. **Quick navigation bar**. This list highlights the page the user is currently on. By clicking any of the other entries, the user can quickly navigate to the other pages or return to the home page.

3. **Theme options control**. These two options toggles between dark mode and colorblind friendly mode. These two will change the color scheme for the whole page, including the objects drawn in the canvas.

A.3 Sorting algorithm controls



Figur A.4: Sort page

Figure A.4 shows the sorting algorithm page with the sorting algorithm ”Insertion sort” selected. The buttons that are unique inside this page and was not already mentioned under Section A.1 are as follows:

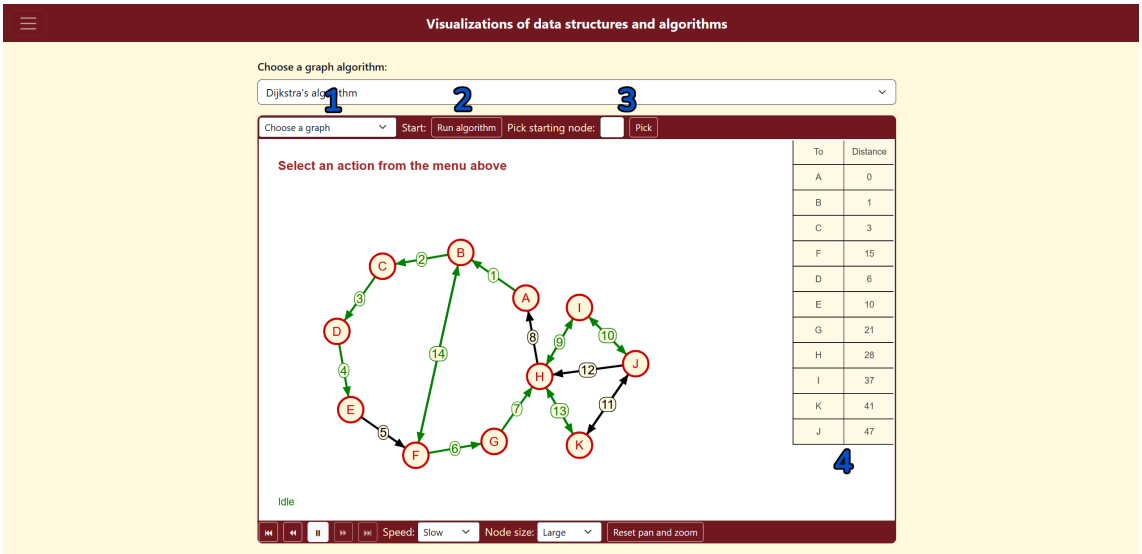
1. **Array size control**. The user can select a specific starting size of an array to be sorted. Once an option under this drop-down menu is selected, the staples will also default to a size so that every single element will be visible on the canvas without the need of zooming or panning.

2. **Sort button**. Clicking this button will start the execution of the selected sorting algorithm, which after its execution, should result in an array that is in ascending order.

3. **Shuffle array menu**. This drop-down menu can be used to manipulate the array by shuffling all elements, reversing the array, or to change the array into an almost sorted state. These are useful with trying to visualize the pros and cons of each sorting array on different types of values.

4. **Staple size control**. Using this drop down menu the user can change the size of the staples in the canvas above. If the size selected is ”large”, or ”huge”. There will also be a number at the bottom of the staples showing it’s exact value.

A.4 Graph algorithm controls



Figur A.5: Graph page

Figure A.5 shows the graph algorithm collection page with the graph algorithm "Dijkstra's algorithm" selected. The buttons that are unique inside this collection and was not already mentioned under Section A.1 are as follows:

1. **Choose graph drop-down menu**. To initiate a graph algorithm, the user must first select a pre-made graph to run the algorithm on. There are 11 different types of graphs to choose from within the drop-down menu, consisting of simple graphs, cyclic graphs, hamiltonian graphs and more.

2. **Run algorithm button**. When the user has selected a graph and the starting node of their choosing. The user can press this button to initiate the graph search algorithm.

3. **Picking starting node field and button**. When a graph is selected, the starting node is always defaulted to "A". If the user wishes to change the starting node, they must enter the node by their letter within this field and click on "Pick". If the node does not exist on the graph, it will default back to "A".

4. **Edge table**. In addition to the graph on the canvas there is a table included. This table, depending on the algorithm, will show either distance from startnode to other nodes or show unexplored edges. For the algorithms Dijkstra and Floyd-Warshall the table will show distance during its animation and the final distances will linger after the animation finishes, see figure A.5 for a concluded Dijkstra and a lingering table. In the case of the other algorithms the table will show unexplored edges during the algorithm and cease to exist upon conclusion of the algorithm.

B

Användartester

B.1 Kommentarer: Generell utvärdering

I denna sektion presenteras de öppna kommentarer som testpersonerna gav efter att ha använt visualiseringsbiblioteket. Kommentarerna är uppdelade efter område för att tydligare visa vilka delar av systemet som användarna hade synpunkter på. Alla svar är numrerade efter respektive svarsperson.

B.1.1 Grafer

Denna del innehåller kommentarer relaterade till visualiseringarna av grafalgoritmer. Testpersonerna gav synpunkter kring tydlighet, tempo, grafrepresentation samt hur lätt det var att följa algoritmernas stegvisa progression.

[1]”Hade velat ha fler oriktade grafer, tror det bara fanns typ en. Dessutom på typ chordal graph så hade den ibland pilar och ibland inte. Tycker att oriktade grafer ej ska ha pilar, det blev mycket visual clutter när det var pilar fram-å-tillbaks med alla. Har jag på slowest hinner jag inte ens läsa, jag tycker att på läge som slowest förväntar man sig att det ska funka. Allt över är jag okej med att man inte hinner. ”Add start node to min stack” var otydligt för början på djikstras.”,

[15]”Knapparna som hoppar till slutet på visualiseringen hade kunnat ha en hover text som beskriver vad de gör, lite oklart just nu. Vissa steg bad om att välja alternativ i menyn ovanför, men det va oklart vilken meny det var. Det saknades visualisering av avstånd som redan har beräknats vilket gjorde det lite svårt att hänga med. Hade varit bra med en kort text som förklarar vad algoritmen eller datastrukturen försöker åstadkomma. Itan den information är det svårt att veta vad man kollar efter”,

B.1.2 Sortering

Här presenteras kommentarer kopplade till sorteringsvisualiseringarna. Användarna beskrev bland annat hur tydliga animationerna upplevdes, hur lätt det var att förstå algoritmernas beteende samt hur kontrollpaneler och inställningar fungerade under användning.

[2]”Jag inserta 67 men den försvann efter jag tryckte på sort och jag blev ledesen över det... Sort knappen är lite out of Place tykcer. Mer naturligt att sort knappen är typ helt bort i sitt egna hörn. Typ upp till höger i rutan istället. Jag tryckte pause och animations spelades fortfarande fast inga av staplarna rörde sig. Jag bara flög runt skärmen på Merge sort Nice animationer, väldigt sattisfying, blev bättre när jag tryckte av follow recursion fast det är bara en personal prefrence tror jag När jag tog större arrayer och hade samma stable size på buble sort verkade som att inget fungera fast det va bara utanför skärmen. Jag tryckte på reset zoom men det va fortfarande utanför skärmen. Allmänt så är animationerna bra men default settings är lite dåliga för mig och det är jobbigt att settingsen resetas mellan sorterings algoritmer och när jag shufflar arrayen. T.ex när jag shufflar arrayen med tiny staple size så blir den large igen.”,

[3]”Det var lite otydligt tyckte jag varför figuren såg ut som den gjorde samt att det var otydligt varför de jämfördes i just den ordningen de gjorde”,

[4]”Detta var en metod som var lite klurigare att förstå vad som hände men staplarna och den gula linjen som ritas upp gjorde att det var lättare att förstå vad som hände genom enbart visualiseringen. Det var också hjälpsamt att man kunde prova att sortera utifrån flera olika parametrar (first, last, median-3 t.ex) för att genom fler angreppsvinklar få en bättre förståelse”,

[5]”Jag tyckte att det var tydligt men det hade möjligtvis kunnat vara lite tydligare med att visa vilka siffor som hamnat på rätt plats och blivit låsta. Det är dock svårt att säga om det hade blivit för mycket olika färger eller dylikt så det ändå inte hade hjälpt.”,

[6]”Tyckte att det blev tydligt hur det fungerade i och med att de element som jämfördes blev tydligt markerade. Att man följde med operationen gjorde det också tydligare att se vad som hände i sorteringen.”,

[7]”Jag tyckte att visualiseringen var bra och tydlig. Men när man körde på snabbaste hastigheten så blev det inte så estetiskt tilltalande att man såg att orden ändrade sig men man hann inte läsa så det kändes lite onödigt. Jag tyckte också att det saknades en knapp att avbryta på.”,

B.1.3 Grafiskt gränssnitt

Följande kommentarer behandlar det övergripande grafiska gränssnittet och användarupplevelsen. Synpunkterna berör exempelvis navigation, struktur, utseende och hur lätt det var att hitta olika funktioner i systemet.

[8]”bra”,

[9]”Istället för endast ”About” hade det varit trevligt men ”About us” knapp istället. ”About” kan vara om lite vad som helst, medan ”About us” vet man att man får info om er!”,

[10]”Sidan kändes rimlig att navigera. Som en person som har läst en kurs i detta

kändes det rimligt att hitta rätt sida för den algoritm jag ville köra”,

[11]”bra strukturerat program”,

[12]”Bilder är najs”,

[13]”De alternativ som va quizar var markerad med ”open visualization” eller liknande vilket var förvirrande”,

[14]”About skulle kunna vara expanderad hela tiden. Missade den först”

B.2 Kommentarer: Jämförelse med tidigare version

[1]”Väldigt bra jobbat utifrån vad ni fick.... Den hemsidan sög jämfört med eran!”,

[4]”Jag tyckte att den gamla var lite otydlig i just själva visualiseringen med vad det var som hände. Dock var det bra att det likt den nya tydligt markerade vilka som har blivit sorterade redan och vilka det är som kontrolleras.”,

[5]”Visualiseringen med staplar gör att det direkt syns vad som ska ske och varför det sker, något som inte är lika tydligt i den gamla med bokstäver och markerade rutor. Det är också lättare att se skillnaden i effektivitet av de olika sorteringsmetoderna i den nya versionen eftersom att det är enklare att få ett grepp om effektiviteten när det visualiseras med större datamängder.”,

[6]”jag tyckte att det var ganska dumt att man inte följde med animationen för när det blev större mängder data som skulle sorteras var det jobbigt att behöva trycka på pilarna för att hänga med. Det faktum att det inte går styra skärmen med musen är ett stort minus jämfört med den nya versionen. Dock tyckte jag att det var lite bra med att all data låg kvar i sub-arrayerna så man kunde gå tillbaka och se hur datan hade sorterats i efterhand.”,

[7]”Tyckte det var svårare att hänga med när det bara var bokstäver och det var lite otydligt vad man skulle sätta in som värde. Saknade randomizer-knappen som fanns i den nya versionen som gjorde det lättare att göra om samma sorterings typ men med nya siffor utan att behöva tömma och skriva dit nya.”,

[8]”mycket bra”,

[9]”Jag hade föredragit om man körde den äldre verisionen först, då man med hjälp av den nyare kunde lättare navigera i den äldre. Bara en småsak. Den nya var mycket mycket bättre. Good job!”,

[10]”Att navigera fram och tillbaka var betydligt enklare i nya versionen, det kändes även lättare att hitta till det man ville hitta. Utseendet, särskilt i landing pagen var betydligt bättre. Tycker dock att det var något svårare att hitta information om sidan i den nya, eventuellt kan about knappen behöva stylas eller placeras om lite.”,

[11]”mer tydligt i nya versionen jämfört med andra, blev tvungen att leta noggrannare i första versionen”,

[12]”Jag gillar bilder. att paus knappen ser ut som en paus knapp är också najs. ljud kanske som sorting algorithms videor”,

[13]”I och med att man kan zooma så ser jag inte så mycket syfte med att sätta nod storlek separat. Ett helskärms läge med flytande kontroller ovanpå hade varit trevligt. Animeringshstigheterna är för snabba, man hinner inte upptäcka vad som har förändrats och läsa texten innan nästa steg börjar. Slowest borde bli slow. Tabeller som visas vid sidan om borde inte följa med i panorering. Gillade panoreringen så att man kan fokusera på de delar som är intressanta för stunden.”,

[14]”Drop-down menyn borde visa vilken graf som är vald”,

[15]”En historik av de senaste stegen hade kanske kunnat förtydliga vad som händer t.ex. när algoritmen blir väldigt kor”

B.3 Utvärdering: frågor om visualiseringar

Utöver de öppna kommentarerna fick testpersonerna besvara ett antal påståenden (A1-A6) om visualiseringarna genom en enkät. Frågorna syftade till att undersöka hur tydliga och pedagogiska visualiseringarna upplevdes samt hur användbara kontrollpanelerna och de förklarande texterna var.

A1: Visualiseringen hjälpte mig att förstå hur algoritmen och/eller datastrukturen fungerar.

A2: Visualiseringen var tillräckligt tydlig.

A3: Visualiseringarna var uppdelade i lagom stora steg.

A4: Den förklarande texten till varje steg hjälpte mig att förstå vad som hände i stegen.

A5: Det var lätt att förstå vad de olika knapparna i kontrollpanelerna gjorde.

A6: Det var lättare att förstå hur algoritmen och/eller datastrukturen fungerar genom visualiseringarna i den nya jämfört med den tidigare versionen.

	Håller helt med	Håller delvis med	Varken eller	Vet ej	Fanns inget att jämföra med
A1	8	1	2	0	0
A2	6	4	1	0	0
A3	8	2	1	0	0
A4	6	3	1	1	0
A5	6	5	0	0	0
A6	4	2	0	2	3

Tabell B.1: Testares upplevelse av biblioteket. Svaren ”håller delvis inte med” och ”håller inte alls med” va även del av svarsalternativen, men fick inga svar.

B.4 Utvärdering: frågor om grafiskt gränssnitt

Testpersonerna fick även besvara frågor (G1-G6) om det grafiska gränssnittet och den övergripande användarupplevelsen. Frågorna behandlade bland annat navigation, färgscheman och jämförelser mellan den nya och tidigare versionen av systemet.

G1: Det var lätt att navigera på sidan.

G2: Det är bra att kunna välja flera olika färgscheman.

G3: De färgscheman som fanns tillgängliga var bra.

G4: Utseendet i den nya versionen var bättre än i den tidigare.

G5: Det var lättare att navigera i den nya jämfört med den tidigare versionen.

G6: Den övergripande upplevelsen av den nya versionen var bättre än den tidigare.

	Håller helt med	Håller delvis med	Varken eller
G1	9	3	0
G2	11	1	0
G3	9	3	0
G4	9	3	0
G5	7	4	1
G6	10	2	0

Tabell B.2: Testares upplevelse av det grafiska gränssnittet. Svaren ”håller delvis inte med”, ”håller inte alls med” och ”vet ej” va även del av svarsalternativen, men fick inga svar.