Visualisering av algoritmer

INF219

Universitet i Bergen

18-May-18

Skrevet av

Kenneth Apeland, Øyvind S. Liland, Philip T. Hoang

Innholdsfortegnelse

1. Introduksjon

Visualisering av algoritmer er et prosjekt som har som formål å kunne illustrere algoritmer og datastrukturer. Prosjektet er i samarbeid med professor Marc Bezem som var underviser for emnet i INF102, Algoritmar, datastrukturar og programmering, høsten 2017 og har som ønske å bruke visualiseringene til undervisning neste gang emnet går.

1.1 Mål for prosjektet

Målet med prosjektet er å lage en nettside som inneholder visualiseringer av algoritmer og datastrukturer. Nettsiden vil kunne være interaktiv slik at brukeren av nettsiden kan kjøre algoritmen etter valgfritt tempo.

Det finnes allerede et tidligere prosjekt for visualisering av algoritmer og datastrukturer, og for å kunne få tilgang til visualiseringene må man laste ned filen. Vi ønsker å konvertere de eksisterende visualiseringene til en nettside hvor alle skal kunne få tilgang til disse uten å ha behovet for å måtte laste ned filen. Vi har også som mål å legge til flere algoritmer og datastrukturer slik at nettsiden skal kunne være til bruk til undervisning eller hjelpe brukeren av nettsiden til å forstå algoritmene bedre.

1.2 Verktøy

Siden vi skal konvertere de eksisterende visualiseringene slik at de kan bli vist på en nettside, tar vi i bruk programmeringsspråket TypeScript. Dette språket kompilerer til JavaScript og vil dermed fungere på nettet. Grunnen til vi bruker Typescript i stedet for JavaScript er fordi TypeScript er et superset av JavaScript og dermed tilbyr mer funksjonalitet enn det JavaScript gjør.

TypeScript har blant annet mulighet for streng type sjekking. Ved bruk av JavaScript vet man ikke hvilken type en variabel har før den har blitt instansiert ved eksekvering. Dette gjør at det med TypeScript blir enklere å oppdage feil siden man kan spesifisere typen en parameter eller variabel skal ha.

Fordi gruppen har mest erfaring med Java, er det enklere for oss å komme raskt i gang med TypeScript siden syntaksen er svært lik Java. TypeScript utvikles av Microsoft og låner ideer fra Java, C# og JavaScript.

I tillegg til TypeScript har vi tatt i bruk rammeverket jQuery som forenkler manipulasjonen av Document Object Model (DOM) elementer, bruken av animasjoner og håndtering av events. jQuery er et Open Source prosjekt som er utgitt under MIT-lisensen.

https://www.typescriptlang.org/

https://jquery.org/

2 Utviklingsprosess

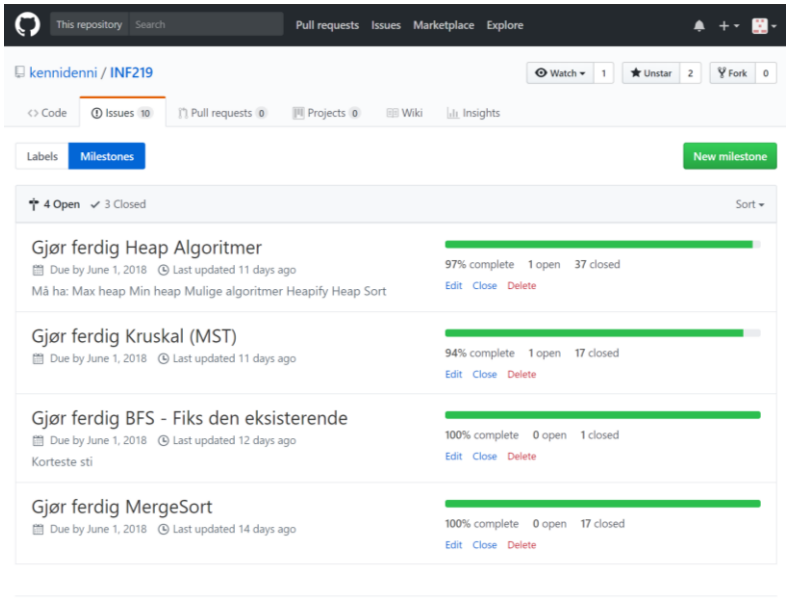
2.1 Smidig utviklingsmetode

Utviklingsmetoden vi tar i bruk var Git-Flow i kombinasjon med GitHub. Dette er en utviklingsmetode som fokuserer å få ut produkter til brukeren. Man implementere funksjoner etter ønske fra brukeren i første omgang, og deretter skal det utgis til brukeren. Dette skjer iterativt hvor man fokusere på oppdateringer og implementasjon av nye funksjoner.

GitHub har også mange funksjonaliteter som man kan bruke sammen med utviklingsmetoden, f.eks. er det lurt å bruke Issue-tracker og Milestone-tracker for å følge med på fremgangen og holde frister.

Figur 1 Commits per dag

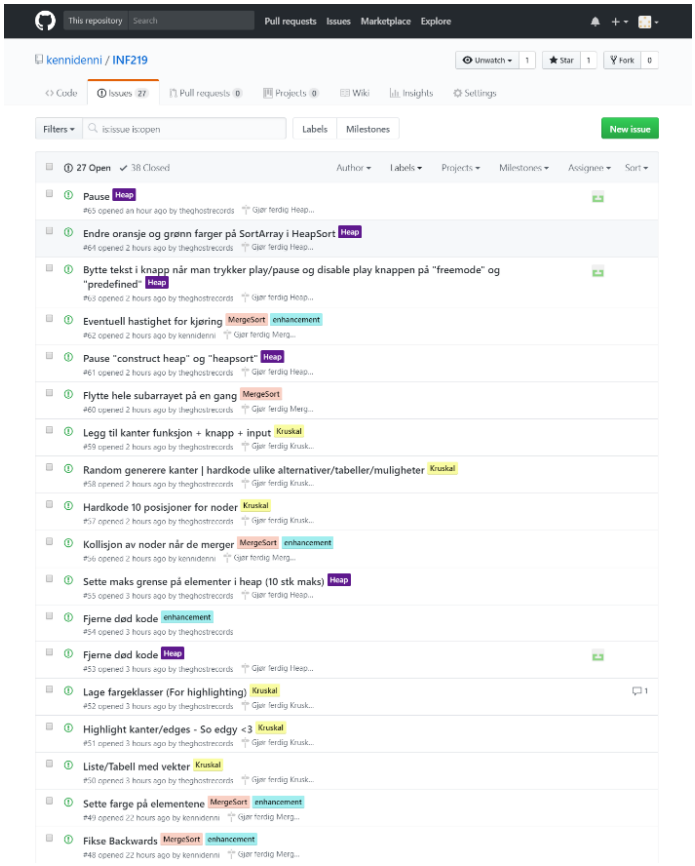
2.2 Prosessen



Figur 2 Oversikt over milestones

Vi begynte med å skrive ned funksjonalitetene som visualiseringene skulle ha og hvordan de skulle visualiseres i en backlog. Backloggen skrev vi i Git Issues og grunnen til vi har valgt å bruke Issues er fordi det gjorde det enkelt å holde styr på hva som skulle bli implementert som f.eks. funksjoner, forbedringer eller bugs. Når man oppretter en issue kan man også tildele issuen til en person slik at man hadde oversikt over hvem som holder på med hva. Dette hjalp oss med å unngå konflikter som f.eks. at to personer holdt på med samme problem. Ved å bruke Git Issues kunne vi opprette milestones som ble brukt for å kunne dele prosjektet opp i flere deler. Issuene kan også bli markert for å vise hvilken milestone den tilhørte, og hver gang en issue ble løst kom vi nærmere milestonen. Denne funksjonen av

Git var veldig nyttig fordi vi da kunne følge med på fremgangen i hver milestone og hvor langt vi hadde kommet i prosjektet.



Figur 3 Oversikt over Git Issues

2.3 Trøbbel i prosessen

I begynnelsen av prosjektet møtte vi på problemer da vi skulle skrive om Java til TypeScript. Grunnen var at vi hadde ingen erfaring med webutvikling. Derfor tok det en del tid før vi kom inn i en rutine om hvordan vi skulle skrive koden og hvordan vi skulle ha designene våre.

Vi hadde også interne problemer med gruppen i februar/mars der det ble veldig skeiv fordeling av hvem som hadde gjort hva. Etter en samtale der vi snakket om hvordan vi skulle jobbe videre, kom vi frem til en løsning som fungerte bra. Løsningen ble å ha møte hver mandag og torsdag, passe på å bruke Issue-trackeren mer aktiv og kommunisere mer de dagene vi ikke hadde møte.

2.4 Designmønster

Siden vi alle jobbet på forskjellige algoritmer, tenkte vi at det var lurt å følge et designmønster som gjorde det enklere å forstå og vedlikeholde hverandres kode. Det tidligere prosjektet fra Kristian, Anders og Ragnhild ble det brukt designmønsteret «Model-view-controller» (MVC). Vi bestemte oss for å bruke

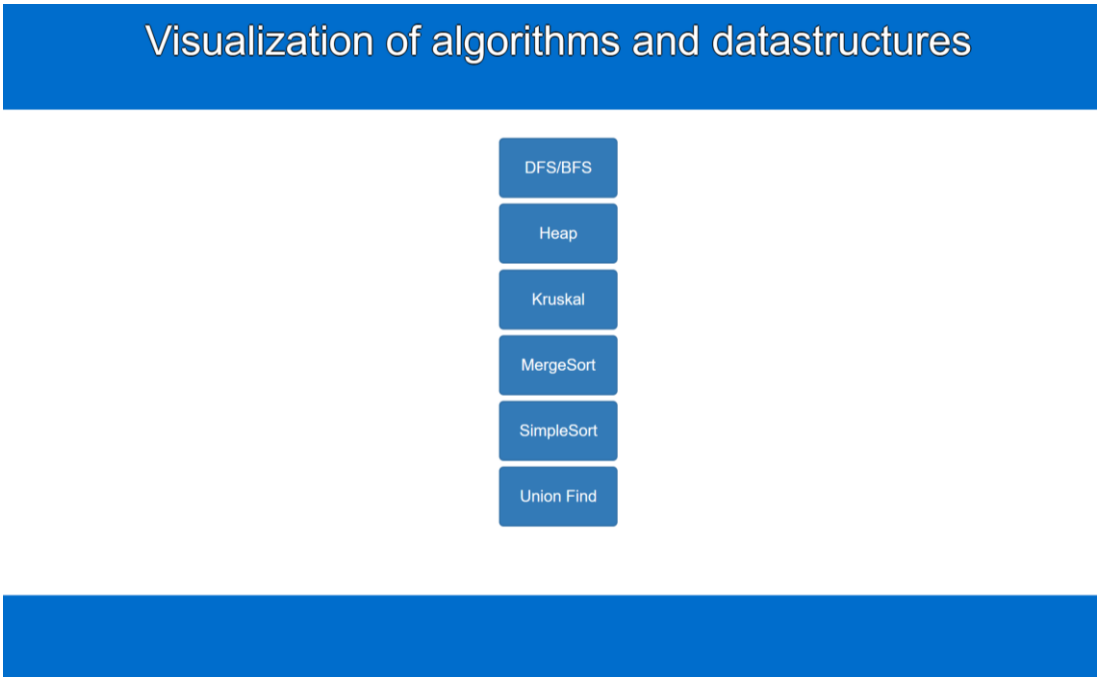
det samme designmønsteret fordi det ga oss en klar seperasjon over delene i algoritmene, spesielt når vi hadde mange algoritmer som skulle implementeres forskjellig. Vi fulgte ikke designmønsteret helt nøyaktig, men tok inspirasjon fra den. For eksempel ble algoritmene delt opp i backend (model) og frontend (view), og på denne måten vil ikke endringer i frontend endre noe i dataene. For at brukergrensesnittet og dataene skal kunne kommuniserer med hverandre må en kontroller-klasse implementeres.

Siden MVC separerer delene i algoritmene fra hverandre, kunne vi gjenbruke det som var felles for andre algoritmer.

3 EventManager

Vi tok i bruk EventManager-objektet fra det tidligere prosjektet som ble skrevet av Knut Anders Stokke. Dette objektet tar imot hendelser (en endring/animasjon i grensesnittet) og informasjon om hvor lang tid som skal gå fra denne hendelsen ble utført til neste hendelse skal utføres. EventManager gjør det også lettere å implementere funksjonaliteter som pause, gå fram og tilbake i animasjonen siden den holder kontroll på hendelsene. EventManager tar vare på hendelsene som skal utføres i en kø, samt samler hendelsene som er blitt utført i en stabel. Når manageren skal gå ett steg frem, blir den neste hendelse i køen utført og lagt til i stabelen. Skal manageren gå et steg bakover, blir det hentet en hendelse fra stabelen og hendelsen blir utført omvendt. F.eks. «flytt A 20px ned» ble «flytt A 20px opp».

4 Hovedside



Ved å trykke på *https://algorithmvis.github.io/* vil brukeren møte på denne hovedsiden. Hovedsiden er veldig enkelt designet og inneholder knapper som viderefører brukeren til den valgte algoritmen.

5 Meny

Vi har prøvd å holde likt design på knappene i de forskjellige algoritmene som har blitt skrevet. Grunnen til dette er at brukeren skal ha knapper som er gjenkjennelig slik at han kan fokusere på selve algoritmen.

Figur 4 Controls

Knappene du kan velge er:

1. Start – for å starte algoritmen og pause når den kjører
2. Frem og tilbake – for å manuelt gå frem eller tilbake i stegene i algoritmen
3. Hastighet for kjøring – algoritmen starter på middels, men du kan velge andre hastigheter etter behov. Hastighetene du kan velge mellom er «Slow», «Medium» og «Fast».

Det er verdt å merke seg at fremover og bakover blir deaktivert når algoritmene (ekskludert Kruskal) kjører automatisk slik at du ikke manuelt kan gå frem eller tilbake. Dette ble lagt til for å unngå problemer under kjøringen som f.eks. at noen hendelser skal overlappe eller overkjøre hendelser som skal bli kjørt ved å trykke på «Forward» eller «Backward» mange ganger.

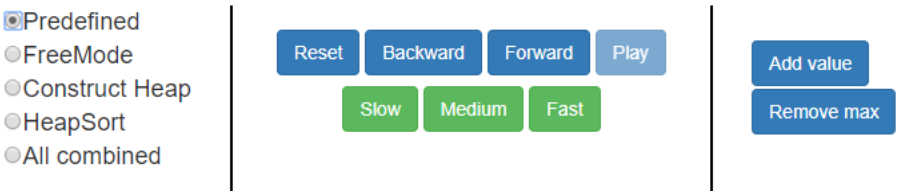
6 Max Heap

VI har valgt å visualisere et maksimum heap. I et maksimum heap er det første elementet i roten det største elementet i datastrukturen og barnenodene er alltid mindre enn forelderen. Dette gjør datastrukturen svært effektiv når man skal lese eller fjerne roten.

I modulen har vi laget fire visualiseringer som skal vise forskjellige aspekter av heaps til brukeren. Disse er følgende

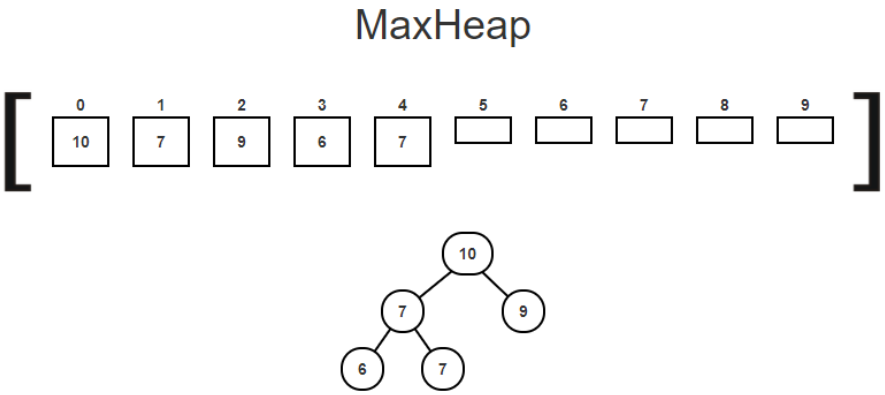
- Predefinert - Denne visualiseringen har som hensikt å vise brukeren hvordan datastrukturen ser ut og hvordan den oppfører seg når man fjerner eller legger til elementer.
- FreeMode – Datastrukturen begynner med et tomt heap hvor man oppretter heapet selv ved å legge til noder. Visualiseringen har samme oppførsel som Predefinert når man legger til elementer og når fjerner elementer.
- Construct Heap - Visualiseringen viser hvordan man går fram for å gjøre en vanlig tabell om til et heap (Slik at heap invarianten opprettholdes i tabellen).
- HeapSort - Denne visualiseringen viser hvordan HeapSort algoritmen fungerer steg for steg
- All combined – Visualiseringen lar brukeren legge til elementer i et binært tre, konvertere det til et heap og sortere elementene med heapsort.

Man bytter mellom de ulike visualiseringene ved å trykke på radio-knappene til venstre i «heap controls» figuren. De andre alternativene i grensesnittet diskuteres nærmere for hver enkelt modul og i kapittelet Meny.



Figur 5 Heap Controls

6.1 Oversikt over design

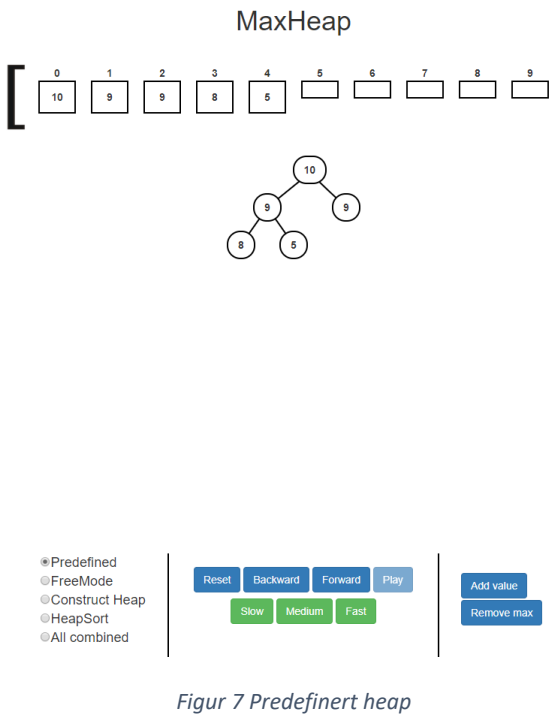


Figur 6 Max Heap design

Den øverste tabellen representerer tabellen slik den er implementert i backend (TypeScript). Nodene nedenfor er en abstrakt representasjon av heapet. Den abstrakte representasjonen er der for å vise studentene hvordan algoritmene fungerer på en mer forståelig måte. Visualiseringen av tabellen er med for å vise sammenhengen mellom hva som skjer abstrakt og hvordan backend utfører operasjonene.

6.2 Predefinert Heap

Slik ser visualiseringen ut når den starter:

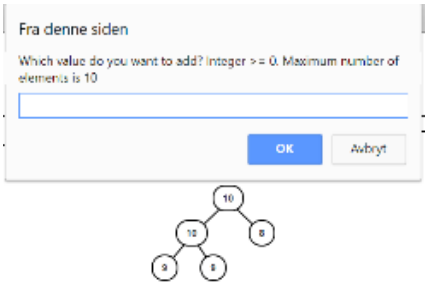


Figur 7 Predefinert heap

I dette alternativet starter visualiseringen med et eksisterende heap på størrelse fem. Brukeren kan fjerne maksimum element og legge til nye elementer. Han kan også velge elementer i tabellen med musen å få markert barna slik at man lett kan få oversikt over hvilken indeks barne-elementene har i tabellen.

- **Legge til nytt element:** Må utføre en «up-heapify» operasjon:
    1. Legg det ønskede elementet til sist i heapet
    2. Sammenlign det tillagte elementet med dens forelder. Hvis de er i riktig rekkefølge – stopp
    3. Hvis ikke bytt elementet med forelderen og gjenta trinn 2-3 til heap-invarianten er oppfylt

  Kompleksiteten avhenger av hvor mange ganger elementet må bytte plass med forelderen så i verste fall er denne: O(log n), og i beste fall er den: O(1)



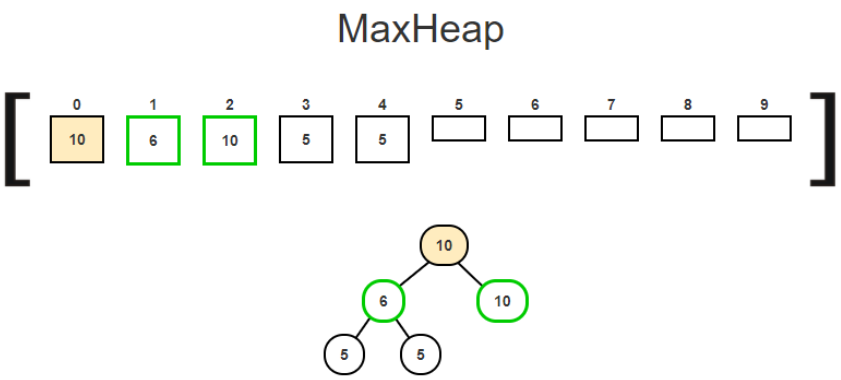
Figur 8 Legg til element

- **Fjerne maksimum element:** Må utføre en «heap-down» operasjon:
    1. Bytt roten med det siste elementet (på nederste nivå i treet)

2. Fjern det siste elementet som nå er maksimum elementet.
3. Sammenlign den nye roten med barne-elementene; Hvis de er i riktig rekkefølge – stopp
4. Hvis ikke, bytt elementet med det største av barna og gjenta 1-3 til invarianten er oppfylt

Kompleksiteten avhenger av hvor mange ganger den nye roten må bytte plass med barna så i verste fall er denne lik høyden av treet altså: O(log n)

- **Velge element med musen**
  Som beskrevet over kan brukeren trykke med musen for å velge et element i tabellen. Det valgte elementet og barna dens vil bli markert både i tabellen og grafen. Denne funksjonaliteten har som hensikt å klargjøre hvilken indeks barne-elementene har. Det valgte elementet blir markert med en oransje farge, mens barna får en grønn farge.



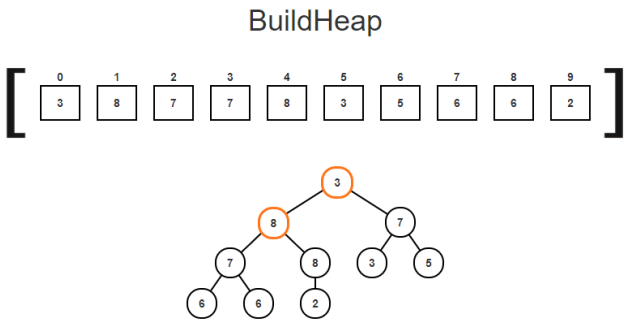
Figur 9 Velge element med mus

6.3 Fri modus / FreeMode

Slik denne er implementert, har den samme funksjonalitet som ''predefinert heap'', men gir brukeren full frihet til å velge hvilke verdier han ønsker at heapet skal inneholde. Derfor starter visualiseringen med null elementer og brukeren kan legge til og/eller fjerner slik han ønsker.

6.4 Construct Heap / Konverter binært tre til maksimum heap

Slik ser visualiseringen ut under kjøring:



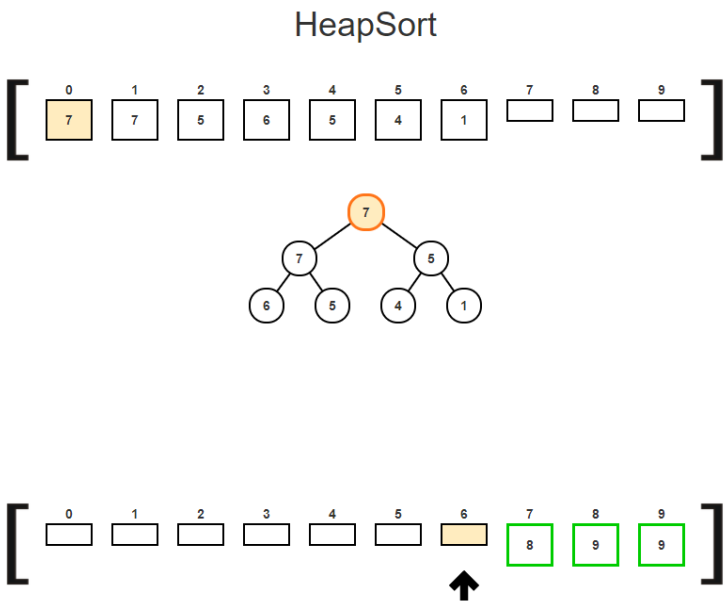
Figur 10 Construct Heap

Når visualiseringen lastes inn vil det genereres en tabell med ti tilfeldige verdier (hvor heap-invarianten ikke er oppfylt). Brukeren kan så trykke play og algoritmen vil starte. Algoritmen fungerer på følgende måte:

1. Start i midten av tabellen, altså elementet på indeks: floor(lengden av tabell / 2)
2. Utfør en «sink»-operasjon, altså sammenlign det valgte elementet med barna, hvis de har riktig rekkefølge utfør steg 2-3 på neste element (i - 1) eller stopp hvis elementet er roten.
3. Hvis de ikke har riktig rekkefølge, bytt plass med det største barnet og gjenta 2-3 til rot-elementet har «sunket» til riktig plass

6.5 Heap Sort

Slik ser visualiseringen ut under kjøring.



Figur 11 Heap Sort

Når visualiseringen lastes inn vil det genereres et heap med ti elementer. Brukeren kan så trykke på start for å starte algoritmen. Vi har valgt å legge til en ekstra tabell i dette eksempelet fordi dette vil tydeliggjøre visuelt hvordan sorteringen fungerer fremfor at den sorterte tabellen opererer på den samme datastrukturen.

Algoritmen fungerer på følgende måte:

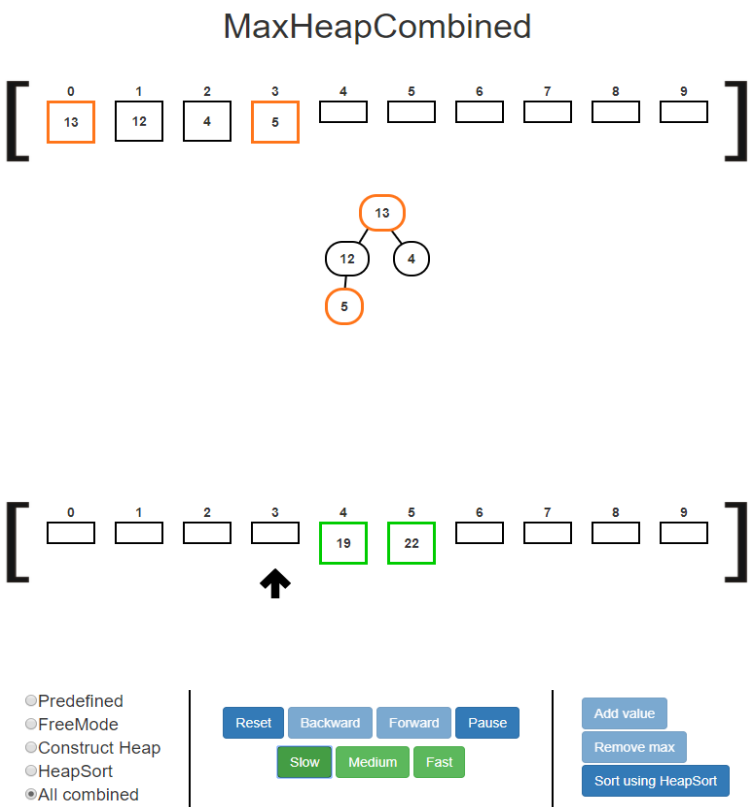
1. Bytt plass på roten og det siste elementet
2. Kopier maksimum-verdien (nå den siste i heapet) til nåværende indeks i sorterings-tabellen (indikert med pil)
3. Slett det tidligere rot-elementet fra heapet
4. Siden heap-invarianten nå kanskje er brutt må det nye rot-elementet «Synke» til riktig plass.
    a. Sammenlign roten med barne-elementene, hvis de har riktig rekkefølge – stopp
    b. Hvis ikke, bytt plass med det største barnet og gjenta a-b til heap-invarianten er oppfylt
5. Gjenta steg 1-4 til heap-en er tom

Kompleksiteten av algoritmen er: O(n log n)

- N-bytter og for hvert bytte må den nye roten synke maksimalt log n ganger

6.6 All combined – De tidligere visualiseringene kombinert til en felles visualisering

Slik ser visualiseringen ut under kjøring:



Figur 12 All combined

Denne visualiseringen gir brukeren mulighet til følgende:

1. **Legge til nytt element**
    a. Legger det nye elementet til sist i det binære treet
2. **Fjerne rot-elementet som spesifisert i 5.2**
3. **Konvertere til Heap og sortere ved å tryke på «Sort using HeapSort»**
    a. Konvertere treet til et maksimum Heap med algoritmen som er spesifisert i 5.4
    b. Utføre HeapSort-algoritmen som er spesifisert nærmere i 5.5

Kompleksiteten av algoritmen er: O(n log n).

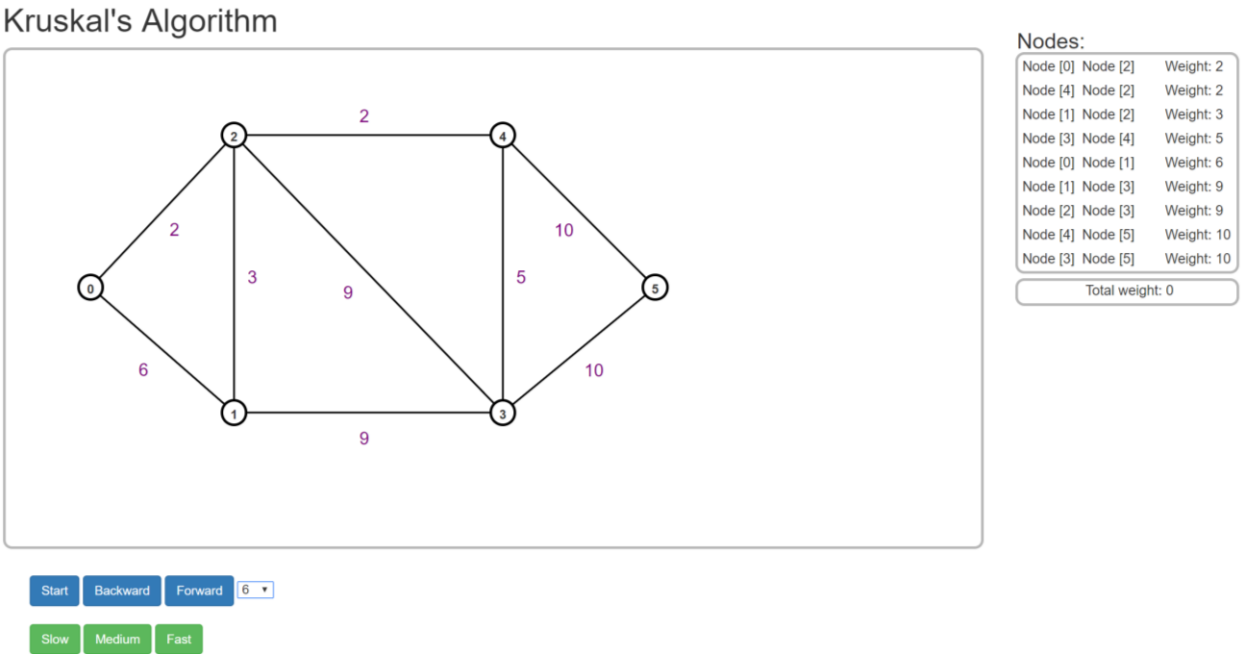
7 Kruskal (Minimum Spanning Tree)

Kruskals algoritme er en metode for å finne kantene som har minst mulig total vekt som kobler sammen alle noder/trær for å danne et så kalt minimalt utspennende tre. Den sorterer kantene etter vekt og velger den kanten med minst vekt fra listen. Deretter kobler algoritmen nodene sammen gjennom denne kanten for å danne et tre. Dersom den ser at kobling av to noder vil danne en syklus, vil algoritmen unngå dette og kaste bort kanten. Algoritmen vil gå gjennom hele listen frem til den har koblet alle nodene og har dannet et minimal utspennende tre. Kruskals algoritme er altså en grådig algoritme siden den bare plukker ut kantene med minst vekt for å kunne danne et minimalt utspennende tre, og den gjennomsnittlige kompleksiteten til Kruskals algoritme er O(|E| log |E|) hvor E er antall kanter.

7.1 Brukergrensesnitt

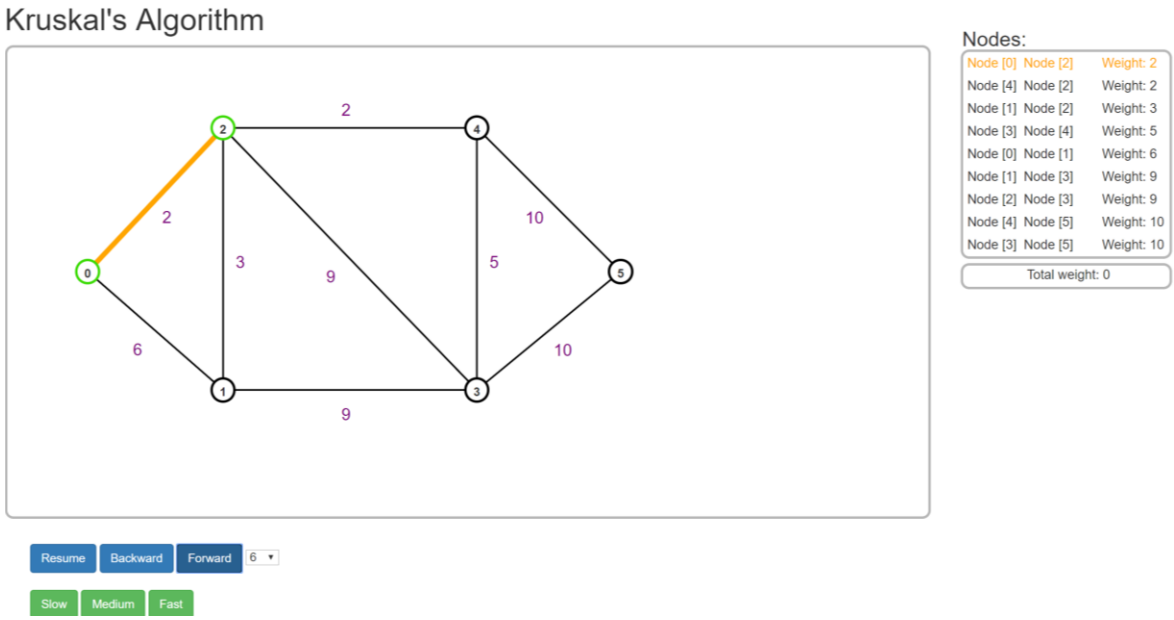
Menyen er som vist i Meny, men i tillegg kan brukeren velge tre til ti noder som skal danne en graf. Nodene vil også bli vist i tabellen med tilhørende kant. Tabellen viser også den totale vekten som inneholder alle kantene som har blitt lagt til treet.

7.2 Kjøring av algoritmen



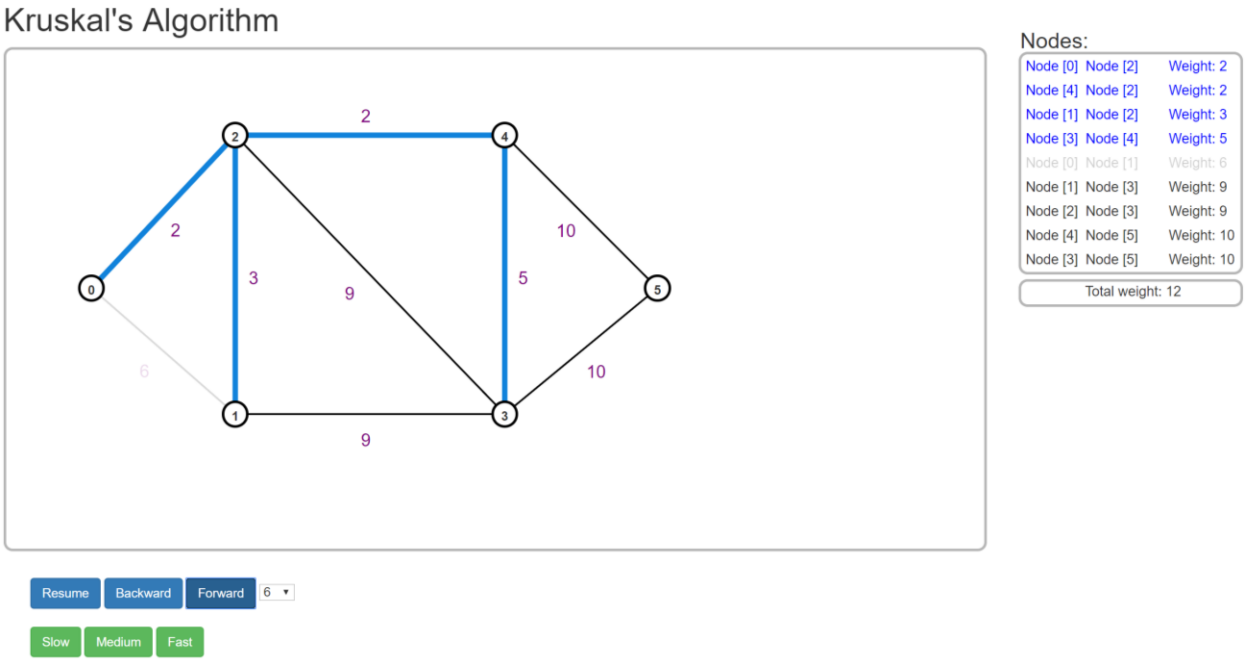
Figur 13 Oversikt over Kruskals algoritme og menyen

Visualisering av Kruskals algoritme innebærer visning av selve grafen, tabell over noder og menyen for å bl.a. starte algoritmen, velge hastighet samt velge hvor mange noder grafen skal ha. Algoritmen henter alle kantene og sortere de etter minst vekt, og den totale vekten vil først starte på null og deretter legges på etter hvert som algoritmen velger ut en kant.



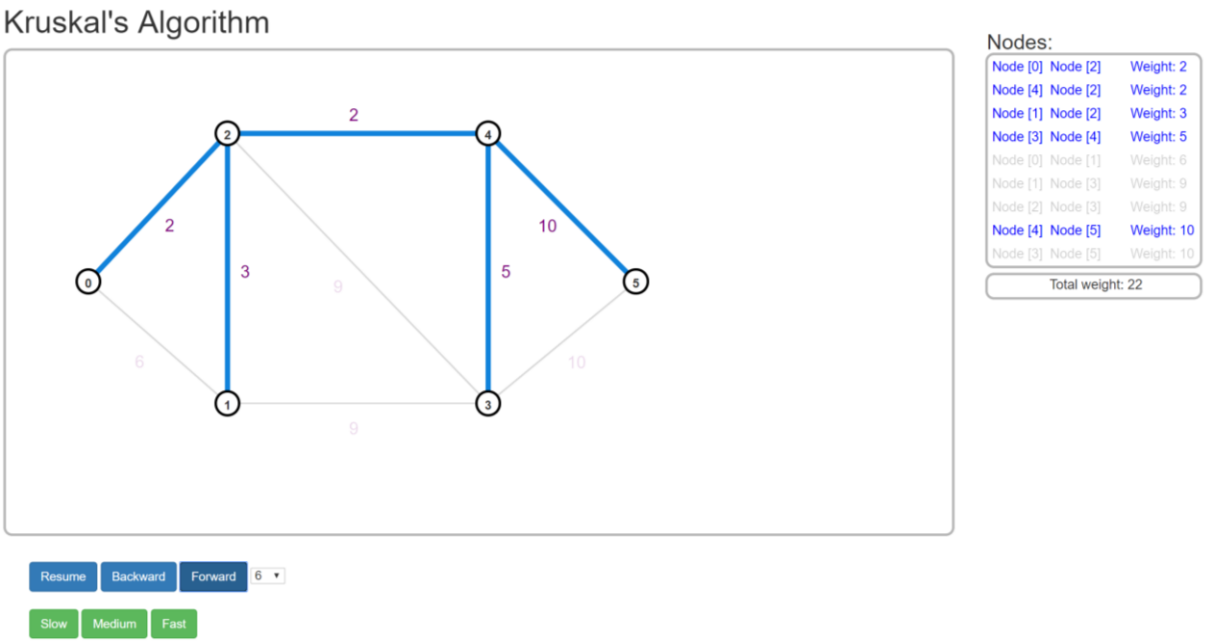
Figur 14 Algoritmen velger en kant

Algoritmen vil i første omgang plukke ut kanten med minst vekt og koble sammen de tilhørende nodene til kanten. Kanten og teksten vil bli markert oransje når algoritmen velger en kant.



Figur 15 Algoritmen har lagt kanter til i treet og ekskludert én kant

Kantene vil bli markert blå dersom de har blitt lagt til i det minimalt utspennende treet. Dersom en kant danner en syklus vil den bli transparent for å vise at den er blitt ekskludert fra treet.



Figur 16 Minste utspennende treet med den totale vekten

Algoritmen er ferdig med å kjøre dersom den har funnet alle kantene og nodene som utgjør et minimalt utspennende tre (N-1 kanter). Den totale vekten av treet vil vises under listen av kanter.

7.3 Motivasjon bak visualiseringen

Algoritmen er visualisert på denne måten slik at brukeren skal tydelig kunne se hvilken kant og hvilke noder som blir valg under kjøringen av algoritmen. Fargene viser tydelig forskjellig på hvilken del av algoritmen som blir vist og tabellen er der for å vise hvilke kanter som blir lagt til i summen av den totale vekten. Vi har også valgt å gjøre kantene transparent i stedet for å fjerne de fra grafen slik at man skal kunne se kantene som ikke har blitt lagt til.

8 Merge sort

Kompleksiteten er: O(n log n)

Merge sort er en rekursiv algoritme for sortering av tall som bruker «splitt og hersk»-teknikk. Denne har vi valgt å vise med top-down implementasjon. Selve algoritmen kjører som følger:

1. Del den usorterte listen i n-underlister, som hver inneholder 1 element (en liste med ett element anses å være sortert).
2. Slå sammen underlistene for å produsere nye sorterte underlister til det bare er en liste igjen. Dette blir den sorterte listen.

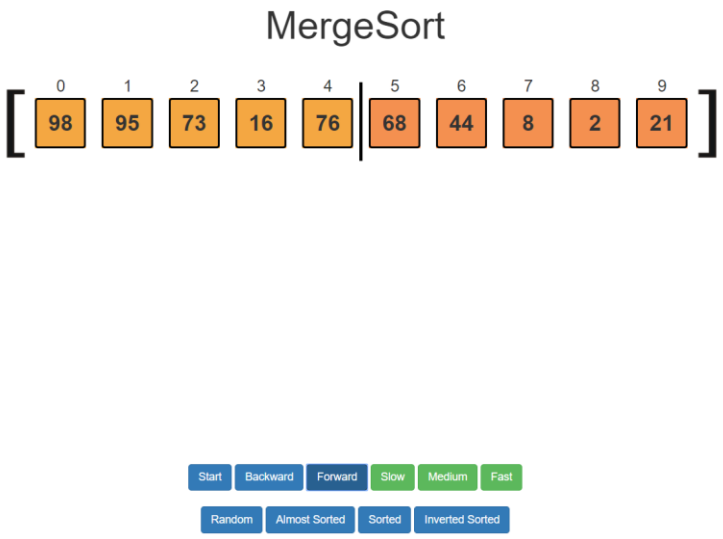
Vi har også implementert optimaliseringen som brukes dersom den ene tabellen er tom. Dette betyr at algoritmen vil flytte resten av den andre tabellen på rett plass dersom han ikke har flere tall han kan sammenligne med.

8.1 Brukergrensesnitt

Visualiseringen har knappene beskrevet i kapittelet Meny, men har også fått inn ekstra knapper slik at brukeren kan velge hvordan tabellen som algoritmen får ser ut. Her kan brukeren velge mellom:

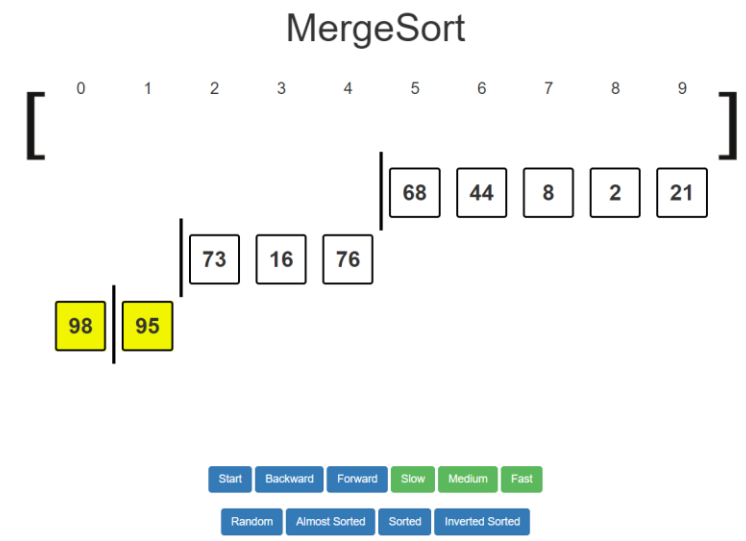
1. Tilfeldig – genererer en tilfeldig tabell
2. Nesten sortert – genererer en sortert tabell der noen tall har blitt flyttet på
3. Sortert – genererer en sortert tabell
4. Omvendt sortert – genererer en sortert tabell som blir invertert

8.2 Kjøring av algoritmen



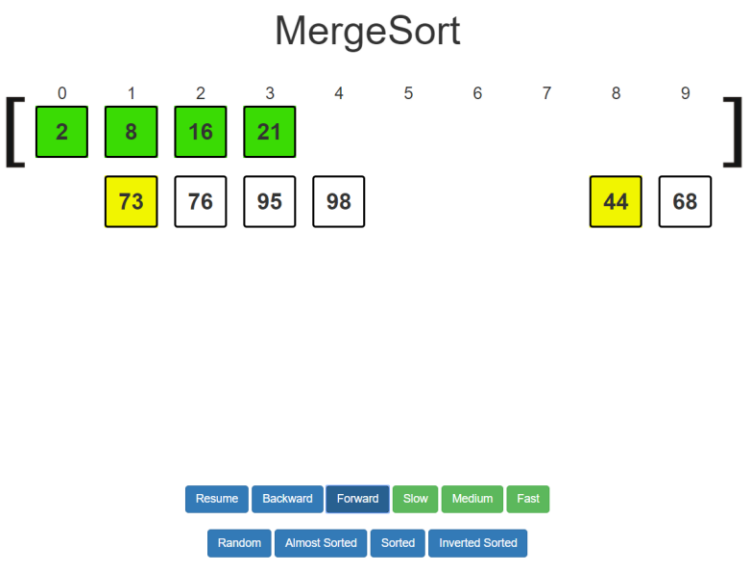
Figur 17 Splitting av tabell

Når du trykker på start vil algoritmen vise at den splitter opp i nye tabeller, ved å skifte farge og flytte elementene ned. Når den har gjort dette frem til det er to enkle elementer igjen, vil den kalle merge().



Figur 18 Første merge

Her har algoritmen kommet til der det er to enkle tall og da kaller den merge slik at de blir sortert. Den vil nå gjøre det samme med resten.



Figur 19 Siste merge

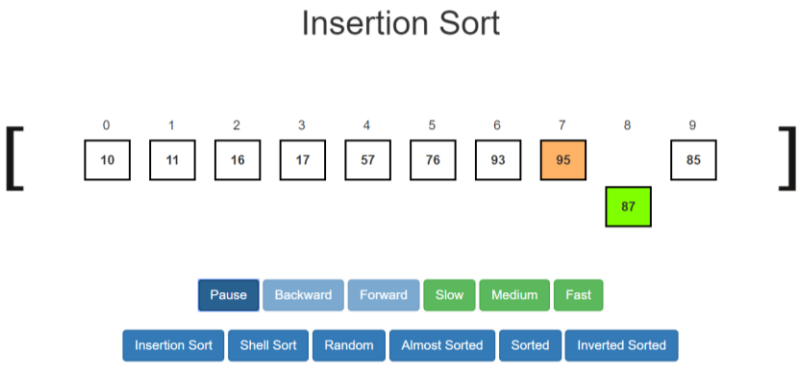
Nå er den på siste merge og du kan se at de som er grønne ligger på rett plass. De som er gule er de som blir sammenlignet akkurat nå. Dette betyr at 44 vil bli grønn og så stille seg ved siden av 21 i det neste steget.

9 Port av eksisterende algoritmer/Prosjekt

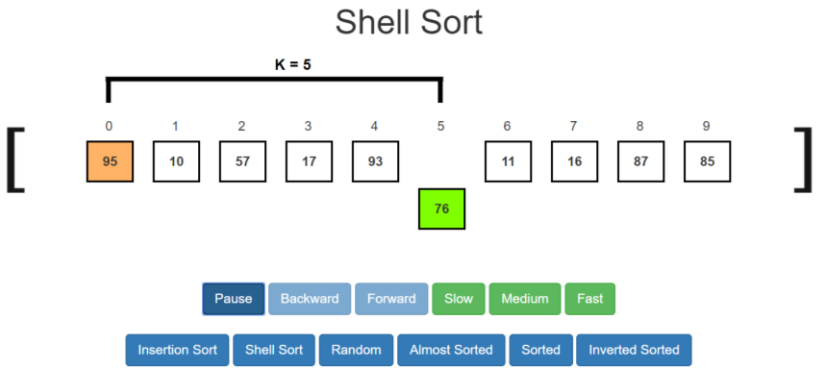
Første del av prosjektet gikk ut på å konvertere det tidligere 219-prosjektet til webteknologi slik at dette kunne bli en del av webplattformen vi skulle lage. Utfordringen med denne delen av prosjektet var hovedsakelig å sette seg inn i et gammelt system, forstå hvordan det systemet virket og så skrive det om til TypeScript. Januar og februar ble satt av til å konvertere den gamle koden siden vi måtte lære oss HTML, CSS, TypeScript og hadde vanskeligheter med å forstå det gamle systemet. I tillegg introduserte den nye løsningen bugs som ble fikset senere i prosjektet. QuickSort algoritmen ville et medlem av det tidligere 219 prosjektet gjennomføre selv (Knut Anders Stokke). Dermed ble ikke QuickSort en del av vårt prosjekt. Vi har også lagt til litt ekstra funksjonalitet og fikset noen bugs som var tilstede i det gamle visualiserings-prosjektet.

9.1 SimpleSort

I SimpleSort har vi lagt til nye hastigheter og fikset problemer som oppstod da vi skrev om fra Java, som for eksempel problemer med resetting av algoritmen. Vi har også gjort menyen lik de andre menyene, og lagt til en View klasse som er mellomleddet mellom brukergrensesnittet og backend.

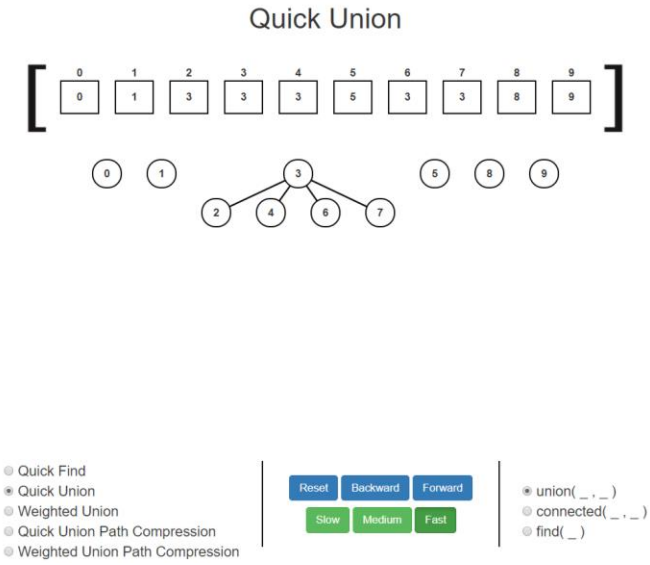


Figur 20 Insertion Sort



Figur 21 Shell Sort

9.2 Union Find

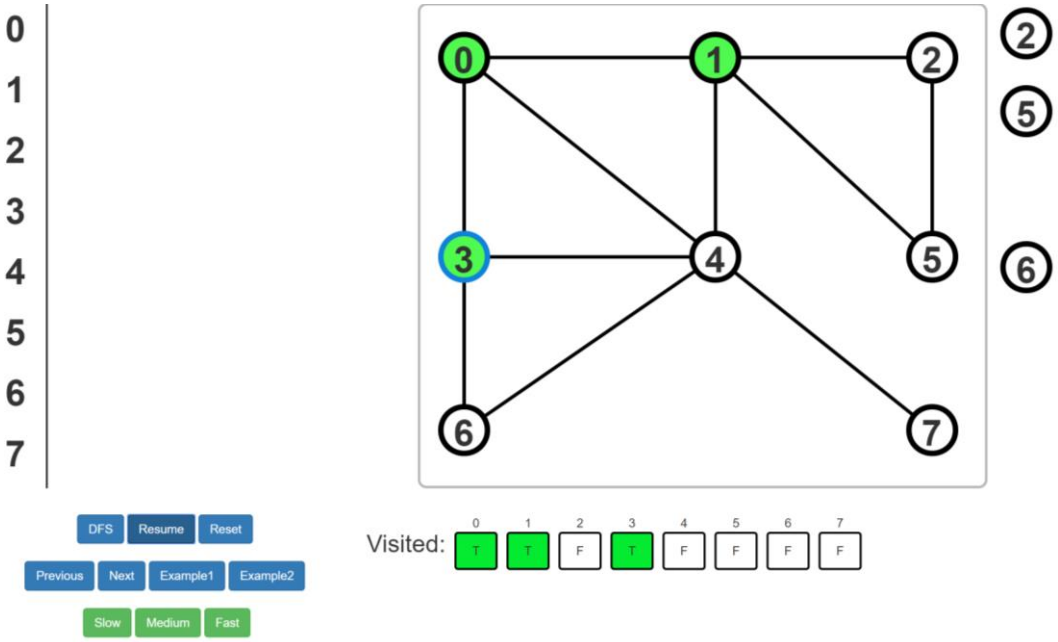


Figur 22 Union Find

Når vi konverterte prosjektet til TypeScript introduserte dette store problemer med mye av front-end funksjonaliteten. Det ble derfor brukt mye ressurser på å skrive om front-end funksjoner slik at de oppførte seg som forventet. En av problemstillingene vi syntes var ekstra utfordrende var bruken av JSON parametere i det originale prosjektet. I det originale prosjektet brukte de JSON til å utveksle informasjon om tabeller og hvilke noder som var koblet sammen mellom Java og Javascript. Dette inkluderte blant annet å lagre «States/tilstander» slik at man kunne gå fremover og bakover mellom tidligere tilstander. Vi tenkte i starten at implementasjonen virket elegant og at det kom til å være nokså greit å få det til i TypeScript, men vi oppdaget etter hvert at dette ikke var tilfellet. Først og fremst viste det seg at TypeScript og Javascript manglet et skikkelig JSON-bibliotek og vi måtte derfor implementere enkelte funksjoner som ble kalt på i det originale prosjektet selv. I tillegg til dette var det ingen på gruppen som hadde erfaring med JSON fra før og vi brukte derfor ekstra mye tid på å forstå implementasjonen og feilsøke den. Til slutt endte vi opp med en løsning som fungerer likt den i det originale prosjektet og selv om vi kunne ha gjort dette mye lettere med vanlige tabeller har prosessen vært veldig lærerik.

9.3 Graf - DFS/BFS

DFS og BFS hadde noen problemer før vi gjorde prosjektet om til frontend, og i tillegg kom det nye problemer. De verste problemene skal være fikset slik at det nå er lettere å bruke denne algoritmen.



Figur 23 DFS/BFS: BFS under kjøring

10 Retrospektiv

10.1 Hva ville vi gjort annerledes

- Vi ville valgt å bruke tabeller fremfor JSON i porten av Union Find. Selv om dette var lærerikt ble det mer ressurskrevende enn det burde vært når vi kunne oppnådd samme funksjonalitet med vanlige tabeller.
- Vi ville sørget for å ha bedre struktur på utviklingsprosessen vår fra starten av. Hvis vi hadde hatt strengere prosess med faste møtetider og bedre kommunikasjon kunne vi unngått skjev arbeidsfordeling de første ukene. I tillegg kunne vi kanskje også satt oss raskere inn i det gamle 219 prosjektet.
- Vi burde ha mer kommunikasjon med den tidligere gruppen slik at de kunne fortelle om deres tidligere prosjekt og hjulpet oss med å sette oss inn i prosjektet. Som nevnt tidligere hadde vi problemer med å skrive om Java til TypeScript på grunn av manglende erfaring med webutvikling. Dersom vi hadde snakket med den tidligere gruppen hadde dette gitt oss en bedre forståelse av verktøy og prosjektet deres, samt hadde det minimert tiden vår i å konvertere algoritmene.

10.2 Læringsutbytte

Gjennom prosjektet har vi fått mange nye erfaringer og tilknyttet oss mye ny kunnskap. Ingen av oss hadde erfaring med webutvikling fra tidligere og under prosjektet har vi gradvis blitt mer og mer komfortabel med å skrive webapplikasjoner i HTML, CSS og Typescript. Vi fikk også erfare hvordan det er å sette seg inn i et eldre system når vi skulle konvertere det gamle 219 prosjektet til webben. Her ble det tydelig hvor viktig god kode-struktur, navngivning av variabler og dokumentasjon er når man skal vedlikeholde et eldre system. I starten av prosjektet fikk vi kjent litt på hvordan vi håndterer problemer med utviklingsprosesser og gruppearbeid, og hvordan vi kan løse disse.

11 Konklusjon

Resultatet av visualiseringsprosjektet er et undervisningsverktøy med seks moduler som kan benyttes av forelesere og studenter i introduksjonsfaget for Algoritmer og Datastrukturer ved universiteter og høgskoler. De seks modulene er

1. Heap
2. Kruskal
3. Merge Sort
4. Insertion Sort / Shell sort
5. Union Find
6. DFS og BFS