**Computer Science**

---

**Camilla Fransson**

# Animering av algoritmer

---

Bachelor’s Project

2002:25

**Computer Science**

---

**Camilla Fransson**

# Animation of Algorithms

---

Bachelor’s Project

2002:25

This report is submitted in partial fulfilment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Camilla Fransson

Approved, 2002-06-05

Advisor: Donald F. Ross

Examiner: Donald F. Ross

## Sammanfattning

Detta projekt gjordes för att underlätta förståelsen för de algoritmer och datastrukturer som ingår i kursen Datastrukturer och algoritmer. Detta skulle göras grafiskt med hjälp av animeringar. De datastrukturer/algoritmer som användes var sorteringsalgoritmer, hashtabeller, träd och grafer. För implementationen användes programmeringsspråket Java.

Animering går ut på att skapa rörliga bilder. Detta kan ske på framför allt tre olika sätt:

Det första sättet används ofta när man tillverkar tecknad film och bygger på en slinga som håller reda på vilken bild som står i tur för att visas. Sedan ritas den aktuella vyn upp i ritmetoden paint().
Det andra sättet är att helt enkelt flytta ett objekt på skärmen. Om positionen för ett objekt ändras med jämna mellanrum kommer det att se ut som att objektet rör sig. Förändringen sker i en slinga, det vill säga att den nya positionen för objektet räknas ut och själva visningen sker i metoden paint().
Det tredje sättet som är det mest traditionella, är att visa flera riktiga bilder i snabb följd för att på så sätt ge sken av att bilderna rör sig. Även här väljs rätt bilder ut i en slinga för att sedan ritas ut i metoden paint().

Efter försök att med att spara siffror för sortering som bilder visade det sig att det andra sättet att förändra positionen för ett objekt genom beräkningar var det som fungerade bäst.

I detta projekt har animering för bubblesort och quicksort implementerats och designen gjorts för de övriga algoritmerna som förslag för fortsatt arbete.

## Abstract

This project was produced for the purpose to facilitate the understanding of the algorithms and data structures that is included in the course Data Structures and Algorithms. This should be shown in graphic by using animations.
The data structures and algorithms that were used were sorting algorithms, hash tables, trees and graphs. The programming language Java was used for the implementation.

Animation is achieved by creating mobile pictures. There are three different ways to do that:

The first way is used when cartoons are created and are built using a loop that is responsible for showing the next picture. Then is the current picture drawn in a drawing method called paint().
The second alternative is just to move an object on the screen. If the position of an object is changed at regular intervals will it look as if the object moves. The changes are done in a loop, the new position are calculated an then shown using the method paint():
The third way that is the most traditional is to show many real pictures in quick succession so that the pictures appear to move. Even here the right pictures are put in a loop and drawn using the method paint().

After repeated tries to save the numbers that should be used in the sorting process as pictures, I discovered that the best way to implement the animation was to change the object's position by repeated calculations as in the second method.

The animations for bubble sort and quick sort have been implemented and a design presented for the remaining algorithms.

# Innehållsförteckning

# Figurer

# 1 Introduktion

Detta examensarbete går ut på att genom att med hjälp av animeringar göra de datastrukturer och algoritmer som ingår i kursen Datastrukturer och Algoritmer mer lättförståeliga. Då dessa algoritmer kan vara svåra att förstå vill man genom att visa hur dessa fungerar grafiskt underlätta förståelsen samt göra dem mer begripliga. Det är lättare att förstå om man visuellt visar hur en algoritm fungerar med animering istället för att sätta sig in i kod.

De områden som tas upp är:

- Sorteringsalgoritmer

  Bubblesort, Selectionsort, Insertionsort, Shellsort, Mergesort, Quicksort
- Hashtabeller
- Träd
- Grafer

I kapitel 2 görs en bakgrundsundersökning av problemet. Där tas även upp andra system som har hittats. En enkel beskrivning av de algoritmer som ska inleda projektet nämligen sorteringsalgoritmerna bubblesort och quicksort samt en omnämning av de övriga algoritmerna som skall göras i mån av tid.

I kapitel 3 diskuteras design med bla skärmlayout, text, bilder och knappar. De olika designaspekterna så som färger, bilder osv diskuteras i detalj. Sedan följer designbilder för de algoritmer som ska implementeras.

I följande kapitel dvs kapitel 4 tas de olika implementationsteknikerna upp som tex trådar, dubbelbuffring och animering.

I kapitel 5 diskuteras de resultat som fåtts och här visas hur animeringen visas på skärmen genom bilder och pilar. Sedan följer en genomgång av de algoritmer som inte hunnits implementeras .

Detta sista kapitlet 6 består av en slutsats av projektet samt tips för fortsatt arbete med animering av datastrukturer och algoritmer.

# 2 Bakgrund

## 2.1 Introduktion

Denna rapport görs för att med hjälp av grafik förtydliga olika algoritmers tillvägagångssätt för elever som läser kursen Datastrukturer och algoritmer. Då man ser koden till de algoritmer som ingår i kursen ser man genast att de inte alltid är så lätta att förstå. Genom att visuellt visa dessa algoritmer ska förståelsen underlätta för dessa.

Det finns flera olika aspekter som ingår i projektet:

- De tekniska aspekterna: Vilket språk som är lämpligast att programmera i. Jag har valt att använda mig av JAVA då jag tycker detta fungerar bra när man ska använda sig av grafik och sedan visa det på nätet.

- De designtekniska aspekterna: Hur ska man på ett bra, lättfattligt och samtidigt snyggt sätt kunna visa de olika algoritmerna grafiskt?

- De pedagogiska aspekterna: Vad kan man göra för att göra algoritmerna mer lättförståeliga? På vilket sätt ska den grafiska utformningen vara för att underlätta förståelsen av algoritmerna.

## 2.2 Andra system

I undersökningarna som gjordes på Internet hittades många intressanta animeringar. De flesta var dock animeringar som hanterar bilder som tex vykort med rörlig text och rörliga bilder.

Det förekom även många olika spel som tex det sk femton-spelet som innebär att man ska flytta rutorna med numren i tills alla nummer står i rätt ordning och den tomma rutan finns längst ner i det högra hörnet. Den ruta man klickar på tar den tomma rutans plats och vice versa.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | | 15 |
| 13 | 11 | 14 | 12 |

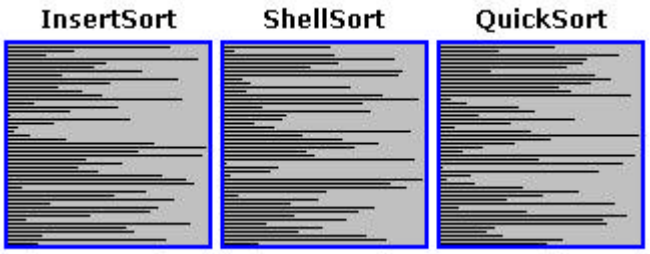
*Figur 2-1: Femton spelet*

Det finns många varianter på detta spel som tex att de olika rutorna är fyllda med olika sorters färger som ska placeras enligt ett speciellt mönster. Variationerna är många och kvalitéen varierar enormt.

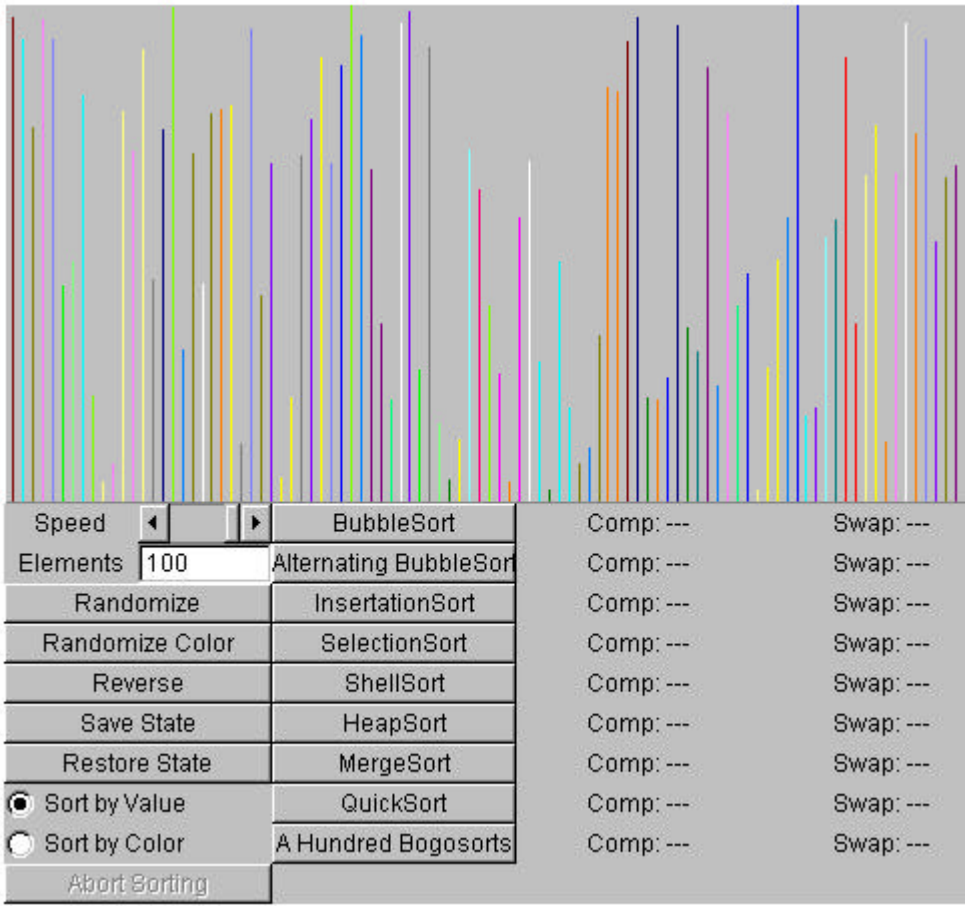
Det som man främst kan bedöma kvalitén på är om det som gjorts verkligen fungerar på det sätt som beskrivs. En bedömningspunkt kan också vara att det ska vara lätt att förstå hur man får igång animeringen om den inte startas automatiskt samt information om hur de olika knapparna och menyerna fungerar. En viktig aspekt är om budskapet i det man ser går fram eller om det blir rörigt och svår förståeligt.

Det finns ett mycket stort utbud på olika spel och man upphör aldrig att förvånas. Det är dock mycket sällan man får tillgång till någon källkod vilket är synd då det skulle vara mycket intressant att följa de olika stegen i animeringen. Där det fanns källkoder visade det sig ofta att den var svår att förstå och det var få som hade kommentarer i koden något som skulle ha uppskattats.

När det gäller sorteringsalgoritmer var utbudet rejält begränsat och de få som fanns var ofta lite svåra att förstå när man såg animeringen. Det var genomgripande väldigt många argument som sorterades vilket gjorde det svårt att hänga med. De visades med hjälp av staplar och i vissa fall kunde man sortera via storlek på staplarna eller genom vilka färger staplarna hade [7, 8].




*Figur 2-2: Sorterings exempel - efter Niemann [7]*

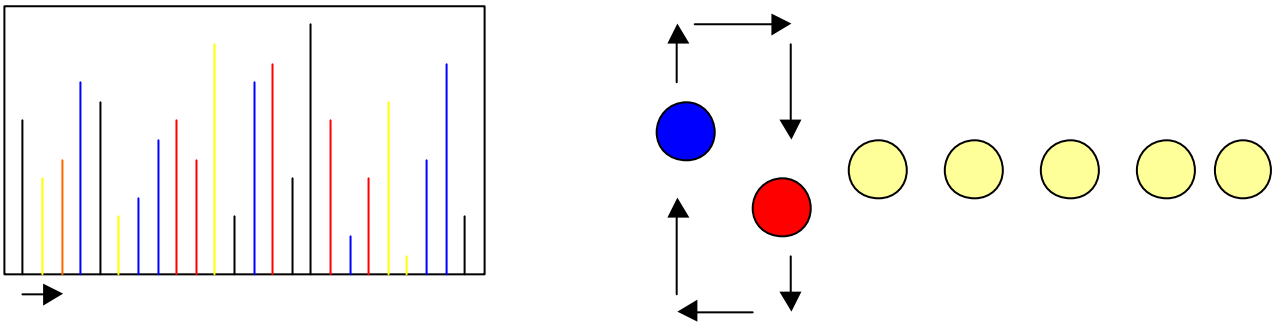



*Figur 2-3: Sorterings exempel - efter Johansson [8]*

De flesta sorteringsanimationer gick ut på att visa antal byten och skillnaden i tid mellan de olika algoritmerna.

Det man har lärt sig genom att titta på andras animeringar är framför allt att göra det så enkelt och lättförståeligt som möjlig. Att ha en beskrivning av det som sker i programmet är bra så att man hela tiden kan hänga med.

Om man jämför de staplar som visades i sorterings animationerna på nätet så tycker jag att det var svårt att se hur själva sorteringen fungerade. Det var svårt att se vilka som jämfördes när det var mycket olika färger. Det som var bra var att man kunde ställa in hur fort man ville att sorteringen skulle gå och hur många element som skulle sorteras.



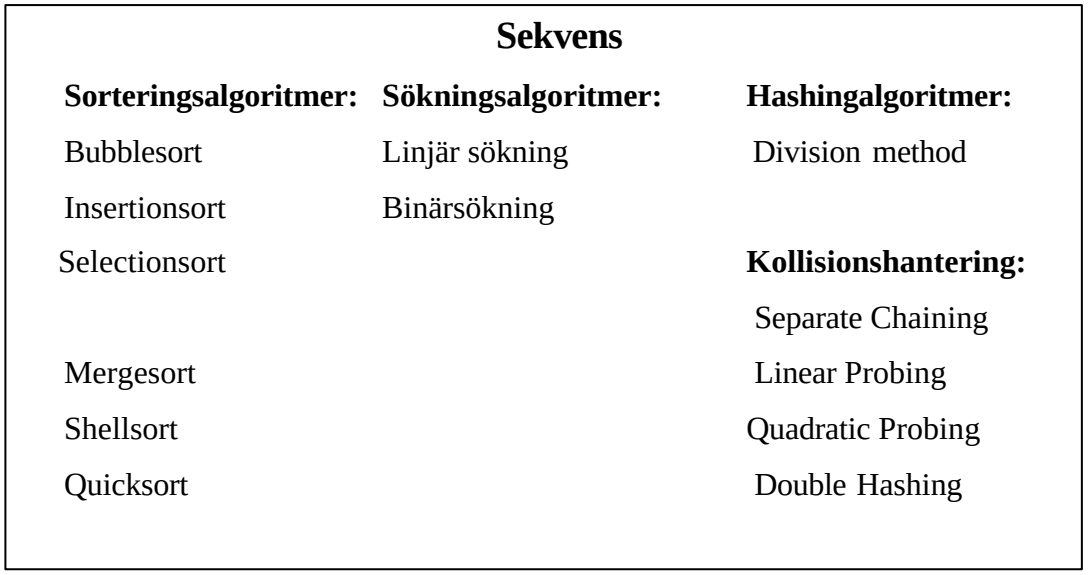
*Figur 2-4: Jämförelse sorterings animeringar*

Om man jämför de olika figurerna ovan så är det lättare att i animeringen med cirklarna, som är den design som använts i projektet, följa varje steg i algoritmen. Cirklarna ändrar först färg för att visa vilka som jämförs och sedan byter de plats genom att förflytta sig uppåt/nedåt till höger/vänster och sedan uppåt/nedåt till sin nya plats. Detta upprepas tills sekvensen är sorterad. I animeringen till vänster som visar staplarna förflyttas de utan att först vara markerade och bara byter plats med varandra vilket gör det svårt att följa algoritmens varje steg.

## 2.3 Översikt av algoritmerna

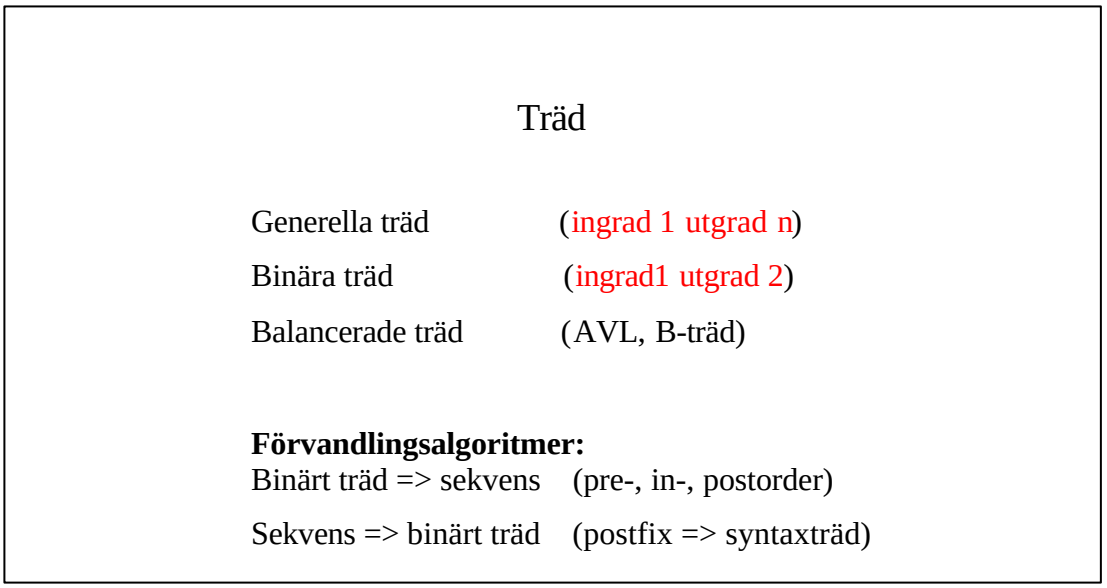
En algoritm är en väl definierad procedur som tillsammans med en specificerad mängd tillåtna inputs, producerar ett värde eller en mängd värden som output[3].

Inom kursen finns det 3 huvuddatastrukturer: **sekvens, träd, grafer** och deras algoritmer.



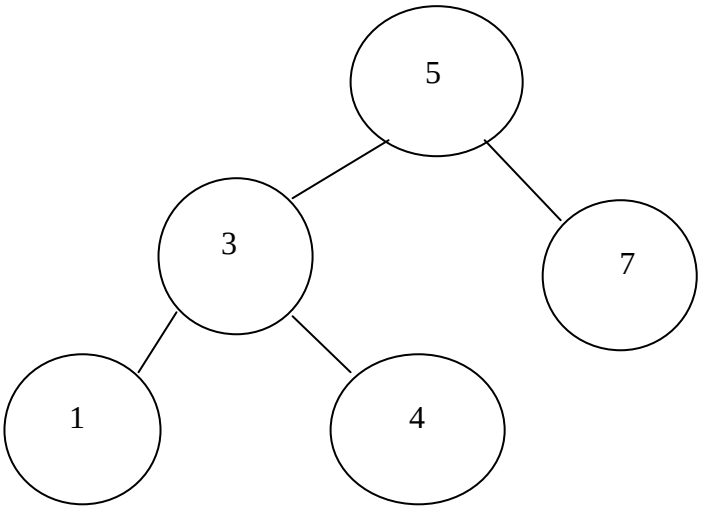

*Figur 2-6: Sekvens algoritmer*

För att illustrera sorteringsalgoritmer använder man slumpade input. Sortering går ut på att med olika sorterings algoritmer ändra om i en serie av värden så att de sätts i stigande eller fallande ordning. Dessa algoritmer är ”jämförelse baserade” dvs det som styr var värdet ska hamna i den sorterade sekvensen avgörs genom att värdet jämförs med ett annat värde som finns i samma sekvens. Olika sorteringsalgoritmer används till olika sorters sorteringar. Vilken som används beror ofta på vilken sekvens värden som ska sorteras.




*Figur 2-7: Träd algoritmer*

Ett binärt träd är ett träd som kan vara tomt, bestå av en nod eller av en nod med max 2 barn där ett barn är själv ett binärt träd. Den nod som är högst upp kallas rot.




*Figur 2-8: Binärt träd*

Dess höjd avgörs av hur många ”nivåer” trädet har. I detta fall är höjden 2 då man räknar från 0. Om datan i trädet lagras så att de värden som finns i det vänstra delträdet är mindre än roten och värdena till höger är större än roten kallas det ett binärt sökträd.

**Grafer**

| **Dirigerade grafer** | **Odirigerade grafer** |
|---|---|
| Dijkstras | Prims |
| Floyds | Kruskals |
| Warshalls | |

Depth/Breadth first search, Cycle Detection

**Specifika tillämpningar**:

| | |
|---|---|
| Connected Components | Never never land |
| Articulation Points | Travelling salesman problem |

*Figur 2-9: Graf algoritmer*

Grafer kan användas för att representera och studera transporterande nätverk, kommunicerande nätverk, parallella datastrukturer och elektriska kretsar.

## 2.4 Exempel på algoritmer

Jag har valt dessa två exempel för att visa två helt olika sorters sorteringar. Bubblesort sorterar iterativt medan quicksort sorterar rekursivt.

### 2.4.1 Bubble Sort

Denna sorteringsalgoritm går ut på att föra stora värden till höger i sekvensen och små värden till vänster av sekvensen. En utförlig förklaring visas nedan.

1. Utför en iteration från första index till näst sista index. För varje indexvärde i iterationen skall aktuell position jämföras med nästa position. Om nästa position (i förhållande till aktuell position) har ett lägre värde än aktuell position skall positionernas värden byta plats. När iterationen är klar vet man att största värdet finns i slutet av sekvensen.
2. Om någon ändring har skett under föregående iteration skall förfarandet utföras en gång till. Om inte någon ändring har skett betyder det att sekvensen är sorterad. För varje gång man utför iterationen på nytt kan man minska slutindex med ett, därför att det största värdet "flyter med" hela vägen till höger [se bilaga A].

### 2.4.2 Quicksort

Väljer ut ett strategiskt pivotelement och placerar alla element mindre än pivotelementet i början av sekvensen och alla element större än pivotelementet i slutet på sekvensen. Sedan anropas QuickSort rekursivt för början av sekvensen och för slutet av sekvensen[se bilaga A].

## 2.5 Återstående algoritmer

De återstående algoritmerna är framför allt träd samt grafer.

### 2.5.1 Binära träd

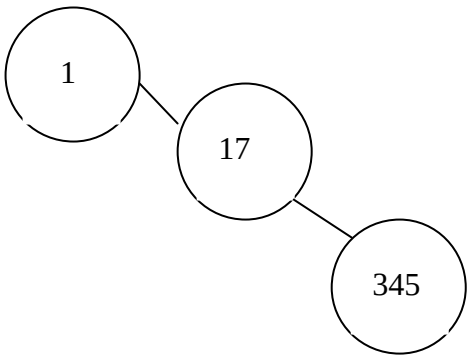
När det finns små tal till vänster om noden och stora tal till höger om noden har man ett *binärt sökträd*, så kallat eftersom det går så snabbt att söka reda på ett objekt i trädet. Låt oss säga att vi söker efter 345. Vår algoritm blir följande

- Kolla först rottalet.
- Om talet är 345 har vi funnit vad vi söker.
- Om talet är större än 345 letar vi vidare efter 345 i vänsterträdet.
- Om det är mindre än 345 letar vi vidare i högerträdet.
- ...men om vi når en nullreferens finns inte 345 i sökträdet.

Det här är ju precis samma sak som binär sökning i en sekvens. I båda fallen blir antalet jämförelser cirka log *N*. Men binärträdet har två stora fördelar [3].

1. Insättning av nytt objekt kräver bara log *N* jämförelser mot *N*/2 för insortering i en vektor.
2. Trädet kan bli hur stort som helst men arrayens storlek ges i dess deklaration.

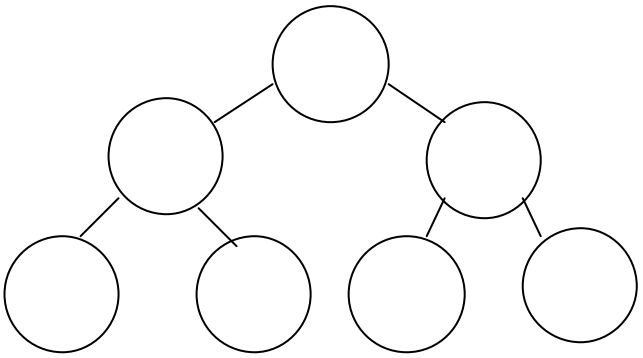
Enda problemet med binärträd är att dom kan bli *obalanserade*, och då försvinner den snabba sökningen. Ett riktigt obalanserat sökträd med dom tre talen i exemplet har 1 i roten, 17 till höger, 345 till höger om det osv. I vissa fall har binärträd en tendens att bli obalanserade med tiden, till exempel när nyfödda förs in i ett träd sorterat på personnummer och då alltid hamnar längst till höger. Och då har trädet blivit en sekvens.




*Figur 2-10: Obalanserat träd*

### 2.5.2 Balancerade träd

Ett balancerat träd är ett träd där balansfaktorn vid varje nod är antingen –1, 0 eller 1.



*Figur 2-11: Balanserat träd*

### 2.5.3 Travelling salesman problem

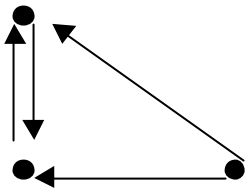
The traveling salesman problem bygger på att han ska besöka varje punkt(stad) i grafen endast en gång och ta den kortaste vägen tills alla punkter(städer) är besökta.

### 2.5.4 Grafer

En graf G = (V,E) där V är en mängd av noder och E är en mängd av kanter (u, w)
där u, w tillhör V. Om (u, w) är ordnad så är grafen riktad annars är grafen oriktad.

### 2.5.5 Riktade grafer

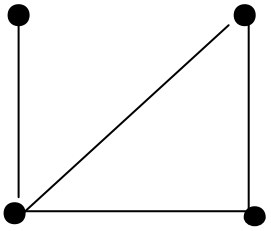
Riktade grafer är grafer där man har en bestämd riktning på varje kant. Man tänker sig att kanterna är "enkelriktade" så att man endast kan gå längs dem i en riktning. Dessa används ofta inom datorvetenskap för att analysera strukturer och data-flöden. När man sysslar med riktade grafer brukar man även tala om in- och ut-valenser, dvs. man skiljer på antalet kanter som går till och antalet kanter som går ut från en nod. . Figur 2-12 ger ett exempel av en riktad graf.



Figur 2-12: Riktad graf

### 2.5.6 **Oriktad graf**

I en oriktad graf är (u, w) som tillhör V oordnade. Figur 2-13 ger ett exempel av en oriktad graf.



*Figur 2-13: Oriktad graf*

### 2.5.7 **Never Never land**

Never Never land används för att finna den bästa resvägen mellan två städer. Man anger varifrån man vill åka, vart man ska och vilken tid man vill åka så räknas den bästa vägen ut och ritas upp på kartan nedan. Den använder sig av Floyds algoritm.


Figur 2-14: Never Never Land

## 2.6 Summering

Detta kapitel presenterar en del av de algoritmer som ingår i examensarbetet. Det innehåller också en beskrivning av dessa för att underlätta förståelsen för dessa. Det har även skrivits om de algoritmer som inte hunnits med i projektet.

## 2.7 Avgränsningar

Jag kommer att börja med att implementera bubblesort och quicksort. Sedan kommer övriga algoritmer att göras i mån av tid.

# 3 Design

## 3.1 Introduktion

Design är ett stort ämne och styrs av vad det man designar ska användas till, vilken målgrupp man vill nå och självklart ens egen smak. Det finns dock lite riktlinjer att hålla sig till [1].

1. Ge god överblick.

   En oöverskådlig sida leder till irritation och ofta lämnar användaren sidan kort därefter. På första sidan gäller det snabbt att förmedla vilken indelning som innehållet har och skapa intresse för vidare läsning.

   Man bör tänka på att användaren ofta är ovillig till att skrolla upp och ner för att hitta information.

2. Gör det enkelt att navigera.

   Eftersom Internet bygger mycket på hyperlänkar är navigeringsvänlighet viktigare än vanligt. På varje sida bör man ge en upplysning om var man befinner sig och hur man kommer vidare. Länkar och navigation blir en viktig del av dispositionen. När man programmerar består länkarna av tex knappar som startar en process eller dyl.

3. Konsistent gränssnitt förenklar.

   Den valda dispositionen skall helst vara likartad på angränsande sidor. Navigeringslänkar i vårt fall knapparna) bör t.ex. alltid vara placerade på samma ställe på sidan. Konsistens är viktigt för att man enkelt skall kunna navigera mellan sidor. Bakgrunder och typografi bör också vara liknande.

De olika aspekterna som ingår i designen är

- Bild och text
- Skärmlayout och meny
- Knappar

## 3.2 Text och bilder

Den bästa kontrastverkan för att läsa text är mörk text på ljus bakgrund eller svagt tonad vit. Om man väljer att använda färger så är rekommendationen ljus bakgrundsfärg och mörk textfärg.

Det är också viktigt att tänka på vad man vill att den person som använder den design man gjort i första hand ska rikta sin uppmärksamhet på. Om man skrivit en viktig text med svarta bokstäver på vit bakgrund och sätter en bild med färger bredvid så riktas automatiskt användarens uppmärksamhet på bilden som framträder starkare än texten. Detta är mycket viktigt att tänka just för att man ska få fram det syfte man hade med designen. Det är ju ofta som man tycker att en text ser lite tråkig ut och så fixar man till det med en bild som kanske egentligen inte har så mycket med texten att göra. Bilder är väldigt bra att använda för att förhöja meningen med tex en text för att visualisera det som skrivits, men då måste bilden vara relevant till texten.

### 3.2.1 Läsbarhet

Inom klassisk typografi kan man hävda att det finns lagar och regler för vad som gör en text läsbar. Den mesta litteraturen i ämnet är mycket samstämmig och reglerna är oftast giltiga även för text som publiceras på webben, för att läsas på en bildskärm [4]. Typografi bygger på kontraster och igenkännande, och god balans mellan dessa. Det är viktigt att texten kontrasterar mot bakgrunden. Lättast att läsa är mörk text på ljus, helst något matt, bakgrund [4]. Vidare är det lättare att läsa en text skriven med gemener, vilket framgår av exemplet nedan. Det beror på att man inte läser bokstav för bokstav utan hela ord eller ordbilder. Gemener har en mer varierad ordbild än versaler, varför man bör undvika att skriva text enbart med versaler.

gemener är lättare att läsa GEMENER ÄR LÄTTARE ATT LÄSA

Ett alltför frekvent användande av fet eller kursiverad stil förtar effekten och förstör textmassans enhetlighet. Likaså bör man var sparsam med att använda flera olika teckensnitt.

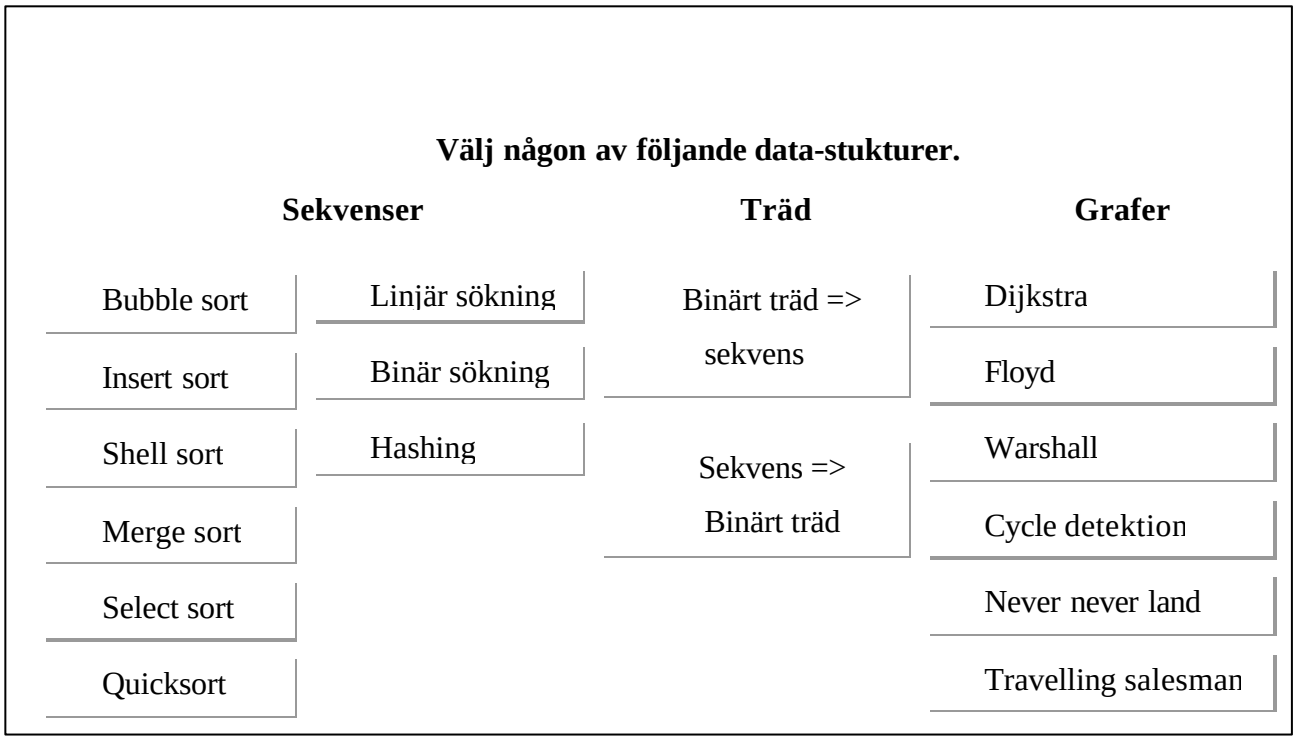
Radlängden inverkar också på läsbarheten. Rader som kan läsas utan att behöva ändra ögonens fixeringspunkt är mer lättlästa än långa rader där man tvingas byta flera gånger. Dock bör raderna inte var alltför korta så att den mesta energin går åt till att byta rad.

### 3.2.2 Teckensnitt

Det finns två huvudtyper bland teckensnitten: antikva t.ex. "animering" och sanserif, t.ex. "animering". För löpande text i tidningar och böcker lämpar sig antikvan bäst. Det beror dels på att vi är mest vana att läsa den typen av text och dels på bokstävernas utformning. Antikva tecken har serifer dvs klackar och flaggor. Klackarna leder ögat utefter textraden, vilket underlättar läsandet och flaggorna gör bokstäverna mer varierade än sanserif, vilket bidrar till att öka läsbarheten. Sanserif, tecken utan klackar och flaggor, är mer lämpade att använda till rubriker och i små teckengrader i figurer och tecken.

## 3.3 Skärmlayout och menyer

För att förenkla för användaren och göra allt klart och tydligt är detta det första man ser när man startar programmet. Här kan användaren välja den datastruktur han/hon är intresserad av genom att med musen klicka sig fram.




*Figur 3-1: Meny*

Detta är den meny som man först ser när man startar programmet. Den bygger på att knapparna som tryckts på leder vidare till den algoritm man vill se.

## 3.4 Knappar

För att skapa knappar i Java används klassen Button. Då knappar är en komponent med en sk etikett dvs det som ska stå på knappen så kan detta sättas och ändras fritt. I många språk anger man var knapparna ska placeras genom att ange pixlar, detta kan man också göra i Java. Men i Java finns också andra metoder att lägga ut komponenter. De kallas GridLayout, BorderLayout, CardLayout och GridbagLayout.

### 3.4.1 GridLayout

GridLayout anger att komponenterna ska ligga i ett osynligt rutnät. Det första argumentet till GridLayout anger hur många rader rutnätet ska innehålla och det andra antalet kolumner. De två sista argumenten anger hur långt det ska vara mellan argumenten i höjd och sidled. Exempel:

```
SetLayout(new GridLayout(1,2,5,5));
```

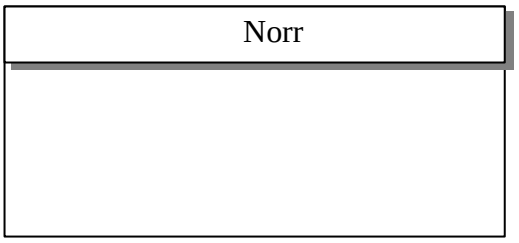
*argument* ↗

### 3.4.2 BorderLayout

BorderLayout arrangerar komponenterna längs med sidorna av det tillgängliga utrymmet. I en metod som heter add ska man sedan ange för varje komponent var den ska ligga, eller om den ska vara i mitten. Exempel:

```
SetLayout(new BorderLayout());
Button btnN = new Button(”Norr”);
Add(btnN, BorderLayout.NORTH);
```

Här har man skapat en knapp som ligger högst upp i rutan.




*Figur 3-2: BorderLayout*

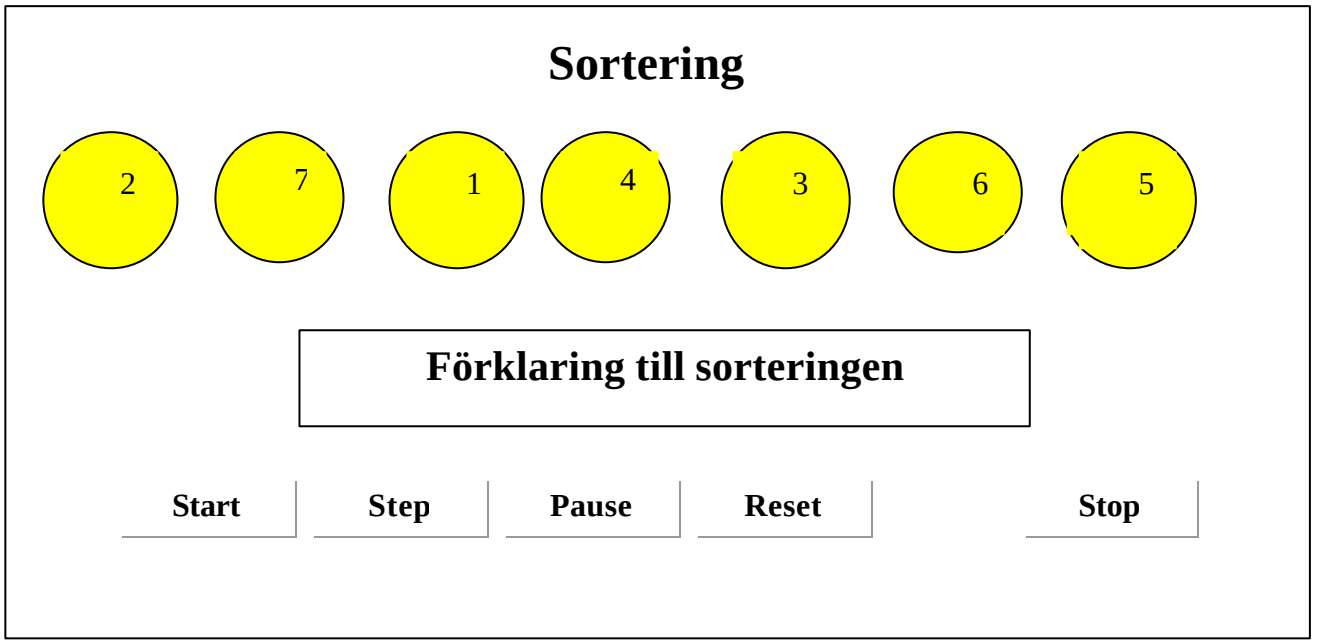
### 3.4.3 CardLayout och GridBagLayout

Med CardLayout kan man definiera flera ”kort” som täcker hela ytan och som alla kan ha varsin layout.

GridBagLayout lades till ganska sent i Javas utveckling och är ganska svår att använda. Den fungerar ungefär som GridLayout, fast rutorna behöver inte vara lika stora och en komponent kan fylla flera rutor.

Dessa är dock ovanliga metoder så jag går inte in på ytterligare detaljer här.

## 3.5 Sorterings algoritmer



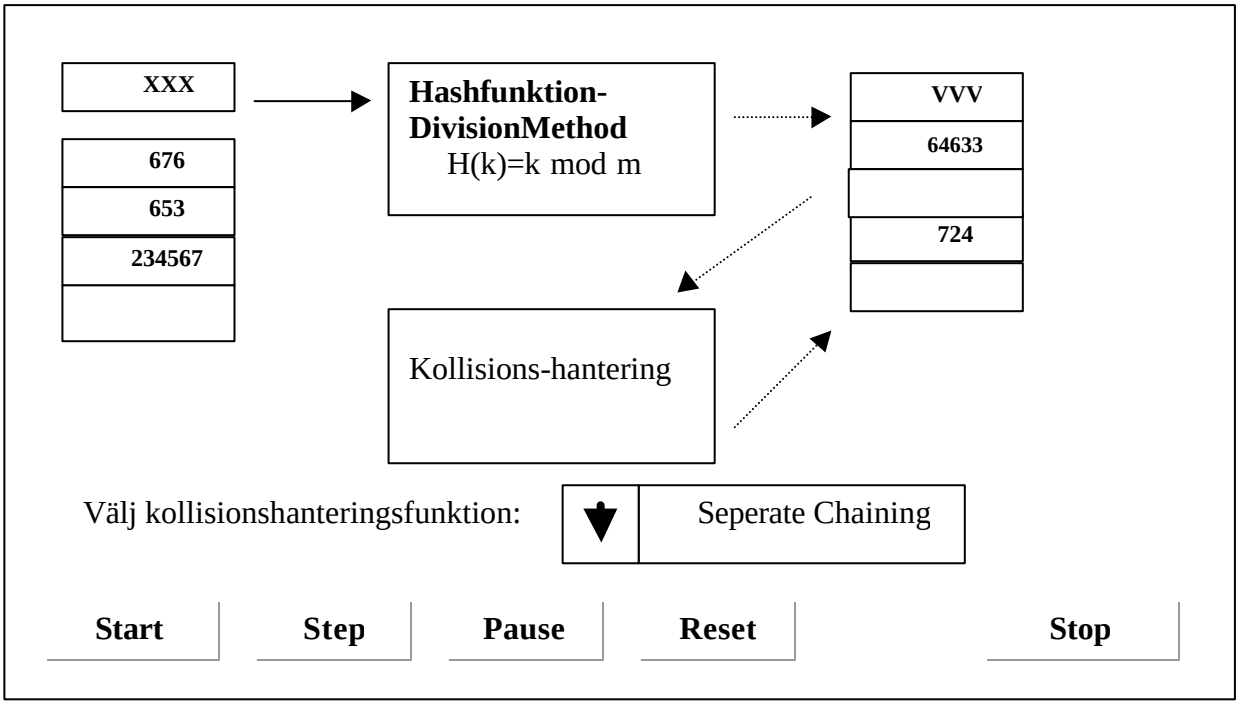

*Figur 3-3: Sorterings design*

Detta är den bild man får upp när man valt någon sortering från menyn i kap 3.3.

När man kommit till den algoritm man önskar så har man fått upp 7 nummer som ska sorteras. Dessa nummer har en randomfunktion i programmet slumpat fram och är därför olika för varje gång algoritmen körs. När man trycker på start börjar sorteringen genom att man ser de olika siffrorna byta plats. Detta pågår tills det är färdigsorterat eller man trycker stop, då kommer man tillbaka till menyn, eller tills man trycker pause, då stannar den upp i sorteringen och man kan fortsätta genom att trycka på pause igen. Om man vill styra sorteringen manuellt använder man knappen step och klickar sig fram steg för steg i sorteringen då visas även en förklaringsruta där information om varje steg visas. Om man vill ha nya nummer att sortera använder man sig av reset som ser till att nya nummer blir slumpade.

Denna design har valts på grund av att den trots sin enkelhet är tydlig och gör det lätt att följa med i det som händer i sorteringen.

## 3.6 Hash algoritmer




*Figur 3-4: Hash algoritm design*

Detta är den bild man får upp när man valt hashing från menyn i kap 3.3.

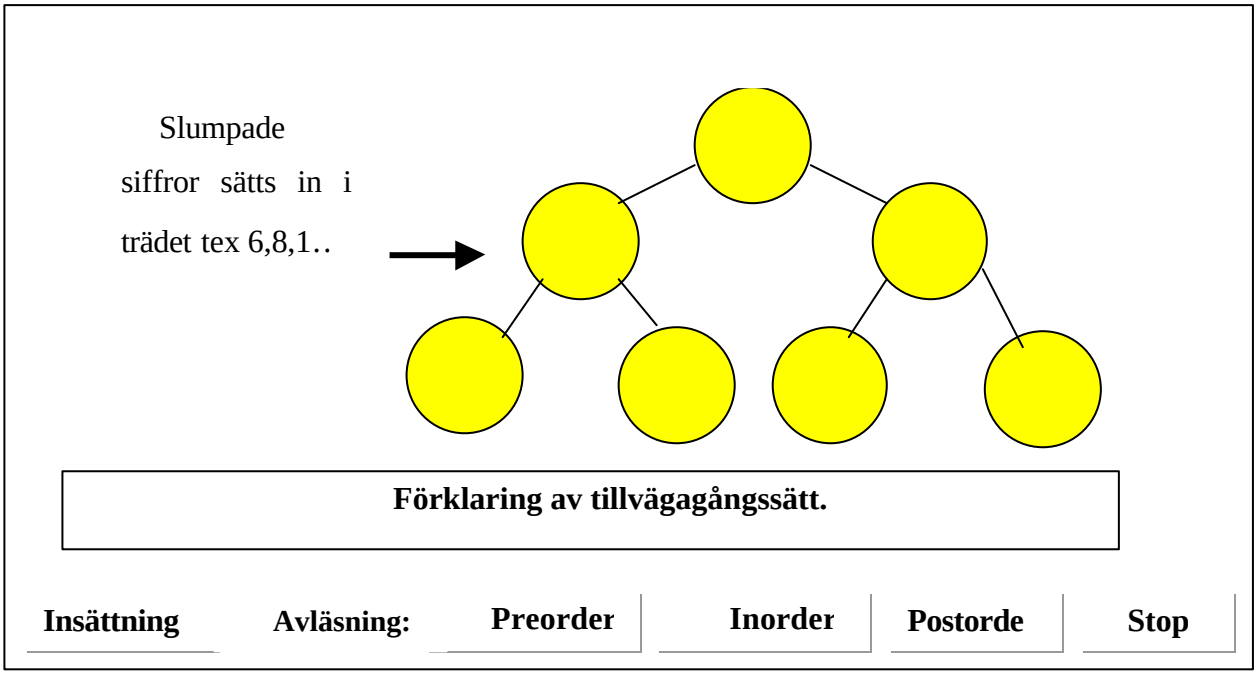
Hashing genomförs på följande sätt, tal slumpas genom random funktionen och skickas sedan vidare till hashfunktionen. När talet är omräknat läggs den in i tabellen. Om det uppstår kollision vid inläggandet skickas talet vidare till en kollisionshanteringsfunktion för omberäkning och läggs sedan in i tabellen. Vidare beskrivningar finns i kapitel 4.

Kollisionshanteringsfunktionen som man vill använda sig av väljs i en rullningslist .

Här ser man de olika talen förflytta sig först till hashfunktionen för omräkning och sedan till tabellen. Om kollision uppstår markeras detta genom att talet ändrar färg för att användaren ska uppmärksamma att kollision har skett. Sedan skickas talet vidare till den kollisionshanteringsfunktion man valt för omberäkning och sedan vidare till tabellen.

De olika kollisionshanteringsfunktionerna som finns att välja i rullningslisten är: Seperate Chaining, Open Addressing, Linear Probing, Quadratic Probing samt Double Hashing (se kap 4.4)

## 3.7 Träd-algoritmer




*Figur 3-5: Träd algoritm design*

Detta är den bild man får upp när man valt träd från menyn i kap 3.3.

När man valt insättning används random funktionen som slumpar sju siffror som sätts in i trädet genom att siffrorna förflyttar sig till den plats i trädet som är korrekt. När avläsning ska göras väljer man vilken sorts avläsning man vill göra. Man kan välja att läsa av trädet inorder, preorder eller postorder. Det finns en ruta för förklaringar så att användaren ska kunna följa med i animeringen hela tiden.

## 3.8 Summering

I detta kapitel har det tagits upp vad som är viktigt när man gör en design. Det har diskuteras text, bilder, skärmlayout, menyer och knappar. Här har tagits upp de olika sätt som de designats och även hur trädalgoritmerna, hashfunktionerna och sorteringsalgoritmernas design är gjord. Det är viktigt att designen är konsekvent och relevant för uppgiften.

Det är viktigt att en animering framhäver det man vill förmedla och inte försvårar förståelsen av det man vill visa. I fallet med animeringar som hjälp att förstå datastrukturer och algoritmer är det viktigt att hålla en enkel design för att framhäva de olika momenten i algoritmerna.

# 4 Implementation

Det som framför allt har implementerats är sorteringsalgoritmerna Bubblesort och Quicksort samt en hashfunktion. Animeringen har skett genom att förflytta olika objekt med hjälp av beräkningar och omritningar. Den implementationsteknik som har använts är språket Java. Java är ett programmeringsspråk samt ett system som stöder program skrivna i detta språk. Java är utvecklat av Sun Microsystem som låter alla använda det fritt. Java lämpar sig väl för programmering för WWW eftersom det är säkert och program skrivna i språket blir små [5].

## 4.1 Introduktion

Java är ett objektorienterat språk som från början var tänkt att användas i olika typer av elektronik. Det man sökte var ett snabbt, säkert, robust och plattformsoberoende språk. Java är ett kompilerande språk på så sätt att ett Java program kompileras till något som kallas bytecodes. Bytecodes utgör ett sorts mellanstadium till källkod och maskinkod som ligger så nära maskinkod som möjligt utan att vara plattformsberoende. När programmet sedan körs, tas det om hand av en interpretator som känner av vilken hårdvara som programmet körs på. Detta gör att Java i princip kan köras på vilken plattform som helst [2].

## 4.2 Teknik och metoder

För att animera algoritmer måste flera olika delar fungera. Dels så måste de algoritmer som ska animeras fungera på ett tillfredsställande sätt samt förstås de funktioner som binder samman det hela med trådar och allt som hör därtill. Därpå ska det finnas knappar för att användaren ska kunna använda sig av programmet.

### 4.2.1 Animeringen

Animering går ut på att skapa rörliga bilder [6]. Detta kan ske på framför allt tre olika sätt:

1. En slinga håller reda på vilken bild som ska visas härnest och i metoden paint() ritas sedan den aktuella vyn upp. Det gäller att visa tillräckligt många bilder per sekund och att tiden

mellan varje bild är densamma för att få en jämn animering. Detta sätt används ofta när man tillverkar tecknad film.

2. Det andra sättet är att helt enkelt flytta ett objekt på skärmen. Om positionen för ett objekt ändras med jämna mellanrum kommer det att se ut som att objektet rör sig. Förändringen sker i en slinga, det vill säga att den nya positionen för objektet räknas ut och själva visningen sker i metoden paint().
3. Det tredje sättet som är det mest traditionella, är att visa flera riktiga bilder i snabb följd för att på så sätt ge sken av att bilderna rör sig. Även här väljs rätt bilder ut i en slinga för att sedan ritas ut i metoden paint().

### 4.2.2 Knappar och händelsehantering

Knapparna skapas genom klassen Button [2]. För att få knapparna på rätt plats så läggs de i en Panel som binder ihop dem så att de hamnar bredvid varandra. För att få Paneln att lägga sig där den ska vara används BorderLayout (se kap3.4.2) som är en komponentdispositions-hanterare för fönsterobjekt och GridLayout (se kap 3.4.1) som delar in behållaren som den hanterar i ett rutnät och placerar ut komponenter i detta ruta för ruta.
När någon av knapparna tryckts så görs ett anrop till actionPerformed () och därifrån anropas den funktion som är kopplad till respektive knapp.

### 4.2.3 Trådar

I Java finns stöd för s.k. trådar, alltså möjlighet att exekvera en separat process - inom ett program eller en applet - separat i sitt eget kontrollflöde. Vissa applikationer, t.ex. animeringar, arbetar annorlunda; de lever sitt eget liv och tar egna initiativ. För att den typen av applikationer skall kunna leva tillsammans med andra är det mycket viktigt att de inte tar all datorkraft i anspråk. Man kan t.ex. inte låta metoden `paint()` sköta en lång animering som består i att visa bild efter bild eftersom datorn under tiden inte kan göra annat. Alla små animeringar som finns på webbsidor skulle då hindra oss från att göra något annat medan de pågår.
. Programmet gör två saker samtidigt, dvs det är uppdelat i två trådar som exekverar bredvid varandra.

En dator kan egentligen inte gör fler saker samtidigt än vad det finns processorer i datorn. Och eftersom de flesta datorer bara har en processor så blir det också bara en process i taget.

För att komma runt det problemet växlar datorns operativsystem (mycket) snabbt mellan flera olika processer vilket gör att det ser ut som om flera olika processer exekverar på en gång.

En stor fördel med trådar är att man kan låta saker som tar lång tid gå i bakgrunden medan användaren av programmet kan fortsätta arbeta med något annat. Å andra sidan gäller det att tänka sig för så att det inte uppstår en konflikt när två olika trådar samtidigt ska använda sig av en enda resurs.

### 4.2.4 Varför använda trådar?

Anledningen till att dela upp ett program i flera trådar istället för att starta flera olika delprogram i olika processer är ett det går väldigt mycket fortare att skapa och byta trådar än att skapa och byta processer, bland annat på grund av att trådar inte har egna program- och dataareor i minnet. Men varför över huvud taget införa parallellitet i ett program? Här kommer några anledningar:

1. Kortare svarstid. För till exempel en webserver är det viktigt att snabbt ge svar till klienterna. Om en ny klient skulle behöva vänta tills alla som anropat tidigare var helt klara skulle det kunna ta väldigt lång tid att få något som helst svar. Det skulle kanske verka som om servern inte var igång. Då är det bättre att skicka svar till flera klienter samtidigt (i olika trådar).
2. Snabbare exekvering. Med flera trådar kan CPU:erna i en dator med mer än en CPU utnyttjas effektivare.
3. Slippa vänta på I/O. Genom att placera I/O i en egen tråd kan processorn arbeta med något annat samtidigt utan att behöva vänta på att I/O-hanteringen avslutas.
4. Simulering. Trådar kan användas för att simulera verkliga parallella förlopp.

### 4.2.5 En tråds livscykel

På samma sätt som ett vanligt program, har en tråd en början och ett slut och däremellan sker trådens exekvering. Detta är vad man kallar en tråds livscykel och dess delar beskrivs nedan.

- En tråd skapas med hjälp av en konstruktor
- En tråd startas (eller oftare hamnar i tillståndet "färdig att köras")
- En tråd kan exekveras varefter den hamnar i något av följande tillstånd:

    - färdig att köras
    - sovande
    - suspenderad
    - väntande
    - blockerad (väntande på att en I/O-operation ska bli klar)
- En tråd kan försvinna (eller snarare fås att försvinna)

Förutom att placera trådar i de olika tillstånden måste man som programmerare ibland också synkronisera olika trådar som arbetar på samma data.

Klassen `Thread` implementerar trådar. Dess vanligaste konstruktorer är:

- Thread()
- Thread (Runnable)
- `Runnable` är ett gränssnitt med en enda metod, `run()` som alltså måste implementeras, endera i en subklass till `Thread` eller i en annan klass som då ges som första argument till den andra konstruktorn. När en tråd startas anropas `run()` automatiskt. Eftersom `Thread` implementerar (en tom) `run()` ska man inte skriva `implements Runnable` om man subklassar `Thread`.

Viktiga metoder i `Thread` är :

- static void sleep(long millis) - söver ner den tråd som exekverar
- static void sleep(long millis , int nanos) - söver ner den tråd som exekverar
- void start() - startar exekveringen av tråden
- void final stop() - stoppar exekveringen av tråden
- void final suspend() - stoppar tillfälligt en tråd
- void final resume() - startar om en tråd som stoppats med suspend()
- static yield() - stoppar tillfälligt den tråd som nu exekverar

Det kan se konstigt ut med statiska metoder men om de anropas i en tråds metod arbetar de ju på den tråden eftersom det är den som exekverar.

Ingen av metoderna ovan ersätts vanligen, man anropar dem vid behov.

### 4.2.6 Dubbelbuffring

Det finns flera tekniker att minska flimret i animeringen och en vanlig teknik är s.k. dubbelbuffring, dvs man ritar sin bild i ett "fönster" i minnet och överför sedan hela bilden till skärmen. I Java är det lätt att ordna detta genom att ersätta metoden `update()`. I större applikationer kan man uppnå mycket genom att rita om bara den del av appleten som ändrats i stället för att rita om allt.

### 4.2.7 Synkronisering

Olika trådar arbetar ibland på samma data. För att detta skall fungera krävs att trådarna synkroniseras så att inte en av trådarna läser data som inte finns.

### 4.2.8 Multithreading

Multithreading är att flera trådar kan exekvera samtidigt i ett program. Med multithreading avses oftast att flera trådar arbetar asynkront: t.ex. att en tråd utför något tungt i bakgrunden medan en annan tråd håller igång användargränssnittet.

### 4.2.9 Strömmar

De flesta program måste kunna läsa och/eller skriva data. Det kan gälla information på en disk, någonstans på nätet, i minnet eller någon annanstans. Java har ett gemensamt gränssnitt för all sådan dataöverföring: **strömmar** (eng: **streams**) [2]. Algoritmer och metoder för att läsa eller skriva data ser exakt likadana ut oavsett varifrån man läser eller var man skriver.

Klasserna kan delas upp litet olika beroende på synsätt. En uppdelning kan vara huruvida man arbetar med tecken eller bytedata (bilder, ljud, etc).

## 4.3 Sorterings algoritmer

De algoritmer som har implementerats är Bubble sort och Quicksort. Animeringen för Bubblesort har implementerats genom att beräkna de nya positionerna dit sorteringen ska förflyttas se bilaga B.

Det bestämdes att de tal som slumpades av slumptalsgeneratorn skulle läggas i en array, vara sju till antal och ha värden mellan 1-20. Samma siffra kan förekomma upprepade gånger i samma sekvens. Dessa siffror skrivs sedan ut på skärmen med hjälp av en applet. För att detta ska fungera krävs dock att applet.drawstrings första argument är en sträng och det som finns i den skapade arrayen är av int typ, dvs heltal, så man måste konvertera heltalen i arrayen vid utskrift. För att rita cirklarna gjordes en egen funktion som anropades från själva sorteringsfuntionen. De lades sedan på position så att de slumpade talen hamnade i mitten. För att animeringen skulle fungera beräknades både ringarnas och de slumpade talens nya positioner och ritades om med hjälp av dubbelbuffring, se kap 4.2.6. För att inte dess gamla position skulle synas rensades skärmen mellan varje uppritning. Genom att trycka på start sattes sorteringen igång och animeringen började. För att tydligt visa vilka siffror som skulle sorteras ändrade dessa färg innan de började sin förflyttning. Sorteringen kunde hela tiden ändra status med hjälp av de olika knapparna. För att detta skulle fungera användes trådar, se kap 4.2.3.

Animeringen för Quicksort har implementerats på sådant sätt att en array med konstanta värden använts. Detta gjordes på grund av att beräkningarna var för tidsödande när de element som skulle sorteras inte stod intill varandra så som de gör i Bubblesort. Sedan gjordes beräkningar för dessa konstanta värden och animeringen skedde på samma sätt som i Bubblesort med hjälp av trådar.

## 4.4 Hash algoritmer

### 4.4.1 Idén med hashning

Binärsökning i en ordnad sekvens går visserligen snabbt, men sökning i en hashtabell är oöverträffat snabbt. Och ändå är tabellen helt oordnad (hash betyder ju hackmat, röra). Låt oss säga att vi söker efter Pelle i en hashtabell av längd 10000. Då räknar vi först fram *hashfunktionen* för namnet Pelle och det ger detta resultat:

```
"Pelle".hashCode()->72260712
```

Hashvärdets rest vid division med 10000 beräknas nu

```
72260712 % 10000 -> 712
```

och när vi kollar hashtabellens index 712 hittar vi Pelle just där!

Hur kan detta vara möjligt? Ja, det är inte så konstigt egentligen. När Pelle skulle läggas in i hashtabellen gjordes samma beräkning och det är därför han lagts in just på 712. Hur hashfunktionen räknar fram sitt stora tal spelar just ingen roll. Huvudsaken är att det går fort, så att inte den tid man vinner på inbesparade jämförelser äts upp av beräkningstiden för hashfunktionen.

### 4.4.2 Komplexiteten för sökning

Linjär sökning i en oordnad sekvens av längd N tar i genomsnitt N/2 jämförelser, (O(n)), binär sökning i en ordnad sekvens log N, (O(log n)), men hashning går direkt på målet och kräver bara drygt en jämförelse (O(1)). Varför drygt? Det beror på att man aldrig helt kan undvika *krockar*, där två olika namn hamnar på samma index.

### 4.4.3 Dimensionering av hashtabellen

Ju större hashtabell man har, desto mindre blir risken för krockar. En tumregel är att man bör ha femtio procents luft i tabellen. Då kommer krockarna att bli få. En annan regel är att tabellstorleken bör vara ett *primtal*. Då minskar också krockrisken.

### 4.4.4 Hashfunktionen

Oftast gäller det först att räkna om en `String` till ett stort tal. I Java gör man ingen skillnad på en bokstav och dess nummer i UNICODE-alfabetet, därför kan `ABC` uppfattas som `656667`. Det man då gör är att multiplicera den första bokstaven med 10000, den andra med 100, den tredje med 1 och slutligen addera talen. På liknande sätt gör metoden hashCode():

public int hashCode()

Returnerar en hashkod för en sträng. Hashkoden för ett String object beräknas som följer:

s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

genom att använda int arithmetic, där s[i] är de i:e tecknet i strängen, n är strängens längd och ^ indikerar exponenten. Hashvärdet på en tom sträng är 0.

Om man vill söka på datum eller personnummer kan man använda det som stort heltal utan särskild hashfunktion. Det visar sej att primtalsstorlek ger bäst spridning [6].

Alla objekt i Java får automatiskt en hashCode()-metod (ärvd från klassen Object). Men i regel returnerar metoden bara objektets minnesadress omvandlat till ett heltal, vilket vi inte har någon större nytta av.

### 4.4.5 Krockhantering

Det naturliga är att lägga alla namn som hashar till ett visst index som en länkad lista (krocklista). Om man har femtio procents luft i sin tabell blir krocklistorna i regel mycket korta.

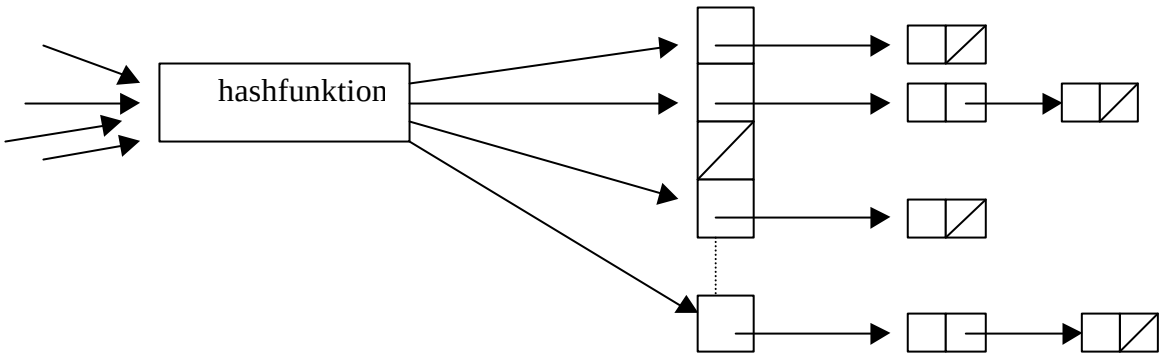
Den andra idén är att vid krock lägga posten på första lediga plats. En nackdel blir att man sedan inte enkelt kan ta bort poster utan att förstöra hela systemet. En fördel är att man slipper alla pekare.

Det finns flera krockhanteringsfunktioner som kan användas[3]:

- Separate Chaining
- Open addressing
  - Linear Probing
  - Quadratic Probing
- Doublehashing

### 4.4.6 Separate Chaining

Separate chaining går ut på att lägga alla element som har samma hashvärde i en länkad lista. Varje element som sparas i en given länkad lista har samma nyckel. I detta fall kommer hashtabellen inte längre att innehålla element utan pekare till länkade listor [3].




*Figur 4-1: Hashing - Separate Chaining*

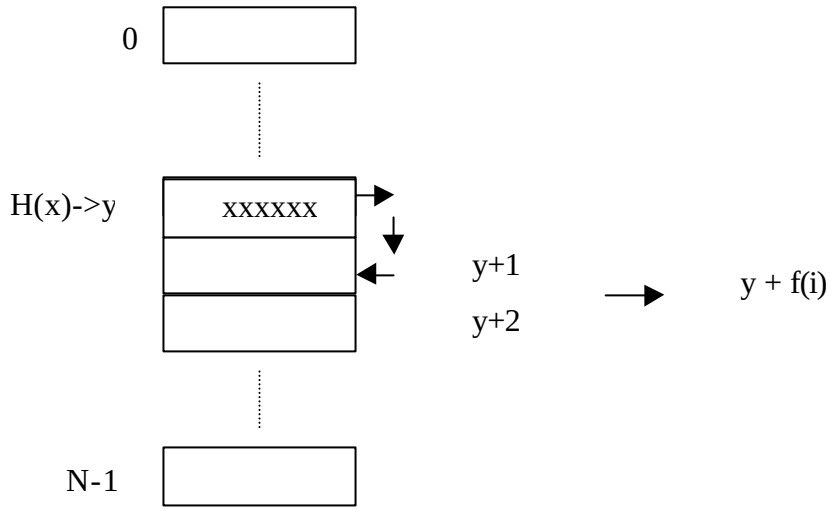
### 4.4.7 Open Addressing

I open addressing är alla data sparade i själva hashtabellen. Kollisioner löses genom att beräkna en sekvens av hash slots. Sekvensen undersöks successivt till man hittar en tom plats. Fördelen är att man kan undvika att använda pekare. Minnet som man sparar genom att inte använda pekare kan användas till att skapa en större hashtabell om nödvändigt. Två exempel av open addressing är linear probing och quadratic probing [3].

### 4.4.8 Linear Probing

Linear Probing går ut på att hashtabellen gås igenom steg för steg tills en ledig plats hittas. Där läggs sedan hashvärdet.

F(i) = i vilket ger att f(1) = 1, f(2) = 2 osv




*Figur 4-2: Linear Probing*

### 4.4.9 Quadratic Probing

Quadric Probing fungerar på ett liknande sätt som Linear Probing, skillnaden är att hashtabellen gås igenom i steg baserade på i^2 istället för i. Detta innebär:

f(i) = i^2 vilket ger att f(1) = 2, f(2) = 4 osv. och då y+1, y+4, osv

### 4.4.10 Double Hashing

Double Hashing fungerar som så att man har två hashfunktioner $h'_1$ och $h'_2$. Sedan beräknas en sekvens på följande sätt:

$H(k, i) = (h'_1(k) + ih'_2(k)) \bmod N$

Där k = konstant och i = antal gånger. För den första beräkningen är i 0.

### 4.4.11 Javaklassen Hashtable

Hashtable implementerar gränssnittet Map [2], vilket betyder att element förknippas med nycklar istället för index. Varje nyckel förvandlas med hjälp av en hashfunktion till ett unikt index. Elementet läggs sedan in på detta index i hashtabellen. När ett element ska plockas ut räknas indexet fram igen av hashfunktionen och elementet kan tas fram. Denna uträkning av index vid insättning och uttag av element sker automatiskt av klassen.

Hashtable är en utmärkt och lättskött klass med två anrop, put och get. Första parametern till put är söknyckeln, till exempel personens namn. Andra parametern är ett objekt med alla tänkbara data om personen. Metoden get har söknyckeln som indata och returnerar dataobjektet om nyckeln finns i hashvektorn, annars returneras null.

## 4.5 Summering

I detta kapitel har de olika tekniker och metoder som använts under implementationen tagits upp. Däribland trådar, dubbelbuffring, knappar, händelsehantering och animering.
Implementation av sorterings algoritmerna och hashtabellen har tagits upp samt olika krockhanteringsmetoder. Den mycket användbara klassen Hashtabel beskrivs också här.

# 5 Resultat

## 5.1 Introduktion

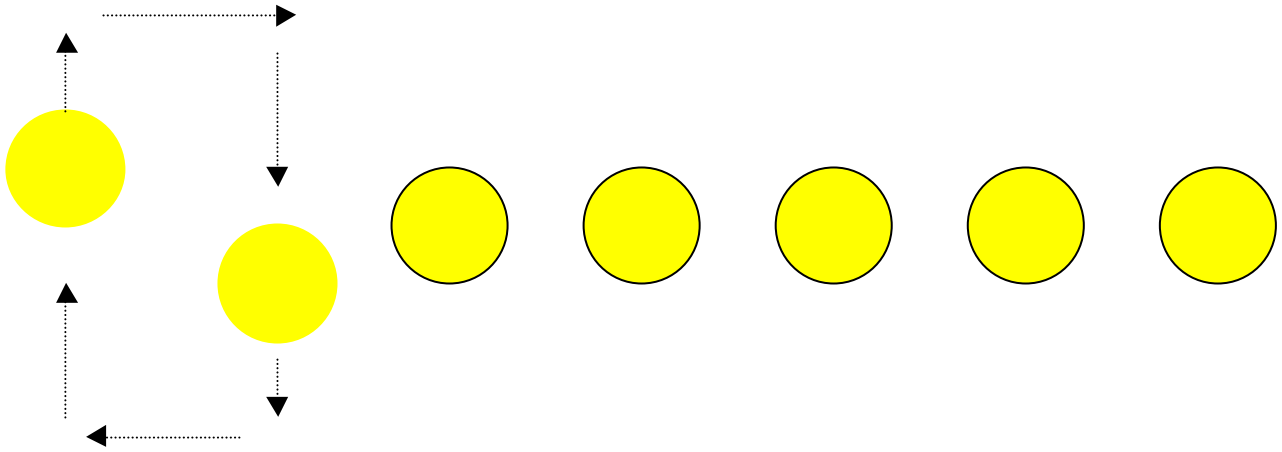
Detta kapitel tar upp de resultat som projektet gett samt en kritisk värdering

## 5.2 Algoritmer

De sorteringsalgoritmer som implementerats är Bubblesort och Quicksort. En hashfunktion har också implementerats.

### 5.2.1 Bubblesort

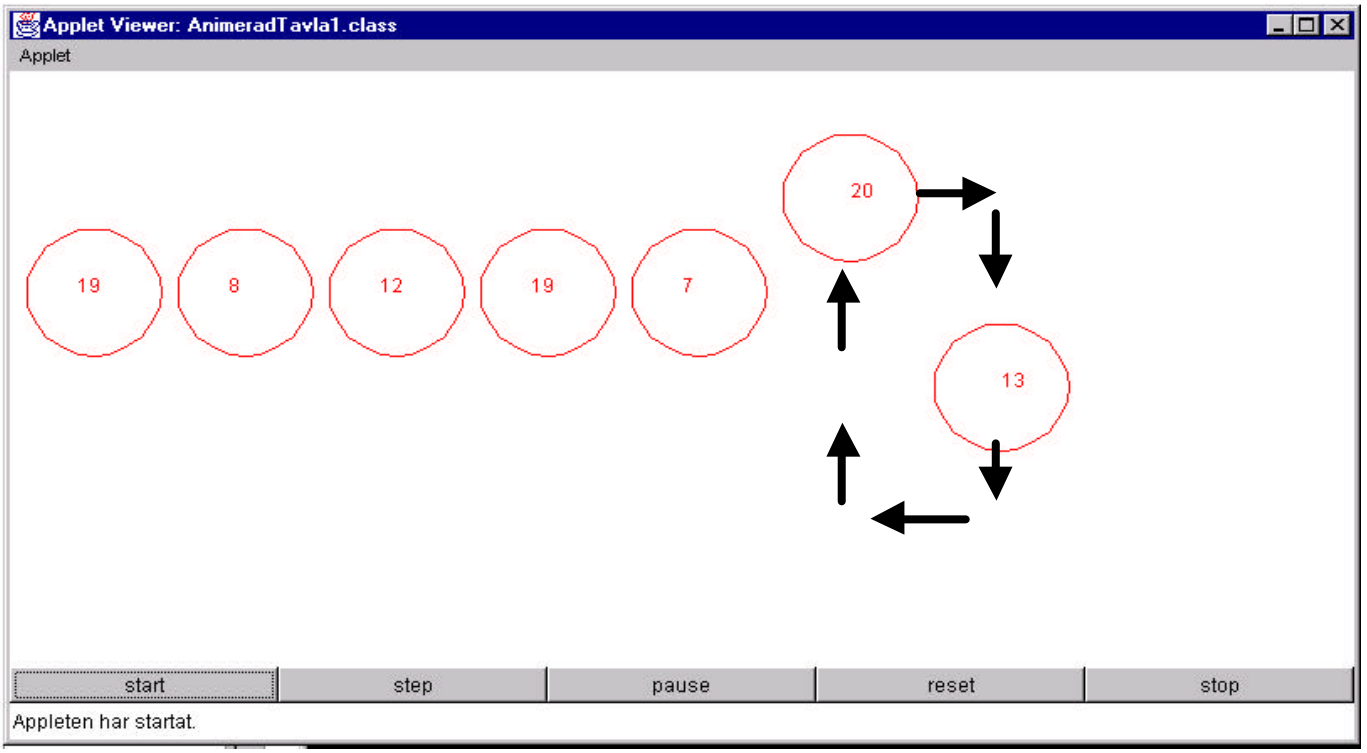
Bubblesort fungerar på så sätt att man väljer den sorteringsalgoritm man vill se genom att trycka på startknappen i programmet. Då startar algoritmen och animeringen som har gjorts genom beräkningar och bilden ritas upp för varje förflyttning. Detta gjordes genom dubbelbuffring .



*Figur 5-1: Resultat av bubblesortsortering*

De olika bubblorna förflyttar sig, den ena uppåt, den andra neråt och sedan åt sidan för att byta plats. För att man ska hinna se vilka som sorteras markeras detta genom att färg bytes två gånger på dem, så att det ger sken av att de blinkar.

Genom att använda flera olika trådar löstes problemet med att köra sorteringen men samtidigt kunna stänga av eller trycka på någon annan knapp.

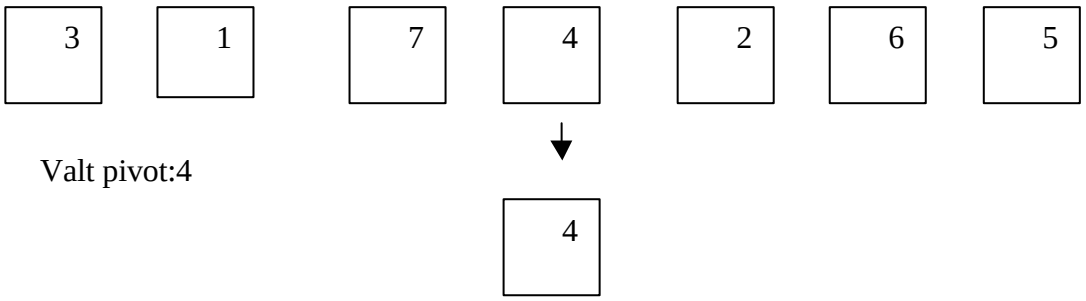



*Figur 5-2: Resultat av implementering av Bubblesort sortering*

### 5.2.2 Quicksort

Quicksort algoritmen bygger ju på rekursion så den gjordes på följande sätt. Ett antal siffror som är konstanta sattes i en array. Sedan sorterades siffrorna med hjälp av quicksortalgoritmen. Då animeringen för bubblesort byggde på att de siffror som sorterades stod bredvid varandra var denna tvungen att göras om. Detta löste jag genom att en konstant array med konstanta värden användes för att förenkla uträkningarna. Sedan beräknades varje förflyttning för just dessa värden. Detta är naturligtvis inte den bästa lösningen. Det är mycket bättre med en allmän lösning som i Bubblesort algoritmen så att det inte spelar någon roll

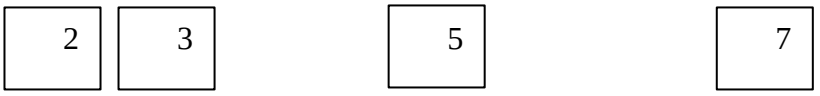
vilka tal som ska sorteras. Tyvärr var detta för tidskrävande och komplicerat för att hinnas med i detta projekt. Förflyttning sker enligt pilarna - se Figur 5-3.
Dessa värden finns från början i arrayen, skrivs i appleten i denna ordning;



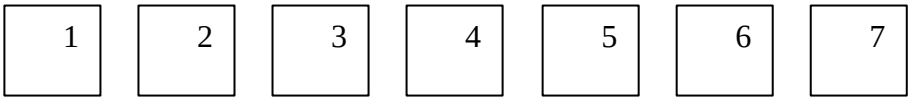

De tal som är < 4 sätts på den vänstra sidan och de tal som > 4 på den högra


Nya pivot väljs ut: 1 och 6


De tal som är < 1 sätts på den vänstra sidan och de tal som > 1 på den högra
De tal som är < 6 sätts på den vänstra sidan och de tal som > 6 på den högra




Nu är sekvensen sorterad och det är dags att sätta ihop detta görs på samma sätt fast åt andra hållet (rekursion) tills man erhåller:




*Figur 5-3: Utförande av quicksort*

### 5.2.3 Hashing

Då jag inte har lyckats med en allmän lösning med hashing har jag löst detta genom att ha förbestämda värden som man ser förflyttar sig till hashfunktionen för omräkning och sedan ser man det nya omräknade hashvärdet skickas till tabellen. När talet är omräknat läggs den in i tabellen. Kollisionshanteringsfunktionen som man vill använda sig av väljs i en rullningslist se design kap 3.6 . Om kollision uppstår markeras detta genom att talet ändrar färg för att användaren ska uppmärksamma att kollision har skett. Sedan skickas talet vidare till den kollisionshanteringsfunktion man valt för omberäkning och sedan vidare till tabellen.
De olika kollisionshanteringsfunktionerna som finns att välja i rullningslisten är: Seperate Chaining, Open Addressing, Linear Probing, Quadratic Probing samt Double Hashing.

## 5.3 Värdering

Området med datastrukturer och algoritmer är väldigt stort och innefattar väldigt många olika algoritmer vilket gör att man måste begränsa sig när det gäller en implementation. För att kunna utföra detta projekt på bästa sätt måste man vara insatt i varje algoritm till fullo något som kan vara svårt då det i många fall är komplicerade algoritmer det handlar om. När man från början satte sig in i de algoritmer som skulle ingå beslutades att man skulle börja med sorteringsalgoritmerna vilket visade sig fullt tillräckligt. Projektet krävde stor inläsning då även de algoritmer som inte innefattades i begränsningen skulle studeras för att kunna implementeras i mån av tid. Kanske hade det varit bättre att begränsa sig redan från början och bara koncentrera sig på en del, sekvens, träd eller grafer redan vid inläsningen. Projektet försvårades av att det inte fanns någon med kunnande inom animering som man kunde diskutera med utan allt måste tas från böcker och webben. Att läsa om något och att göra det är väldigt skilda saker och man hann göra fel ett antal gånger innan man hittat rätt sätt. Detta hade kunnat undvikas om man genom diskussion med någon kommit fram till vilken animeringsmetod som skulle användas i detta projekt från början. Att jobba med detta projekt har varit mycket intressant och lärorikt och är en förberedelse för de projekt man kommer att arbeta med i det verkliga livet..

## 5.4 Summering

I detta kapitel har det tagits upp hur de algoritmer som implementerades fungerar. Det har även gjorts en värdering av resultatet.

# 6 Slutsats

## 6.1 Introduktion

Det som från början verkade vara en överkomlig uppgift visade sig snart innebära mer än man trott. Många timmar gick åt till att studera uppgiften genom böcker och webben. Det som tog mycket tid var att beräkna hur animeringen skulle förflytta sig samt alla de försök som gjordes med implementationen men inte lyckades och man fick återgå till analysfasen.

## 6.2 Slutsats av projektet

Från början planerades att hantera de olika sorteringsobjekten som bilder genom att använda bild klassen image() men det konstaterades efter ett antal försök att detta inte skulle fungera. Istället användes sättet där objektets nya position beräknas och ritas om.

Vid implementationen av Bubblesort gjordes beräkningar som fungerar för alla sorteringar som jämför de tal som står bredvid varandra, detta oavsett vilka tal som slumpats. För att få denna algoritm att uppföra sig på ett önskvärt vis lades mycket tid på att analysera de olika metoderna för animering (se kap 4.2.1). För att kunna beräkna de olika förflyttningarna studerades koordinatsystemet i detalj och många tester utfördes innan det fungerade på ett tillfredsställande sätt.

Då tiden rann iväg så hann jag tyvärr inte skapa en tillfredsställande lösning för Quicksort utan fick nöja mig med att visa med konstanta värden hur denna fungerar. Detta gäller också för hashfunktionen.

Jag lade ner enormt mycket tid på att ta reda på fakta om animeringen och redan där spräcktes min tidsplan. När det sedan var dags att implementera hade mycket tid gått åt till forskning och tiden som fanns kvar var begränsad. Genom den här rapporten så hoppas jag att de som tar vid inte behöver börja från början utan ta vid där jag slutar.

## 6.3 Framtida arbete

För framtiden gäller det att få en generell lösning för de rekursiva algoritmerna samt hashfunktionen.

Det finns mycket att bygga vidare på och visa med t ex grafer och träd som kan vara till nytta för alla som ska studera dessa algoritmer och datastrukturer.

## 6.4 Slutkommentarer

Projektet har varit en nyttig lärdom av hur det är att arbeta med ett projekt under tidspress och med ett eget ansvar. Något som gjorde det hela lite mer komplicerat var att arbeta ensam utan att ha någon att diskutera med. Detta gjorde att timmar gick när man försökte lösa problemen själv. Har man någon att diskutera med så går det ofta mycket fortare att komma på lösningar på problem.

Det som var svårast var att sätta sig in i projektets alla delar. En nyttig lärdom att göra en sak i taget och se till att det fungerar innan man går vidare.

Man har lärt sig att ta egna initiativ och att lära sig genom sina misstag något som tyvärr är mycket tidsödande. För det är ju det det är, en fråga om tid.

## Referenser


1. Bohman, J. *Grafisk design*, [Spektra, Halmstad, 1996]
2. Ek, J.Ekman, R. *Java-applets* [Graphics System, Göteborg, 1997]
3. Heileman, G. Data Structures and Algorithms and Objectoriented Programming [International Edition, 1996]
4. Hellmark, C. *Typografisk handbok* [Ordfront, Stockholm, 1998]
5. Omander, M. *Java 2.0* [Elanders Graphics System, Angered, 1999]
6. Wiess, M.A. *Data Structures & Algorithm Analysis in C++* [Addison Wesley 1999]
7. http://www.epaperpress.com/sortsearch/index.html [Niemann 2002]
8. http://www.dd.chalmers.se/~f96hajo/sortdemo/sd_eng.htm [Johansson 2002]


## Bibliografi


1. Weston, Anthony, *A Rulebook for Arguments* [Hacket Publishing Company, 1992]
2. Widerberg, K. *Att skriva vetenskapliga uppsatser* [Studentlitteratur, 1995]
3. Norman, Donald A. *The Design of Everyday Things* [The MIT Press, 1989]

# Bilaga A - Sorterings algoritmer

**BubbleSort:**

Denna sorteringsalgoritm går ut på att föra stora värden mot slutet av arrayen och små värden mot början av arrayen. En utförlig förklaring visas nedan.

1. Utför en iteration från första index till näst sista index. För varje indexvärde i iterationen skall aktuell position jämföras med nästa position. Om nästa position (i förhållande till aktuell position) har ett lägre värde än aktuell position skall positionernas värden byta plats. När iterationen är klar vet man att största värdet finns i slutet av arrayen.
2. Om någon ändring har skett under föregående iteration skall förfarandet utföras en gång till. Om inte någon ändring har skett betyder det att arrayen är sorterad. För varje gång man utför iterationen på nytt kan man minska slutindex med ett, därför att det största värdet "flyter med" hela vägen till slutet.

```
Public void Bubblesort (int[] arr)
{ int t, size= arr.length-1;
for (int i = 0; i <= size; i++)
  for (int j = size; j < i; j--) if (arr[j]< arr[j-1]
       { t = arr[j-1]; arr[j-1] = arr[j]; arr[j] =  t; }
}
```

**InsertSort:**

Betraktar en del av listan till vänster som sorterad. Vid varje iteration sätts nästa element in på rätt plats i den sorterade delen genom bubbling neråt.
Tänk dig att du har en hög med spelkort som ska sorteras. De ligger med baksidan upp på bordet. Tag upp det översta kortet. Du har nu en sekvens av ett kort i handen och sekvensen är sorterad. Ta nu upp ytterligare ett kort ur högen på bordet. Sätt in det på rätt plats i handen. Du har nu en sekvens av två sorterade kort i handen. Fortsätt på detta sätt tills högen på bordet är slut. Du har nu alla korten i handen i en sorterad sekvens.

```
public void Insertion(int[] arr)
                {
    int tmp, size = arr.length-1; int j; for (int i=1; i<size; i++){
    tmp = arr[i]; for(j=i; j>0 && tmp < arr[j]; j--) arr[j]= arr[j-1]; arr[j]= tmp;
    }repaint();
                }
```

**SelectionSort:**

Betraktar en del av listan till vänster som sorterad. Vid varje iteration letas det minsta av de återstående elementen upp och placeras in på nästa plats.
Selection sort bygger på följande idé:

> Hitta det minsta elementet i listan och lägg det på plats ett. Vi har nu en sorterad lista av längd ett på den första platsen i listan. Sök nu upp det minsta elementet bland de osorterade elementen på plats två till N och lägg det på plats två. Vi har då en sorterad lista av längd två på de två första platserna. Fortsätt att på detta sätt hitta det minsta elementet bland de osorterade elementen och lägga det på den första osorterade platsen, tills hela listan är sorterad.

```
   public static void selectionSort (int[] numbers)
{int min, temp;
for (int index = 0; index < numbers.length-1; index = index + 1)
  { min = index;
    for (int scan = index+1; scan < numbers.length; scan = scan + 1)
    if (numbers[scan] < numbers[min]) min = scan;
    temp = numbers[min];
    numbers[min] = numbers[index];
    numbers[index] = temp;
  }
}
```

**ShellSort :**

Fungerar huvudsakligen som den ökända BubbleSort, men börjar med att jämföra element på stora avstånd (t.ex. halva listans längd) och går ned i avstånd mellan elementen som jämförs tills avståndet är 0 och därmed hela listan är sorterad. På detta sätt genomförs först en grovsortering av listan.

```
public static void shellsort( Comparable [ ] a )
   {for( int gap = a.length / 2; gap > 0;
gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) ) for( int i = gap; i < a.length; i++ )
    {Comparable tmp = a[ i ]; int j = i;
            for( ; j >= gap && tmp.compareTo( a[ j - gap ] ) < 0; j -= gap )
            a[ j ] = a[ j - gap ]; a[ j ] = tmp;}
   }
```

**MergeSort :**

Varje steg i algoritmen sorterar ihop två delar av listan. Delarnas storlek börjar med att vara ett och dubblas tills längden är hela listan och den därmed är sorterad. Problemet med algoritmen är att den under sorteringen behöver extra minne proportionellt mot antalet element i listan.

Den är intressant av flera anledningar. Dels är den en av de snabbare, dels kan den grundläggande algoritmen även utnyttjas vid sortering av sekventiella filer.

Ett sätt är att från början betrakta varje enskild post som en sorterad dellista med längden 1. Dessa dellistor samsorteras parvis till dellistor av längden 2, dessa i sin tur till dellistor av längden 4 osv tills dess en enda sorterad lista erhållits.

```
private static void merge( Comparable [ ] a, Comparable [ ] tmpArray,
                           int leftPos, int rightPos, int rightEnd )
    {
   int leftEnd = rightPos - 1;
   int tmpPos = leftPos;
   int numElements = rightEnd - leftPos + 1;

if( a[ leftPos ].compareTo( a[ rightPos ] ) < 0 )
   tmpArray[ tmpPos++ ] = a[ leftPos++ ];
else
   tmpArray[ tmpPos++ ] = a[ rightPos++ ];
   while( leftPos <= leftEnd ) tmpArray[ tmpPos++ ] = a[ leftPos++ ];
   while( rightPos <= rightEnd )
   tmpArray[ tmpPos++ ] = a[ rightPos++ ];
   a[ rightEnd ] = tmpArray[ rightEnd ];
    }
```

**QuickSort:**

Väljer ut ett strategiskt pivotelement och placerar alla element mindre än pivotelementet i början av listan och alla element större än pivotelementet i slutet på listan. Sedan anropas QuickSort rekursivt för början av listan och för slutet av listan.

```
class QuickSort
{   static int[] tab;          .
    public static void Sort(int[] v)  { tab = v; Quick(0,tab.length-1); }
    static int Partition(int a, int b)
     {  int x;   x = tab[a];
       while (a < b)
       { while (tab[b] >= x && a < b)  b--;
if (a == b) break; tab[a++] = tab[b]; while (tab[a] <= x && a < b) a++; if (a == b)  break;
tab[b--] = tab[a]; }
tab[b] = x; return b;}
static void Quick(int a, int b)
{  int split = Partition(a, b);
if (split > a+1) Quick(a, split-1); if (split < b-1) Quick(split+1, b); }
}
```

# Bilaga B - Java koden

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.lang.Math.*;
import java.util.Random;
import java.awt.Point;
import javax.swing.*;
import java.awt.Graphics;

//Ärver klassen Applet och implementerar gränssnittet Runnable som talar om
//att denna klass kan exekvera som en tråd.

public class Sortering extends Applet
implements Runnable
{
    Thread ritare = null;
    Rectangle ytstorlek;
    Button myButton;
    Label myLabel;
    ButtonListener b;
    Button starta = new Button("start");
    Button stop = new Button("stop");
    Button pause = new Button("pause");
    Panel p1 = new Panel();
    int i;
    int [] slumpning;
    Graphics myGraphics;
    Image myImage;
    boolean lock = false;
```

```
**************************************************************************
//Anropas när klassen startar. Här finns “layouten” för knapparna

public void init()
{
           ytstorlek=this.bounds();
           setLayout(new FlowLayout(FlowLayout.RIGHT));
           myButton = new Button("Bubblesort");
           add(myButton);

           b=new ButtonListener();
           myButton.addActionListener(b);
           myLabel=new Label("Ingen");
           add(myLabel);

           setLayout(new BorderLayout());
                      add(p1,"South");
           p1.setLayout(new GridLayout(1,2));

                      p1.add(starta);
                      p1.add(stop);
                      p1.add(pause);
                      starta.addActionListener(b);
                      stop.addActionListener(b);
                      pause.addActionListener(b);


 myImage = createImage(getWidth(), getHeight());
 myGraphics = myImage.getGraphics();

Slumpa();

}
**************************************************************************

//Anropas när man startar appleten

public void start()
{
           ritare = new Thread(this);
           ritare.start();
           }
```

```
************************************************************************
//Anropas när man lämnar appleten
public void stop()
{
   if (null != ritare) { ritare.stop(); ritare = null;}
   }

************************************************************************

public void run()
{
   while (true ) {    //Håller på så länge tråden finns
      this.Paint(this.getGraphics());
      try {   }       //Fångar upp eventuella fel i programmet
      catch (Exception fel) { }
      }
   }

 ************************************************************************
// Här slumpas de nummer som ska sorteras och läggs i en array

public void Slumpa()
{
    int antal     = 7;
    int max       = 20;
    slumpning     = new int[antal];
    Random Siffra = new Random();

    for (int i = 0; i < antal; i++) {
      int nySiffra = Math.abs(Siffra.nextInt()) % max +1;
      slumpning[i] = nySiffra;
      }
   }

************************************************************************
```

```
// Här är bubblesort sorteringen samt beräkningar föe förflyttning.

public void Bubble(int[] arr)
{
   int tmp, size = arr.length-1;

   for (int i = 0; i <= size; i++) {
      for (int j = size; j > i; j--) {
         if (arr[j] < arr[j-1]) {
            int X=10;
            int Y=100;
            X = X + ((j-1) * 90);
           for(int i3 = 0; i3<4;i3++) {
              myGraphics.clearRect(0,0,getWidth(),getHeight());
              myGraphics.setColor(Color.red);

              //Anropar den funktion som ritar upp  cirklarna
              PaintCircles(myGraphics);
              myGraphics.clearRect(((j-1) * 90) + 10,100,81,81);
              myGraphics.clearRect(((j) * 90) + 10,100,81,81);

              //De circlar som ska byta plats byter färg
              if((i3 % 2) == 0){myGraphics.setColor(Color.yellow); }
              else { myGraphics.setColor(Color.blue); }

             //Sätter positionsvärden
              myGraphics.drawOval(X,Y, 80, 80);
              myGraphics.drawString
                (Integer.toString(slumpning[j-1]), X+40, Y+40);

              myGraphics.drawOval(X + 90, 100 + (100 - Y), 80, 80);
              myGraphics.drawString
                (Integer.toString(slumpning[j]), X+90+40, 100+(100-Y)+40);

               // fördröjning
              lock = false; while(!lock); for(int i2=0; i2<8000000; i2++);

              }
           //Sätter tillbaka färgen till röd
            myGraphics.setColor(Color.red);

            //Flyttar de ringar som ska sorteras upp eller ner från dess
           //gamla position
            for(; Y > 10; Y=Y-2 ) {
               myGraphics.clearRect(0,0,getWidth(),getHeight());
               PaintCircles(myGraphics);
               myGraphics.clearRect(((j-1) * 90) + 10,100,81,81);
               myGraphics.clearRect(((j) * 90) + 10,100,81,81);
               myGraphics.setColor(Color.red);
               myGraphics.drawOval(X,Y, 80, 80);
               myGraphics.drawString
                 (Integer.toString(slumpning[j-1]), X+40, Y+40);
               myGraphics.drawOval(X + 90, 100 + (100 - Y), 80, 80);
               myGraphics.drawString
                 (Integer.toString(slumpning[j]), X+90+40, 100+(100-Y)+40);

               lock = false; while(!lock);
               };
```

```
            //Flyttar de ringar som ska sorteras åt höger eller vänster till
            //över eller under dess nya position
            for(int x2=0; x2<90; x2 = x2+2){
               myGraphics.clearRect(0,0,getWidth(),getHeight());
               PaintCircles(myGraphics);
               myGraphics.clearRect(((j-1) * 90) + 10,100,81,81);
               myGraphics.clearRect(((j) * 90) + 10,100,81,81);
               X= X +2;
               myGraphics.setColor(Color.red);
               myGraphics.drawOval(X,Y, 80, 80);
               myGraphics.drawString
                 (Integer.toString(slumpning[j-1]), X+40, Y+40);
               myGraphics.drawOval(X-(2*x2) + 90, 100 + (100 - Y), 80, 80);
               myGraphics.drawString
                 (Integer.toString(slumpning[j]), X+90+40-(2*x2),
                  100+(100-Y)+40);

               lock = false; while(!lock);
               }
            //Flyttar ringarna upp eller ner till dess nya position
            for(; Y < 100; Y = Y+2 ) {
               myGraphics.clearRect(0,0,getWidth(),getHeight());
               PaintCircles(myGraphics);
               myGraphics.clearRect(((j-1) * 90) + 10,100,81,81);
               myGraphics.clearRect(((j) * 90) + 10,100,81,81);

               myGraphics.setColor(Color.red);
               myGraphics.drawOval(X,Y, 80, 80);
               myGraphics.drawString
                 (Integer.toString(slumpning[j-1]), X+40, Y+40);
               myGraphics.drawOval(X - 90, 100 + (100 - Y), 80, 80);
               myGraphics.drawString
                  (Integer.toString(slumpning[j]), X-90+40, 100+(100-Y)+40);
               lock = false; while(!lock);
               };

            tmp = arr[j-1];
            arr[j-1] = arr[j];
            arr[j] = tmp;
            }
         }
      }
      repaint();
   }

*****************************************************************************
// Sorterar arrayen med insertionsort
public void Insertion(int[] arr)
            {int tmp, size = arr.length-1; int j;
            for (int i=1; i<size; i++){
             tmp = arr[i];
            for(j=i; j>0 && tmp < arr[j]; j--)arr[j]= arr[j-1]; arr[j]= tmp;
             }
            repaint();
            }
*****************************************************************************
//Stannar programmet när stop tryckts

public void stanna()
            {}
```

```
*************************************************************************
//Rit funktion



void Paint(Graphics g)
           {
           g.drawImage(myImage,0,0,this); lock=true;
           }

*************************************************************************
//Ritar upp cirklarna
void PaintCircles(Graphics g)
           {int x=40;
           for(i=0; i<7; i++){
           g.drawString(Integer.toString(slumpning[i]), x, 140); x=x+90;
              }
           int X=10;
           for(i=0; i<7; i++){
           g.setColor(Color.red); g.drawOval(X,100, 80, 80);X=X+90;
             }
           }

*************************************************************************
//Pausar när pause knappen tryckts på
public void pause (int ms)
  {
    try { Thread.currentThread().sleep(ms); }
    catch (InterruptedException ex) {   }
    }

*************************************************************************
// Fördröjningsfunktion
public void SleepWait ()
  {
    if (wait_speed != 20){
           try { Thread.sleep ((10-wait_speed)*(10-wait_speed)*10);}
           catch (InterruptedException ex){  }
           }
    }
*************************************************************************
//

private class ButtonListener implements ActionListener
           {
                      public void actionPerformed(ActionEvent e)
                      {
                         if(e.getSource() == starta){Bubble(slumpning);}
                         if(e.getSource() == stop)  {stop();}
                         if(e.getSource() == pause) {pause(10);}
                      }
           }

}
```