Modellering och Datastrukturer

Dr. Johan Hagelbäck

johan.hagelback@lnu.se

http://aiguy.org

Modellering

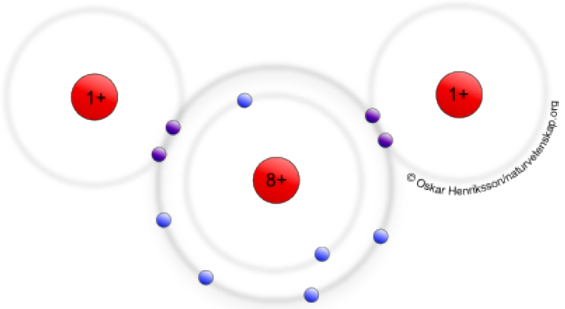
Modellering

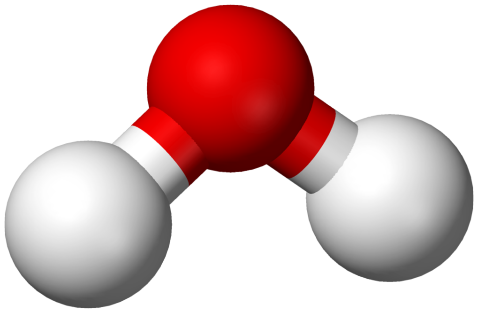
- En modell är en representation av ett objekt, system, algoritm, …
- Modeller är väldigt användbara för att visa saker som är för små, för snabba, för stora, för avlägsna, ...
  - 3D modeller av molekyler
  - Planeterna i solsystemet
  - Vår planets geologiska utveckling
  - Väder och klimatförändringar
  - …
- Modellerna hjälper oss att förstå saker vi inte direkt kan observera

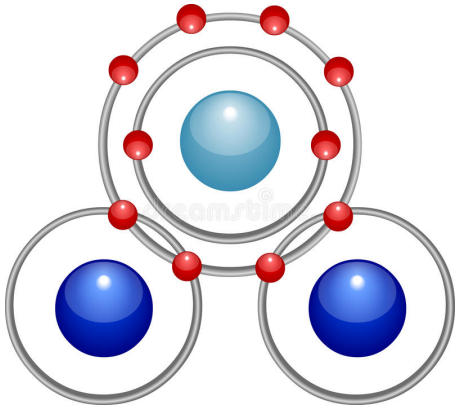
Modellering

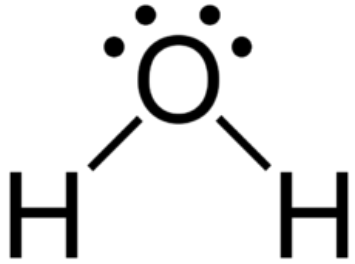
- En modell är en abstraktion av verkligheten
- Modeller kan ha olika detaljnivå, saker kan utelämnas, andra saker kan förtydligas, färger och proportioner kan ändras, …
- Vilken abstraktion vi väljer för en modell beror på syftet med modellen
- Vad är viktigt att ta med, och vad kan utelämnas för att modellen ska vara lättare att förstå?
- Som exempel kan vi ta en vattenmolekyl

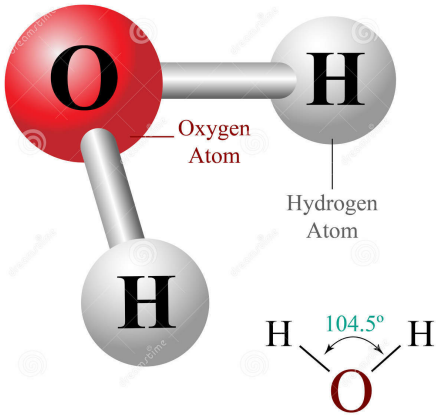
Sätt att modellera en vattenmolekyl











Exempel 2 - tunnelbanelinjer

- Tidiga kartor över tunnelbanelinjer ritades direkt på den verkliga kartan
- Det gjorde det väldigt svårt att hitta

Exempel 2 - tunnelbanelinjer

- Bakgrunden och kraven på att följa geografin är borta
- Avståndet mellan stationer är konstant oavsett det geografiska avståndet
- Uppfanns 1931 av Harry Beck

Abstraktion

- Samma sak kan modelleras med olika abstraktion beroende på vem som ska använda modellen
- För resenärer är Harry Beck's karta utmärkt – det är enkelt att hitta linjer och stationer
- En stadsarkitekt behöver i stället en karta som visar geografiskt hur linjerna går för att kunna planera nya linjer och stationer
- Då är den första kartan bättre
- För att hitta en lämplig abstraktionsnivå måste vi veta målgruppen för vår modell

Mjukvarumodellering

- I datavetenskap använder vi, bland annat, modellering till att beskriva algoritmer och system
- Modeller kan användas i stället för, eller tillsammans med, pseudokod för att beskriva hur en algoritm fungerar
- De kan också användas för att beskriva hur klasser och moduler hänger ihop i en större mjukvara

Mjukvarumodellering

- Det finns ett standardiserat sätt för modeller som beskriver mjukvara, kallat Unified Modelling Language (UML)
- Det finns olika typer av UML notation beroende på vilken typ av modell vi ska göra
- Vi ska börja med en typ av modell som kallas aktivitetsdiagram
- De beskriver vilka instruktioner (aktivitet) som en algoritm ska utföra

Algoritmer

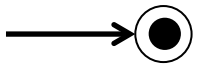
- Syftet med en algoritm är att lösa ett problem
- För att algoritmen ska vara korrekt och fungera, krävs att alla instruktioner utförs och att de körs i rätt ordning
- När vi beskriver en algoritm är det viktigt att betraktaren, som troligen ska implementera algoritmen i programkod, förstår den
- Pseudokod är ett sätt att beskriva en algoritm, men den är inte alltid lätt att förstå
- Vi kan i stället beskriva algoritmen med ett aktivitetsdiagram
- Vi kan också göra beskrivningen extra tydlig genom att både visa pseudokod och aktivitetsdiagram

Aktivitetsdiagram

- Aktivitetsdiagram beskriver flödet (control flow) i en algoritm
- Det visar vilka instruktioner som ska utföras, vilka val samt vilka upprepningar som görs
- Aktivitetsdiagram kallas ibland för flödesschema
- Det finns ett antal olika standardiserade symboler vi kan använda:

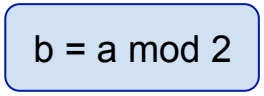
Symboler

Start på algoritmen

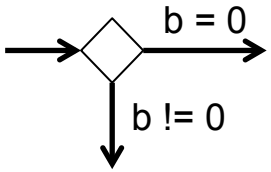


Slut på algoritmen

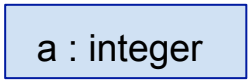
Flöde



Aktivitet



Selektion/upprepning



Inparameter/returvärde

Aktivitet

- En aktivitet kan vara en enskild instruktion, eller en serie instruktioner
- Vilken detaljnivå vi ska använda beror på syftet med diagrammet (detaljerat – övergripande)
- En detaljerad aktivitet kan vara:

  b = a mod 2

- … och en övergripande:

  Drive to work

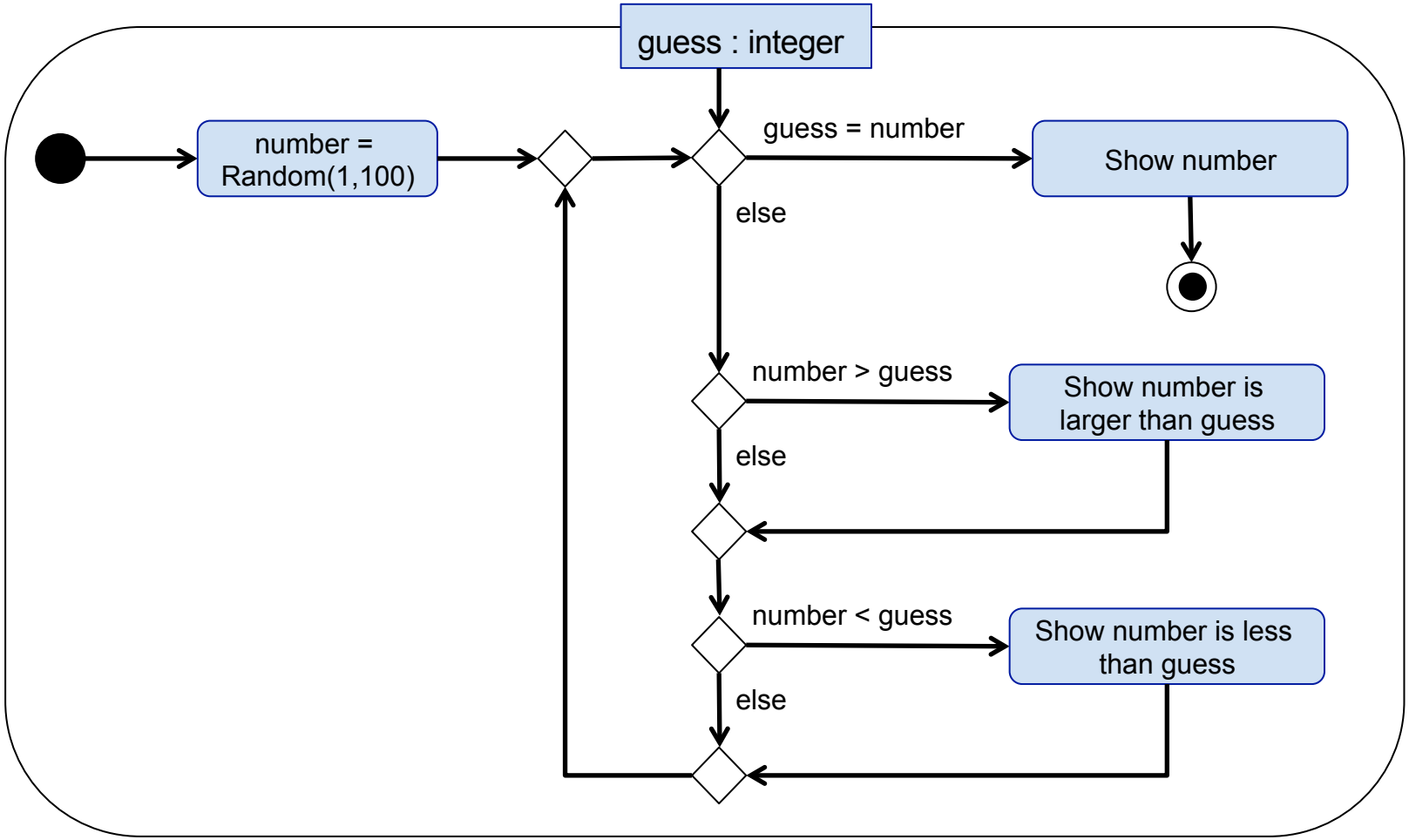
Algoritm: Gissa Talet

- I föregående föreläsning skrev vi pseudokod för spelet där användaren ska gissa ett slumpmässigt tal mellan 1 och 100:

```
10   tal = RandomNumber(1,100)
20   gissning = 0
30   while (tal != gissning)
40     tal = GissaTal()
50     if (tal > gissning)
60       print(”Talet är större än din gissning”)
70     if (tal < gissning)
80       print(”Talet är mindre än din gissning”)
90   print(”Talet är ” + tal)
```

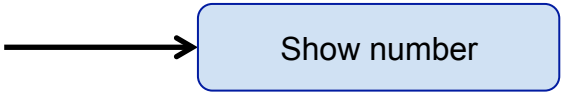
- Hur kan vi beskriva algoritmen i ett aktivitetsdiagram?

Gissa Talet

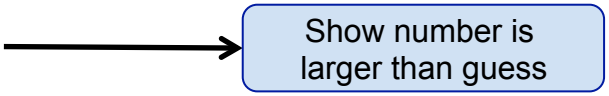

guess : integer
number = Random(1,100)
guess = number
Show number
else
number > guess
Show number is larger than guess
else
number < guess
Show number is less than guess
else

Aktivitetsdiagram

- Även om vi har standardiserade symboler för aktivitetsdiagram, kan en algoritm beskrivas på olika sätt:
  - Ska sista aktivitets-boxen returnera talet i stället för att visa det?



  - Ska Show-boxarna beskrivas i mer detalj, det vill säga exakt vad som ska skrivas ut och hur utskriften ska formateras?



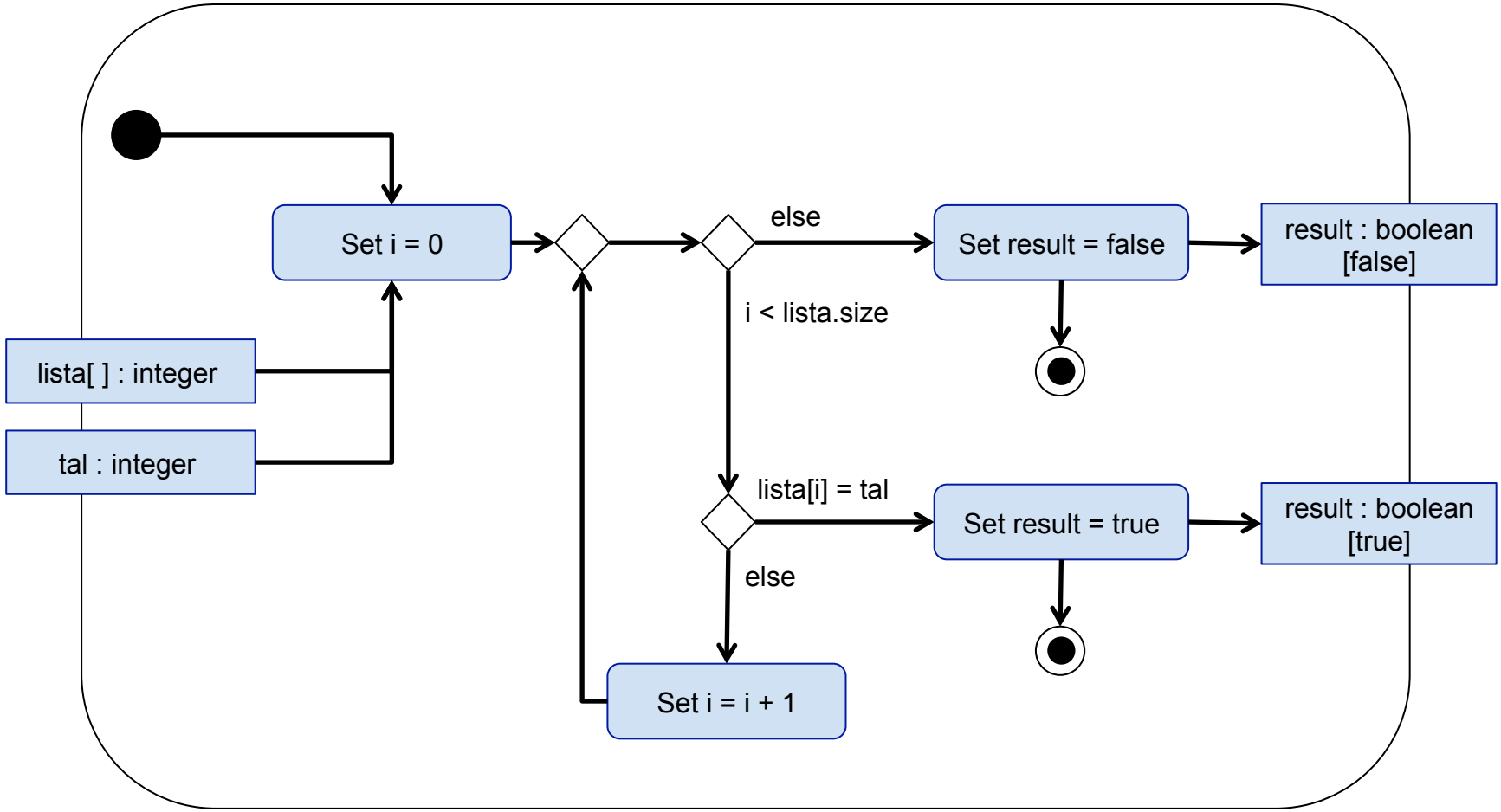
Linjärsökning

- I föregående föreläsning skrev vi pseudokod för linjärsökning:

```
10    boolean linearSearch(int[] lista, int tal)
20      i = 0
30      while (i < lista.length)
40        if (lista[i] == tal)
50          return true
60        i = i + 1
70      return false
```

- Hur kan vi modellera linjärsökning i ett aktivitetsdiagram?

Linjärsökning

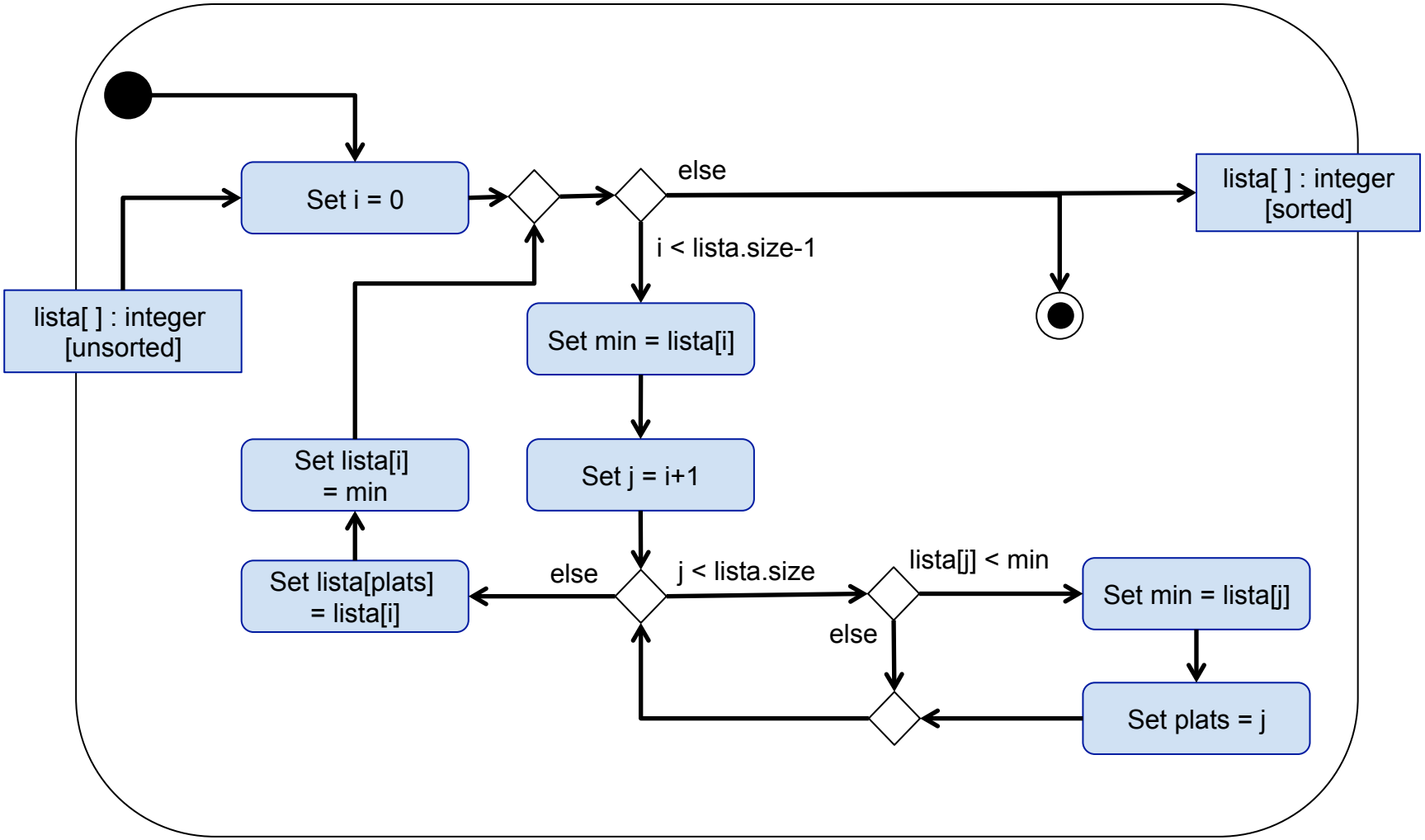


Urvalssortering

- Hur ser ett aktivitetsdiagram för urvalssortering ut?

```
10    //Definiera lista att sortera
20    lista = {9, 3, 12, 1, 7, 19, 13, 6}
30
40    //Urvalssortering
50    for (i = 0 to i < lista.length - 1)
60      min = lista[i]
70      //Gå igenom resten av listan
80      for (j = i + 1 to j < lista.length)
90        //Se om vi har nytt lägsta-värde
100       if (lista[j] < min)
110         min = lista[j]
120         plats = j
130     //Byt plats på talen
140     lista[plats] = lista[i]
150     lista[i] = min
```


Urvalssortering



För- och nackdelar?

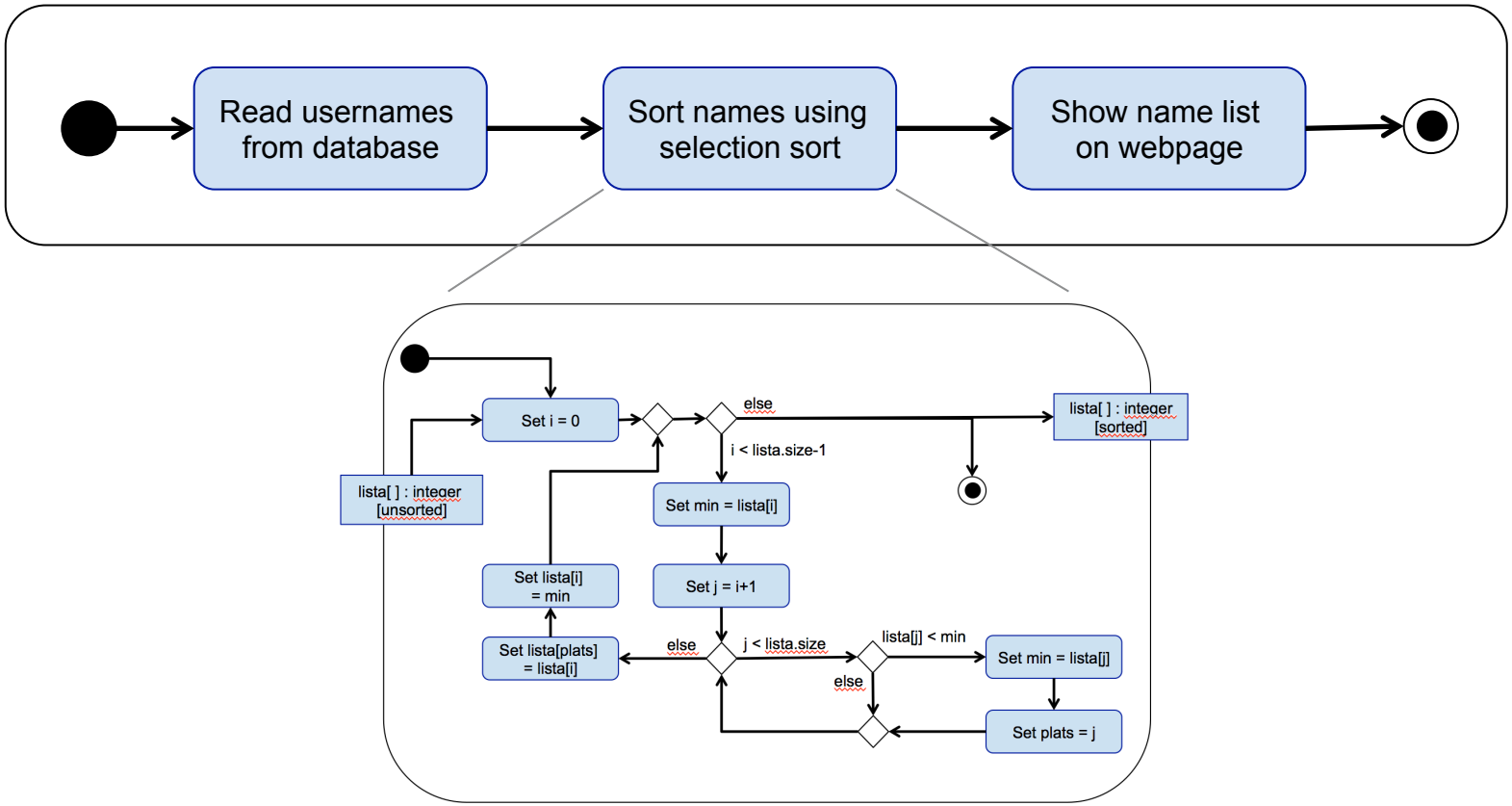
- En fördel med aktivitetsdiagram är att de är lättöverskådliga
- … och att läsaren inte behöver ha programmeringsvana

- Den stora nackdelen är att de snabbt blir stora och komplexa
- Redan urvalssorteringen, en relativt kort algoritm, ser ganska rörig ut

Hierarkiska diagram

- Ett sätt att hantera att aktivitetsdiagram blir för stora och komplexa är hierarkiska diagram
- En eller flera aktiviteter är medveten på en hög abstraktionsnivå
- Dessa beskrivs sedan i egna aktivitetsdiagram
- Detta kan med fördel användas för att beskriva delarna i en större mjukvara
- Varje del beskrivs sedan i egna diagram

Hierarkiska diagram



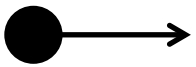
Tillstånd

- Algoritmer består av två komponenter:
  - Instruktioner
  - Data
- Aktivitetsdiagram beskriver vilka instruktioner som ska köras och i vilken ordning
- De säger (oftast) ingenting om själva datan
- Som komplement kan vi använda tillståndsdiagram

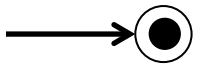
Tillstånd

- Variabler ändrar ofta sina värden under tiden programmet körs
- Till exempel en lista med tal är först osorterad, men efter en sorteringsalgoritm har körts är listan sorterad
- Ett vanligt användningsområde för tillstånd är att hålla reda på om olika saker är uppfyllda, till exempel att en dörr måste vara upplåst innan vi kan öppna den

Symboler



Starttillstånd

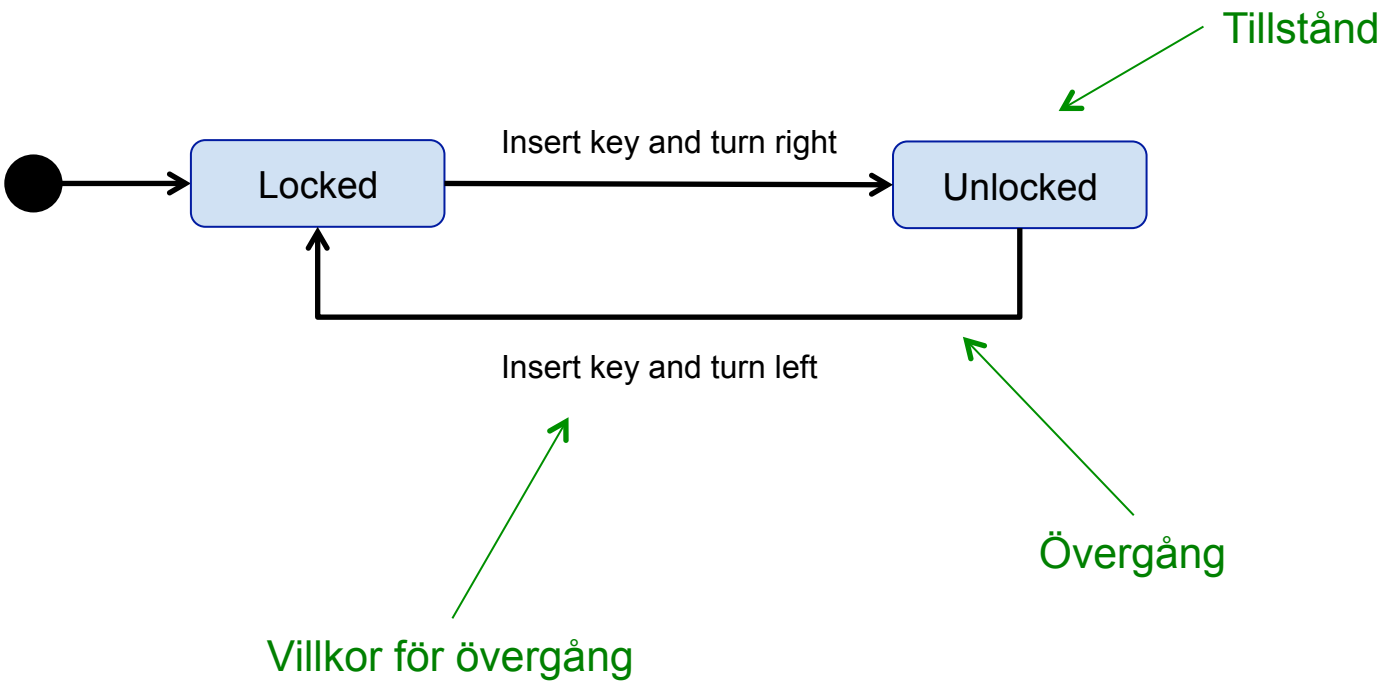


Sluttillstånd

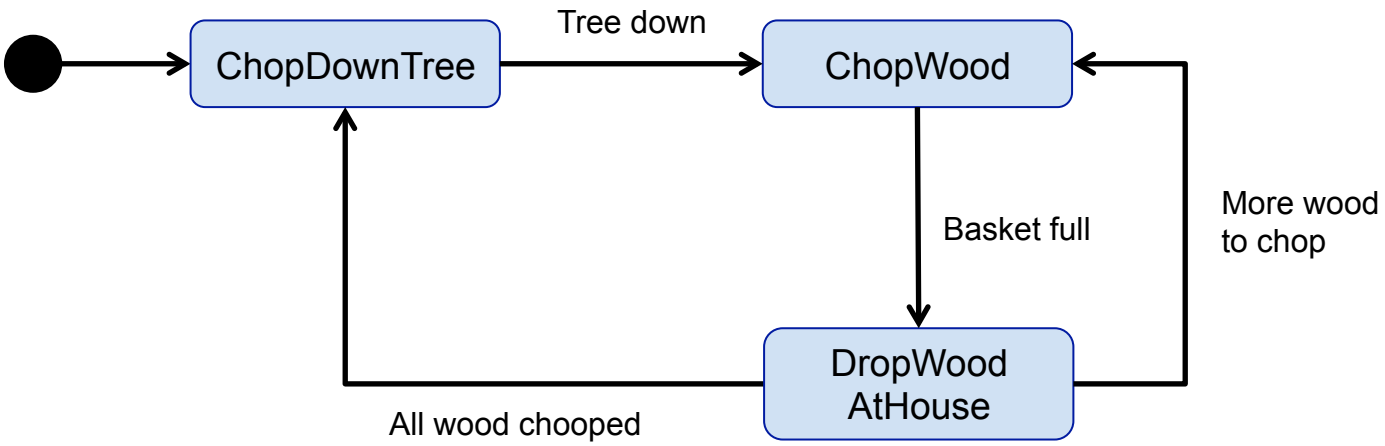
Övergång

Tillstånd

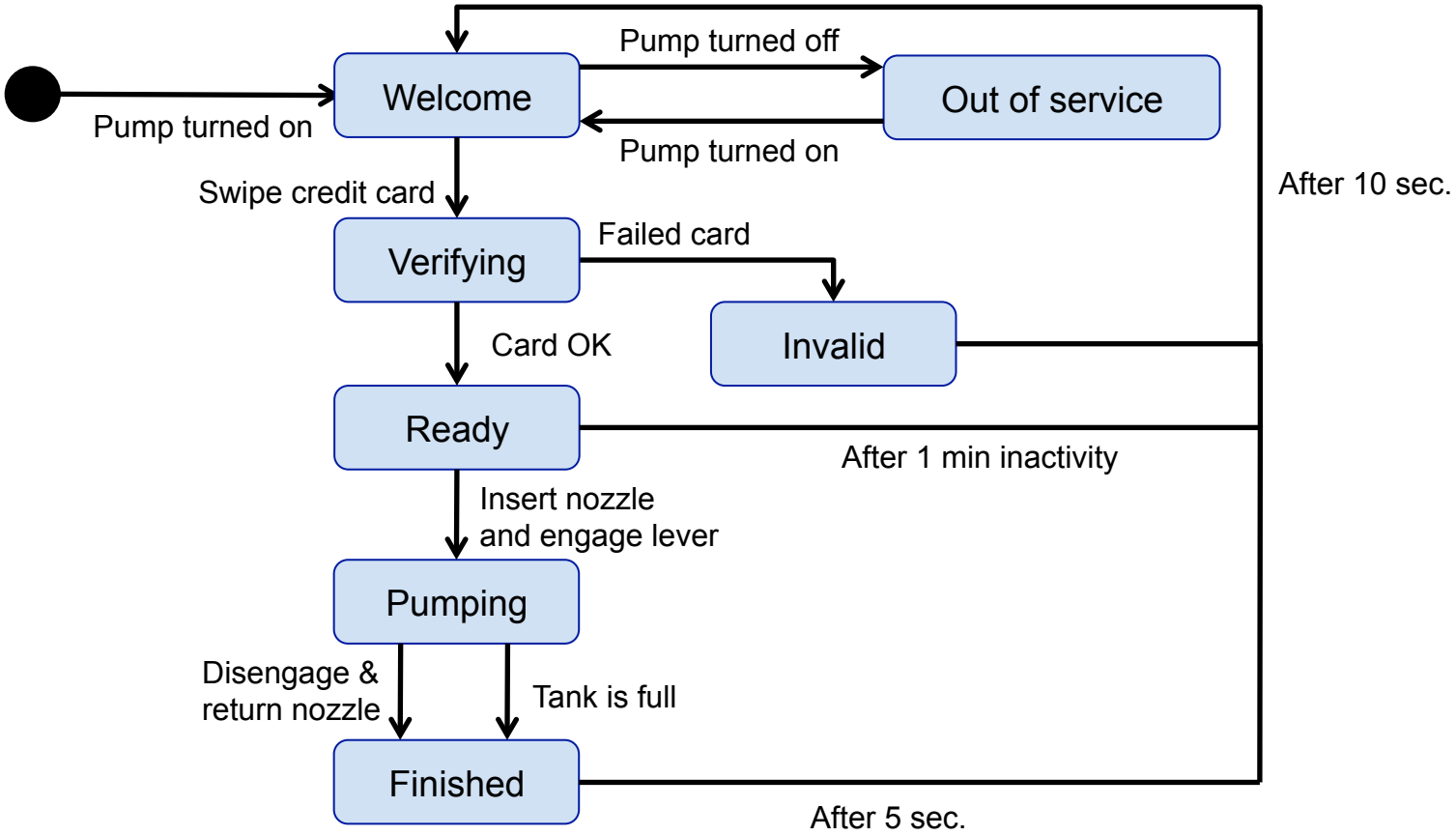
Exempel: Låst dörr



Exempel: Chuck the Woodchucker



Exempel: Tankstation



Tillståndsdiagram

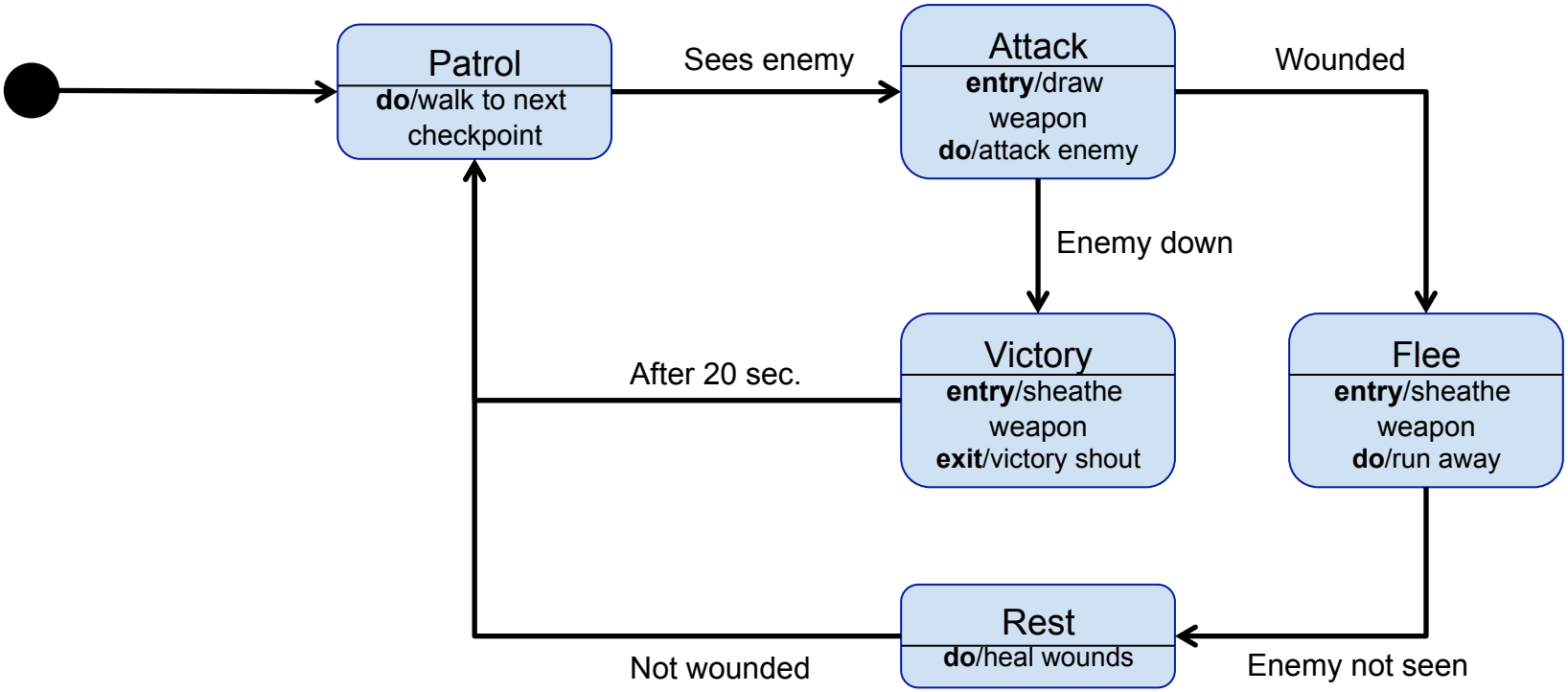
- Tillståndsdiagram är väldigt användbara för vissa typer av program
- I exemplet med tankstationen kan användaren göra olika saker beroende på pumpens tillstånd
- Detta är ett bra exempel på när tillståndsdiagram är användbara
- Ett annat exempel är karaktärer i spel

- Tillståndsdiagram tillför dock inte mycket om vi ska beskriva en sorteringsalgoritm

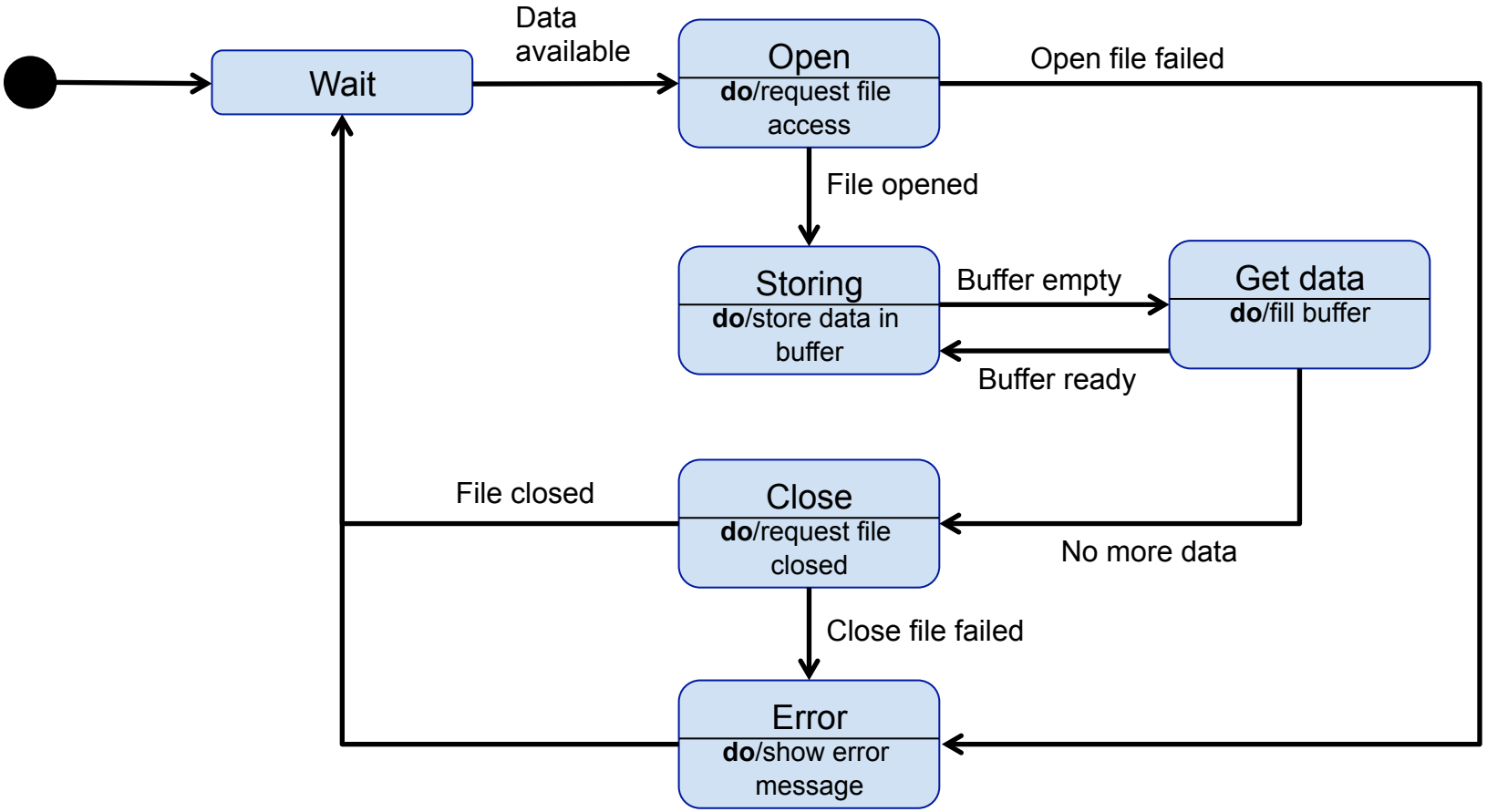
Aktiviteter i tillståndsdiagram

- Vi kan utöka tillståndsdiagram till att även innehålla aktiviteter
- Vi beskriver då vad som ska utföras i varje tillstånd
- Aktiviteter kan vara av följande typer:
  - entry – aktivitet som utförs en gång när övergång sker in till tillståndet
  - exit – aktivitet som utförs en gång när övergång sker till ett annat tillstånd
  - do – aktivitet som utförs kontinuerligt när programmet är i tillståndet

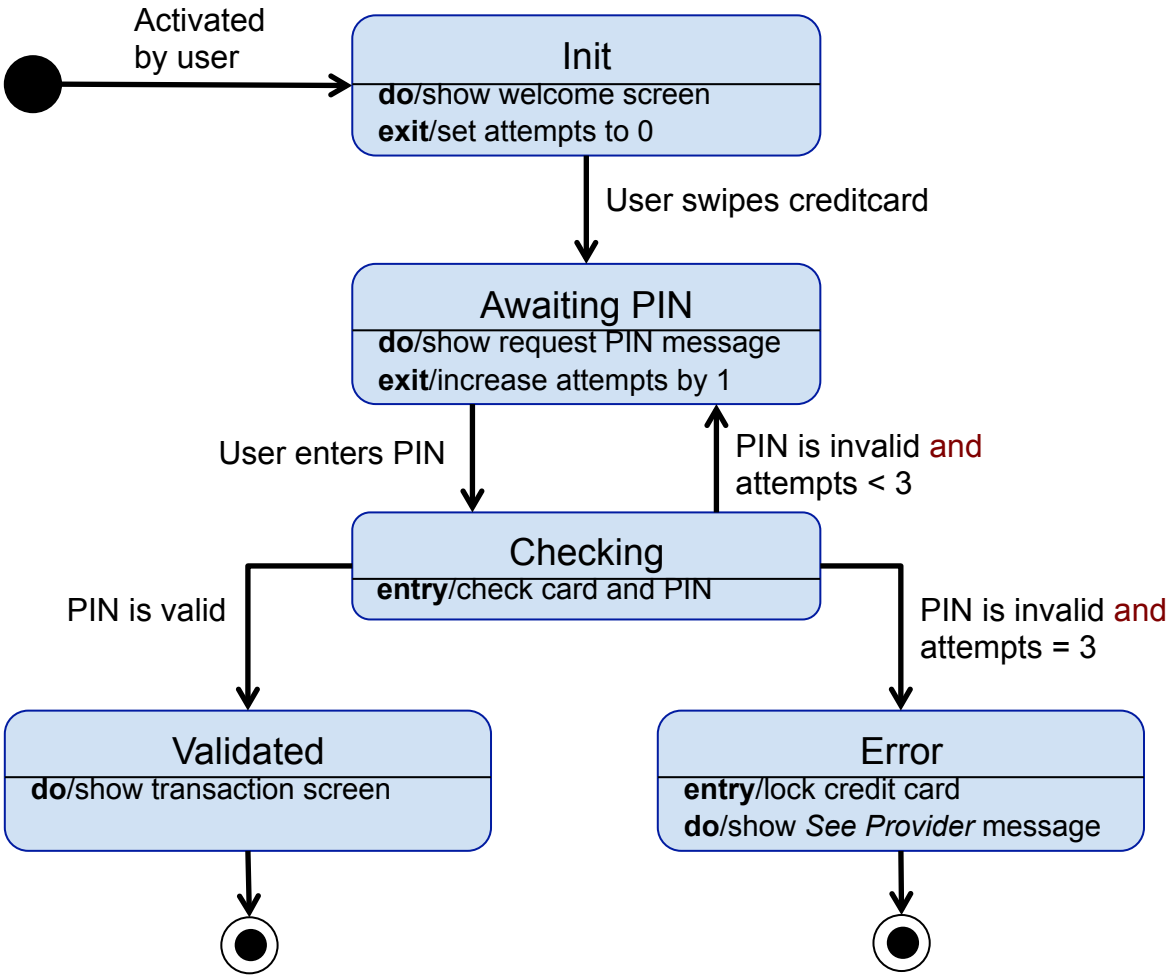
Exempel: Patrullerande vakt



Exempel: Filhanterare



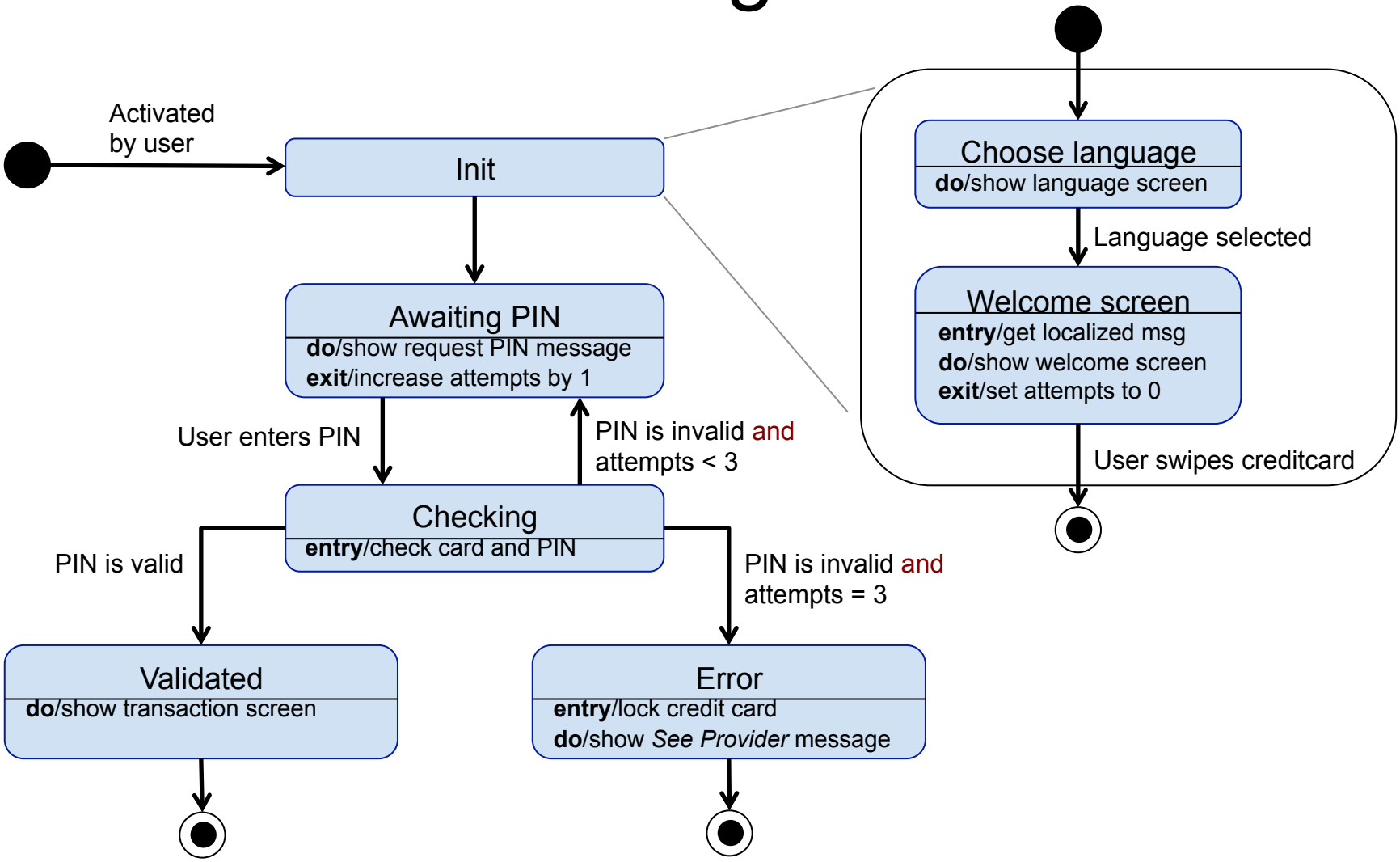
Exempel: Bankomat



Hierarkiska diagram

- Precis som med aktivitetsdiagram kan tillståndsdiagram snabbt bli stora, komplexa och svåra att överblicka
- En lösning är att även här använda hierarkiska diagram
- Ett generellt och övergripande tillstånd kan beskrivas i mer detalj i ett annat diagram

Bankomat igen



Datastrukturer

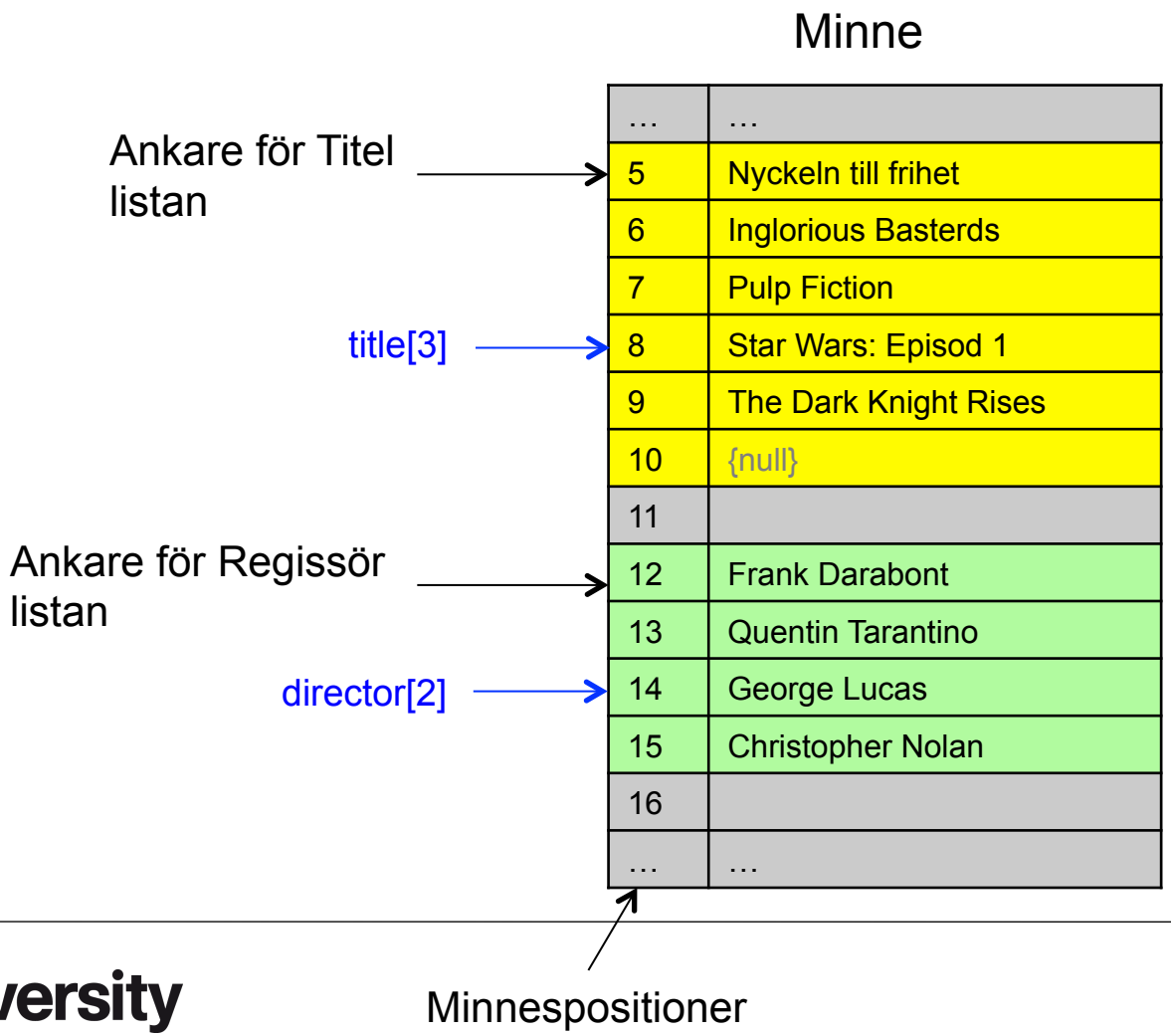
Datastruktur

- En datastruktur organiserar data så att vi effektivt kan använda, bearbeta och lagra den
- Den mest grundläggande datatypen är listor
- Som vi nämnt tidigare är en lista en behållare för fler än ett element av samma datatyp
- Det finns dock lite olika typer av listor som alla har samma funktionalitet, men ser lite olika ut internt

Array

- Array är den enklaste typen av listor
- Alla element i arrayen lagras sekventiellt i minnet
- Vi kommer åt ett element genom att ange ett index:
  - names[0]
  - names[2]
- Internt syftar namnet på en array på dess första plats i minnet, kallat basadress eller ankare
- Index anger den relativa positionen från ankaret

Arrayer i minnet



Arrayer i minnet

Om vi har 32 bitars minnesrymd tar varje minnesplats upp 32 bitar

Minne

…	…	…
5	160	Nyckeln till frihet
6	192	Inglorious Basterds
7	224	Pulp Fiction
8	256	Star Wars: Episod 1
9	288	The Dark Knight Rises
10	320	{null}
11	352	
12	384	Frank Darabont
13	416	Quentin Tarantino
14	448	George Lucas
15	480	Christopher Nolan
16	512	
…	…	…

Ändra storlek på en array

- Nackdelen med en array är att storleken (antal element) anges när arrayen skapas och kan sedan inte ändras
- För att kunna utöka storleken på en array måste vi därför:
    1. Skapa en ny array med den nya storleken
    2. Kopiera över alla element från den gamla till den nya arrayen
    3. Ta bort den gamla arrayen från minnet

Olika datatyper i en array

- Elementen i en array måste vara av samma datatyp
- Vi kan dock komma runt detta genom att skapa en klass som kan innehålla data av olika typer
- I arrayen lagrar vi sedan objekt av klassen

- Exempel:

Olika datatyper i en array

```
DataContainer
```

```
String str_data = NULL
int int_data = 0
```

```
DataContainer()
void setStrData(String value)
String getStrData ()
void setIntData(int value)
int getIntData ()
```

…	…
23	DataContainer {NULL, 5}
24	
25	DataContainer {”XGS”, 0}
26	
27	DataContainer {NULL, 18}
28	
29	
…	…

Varför börjar index på 0?

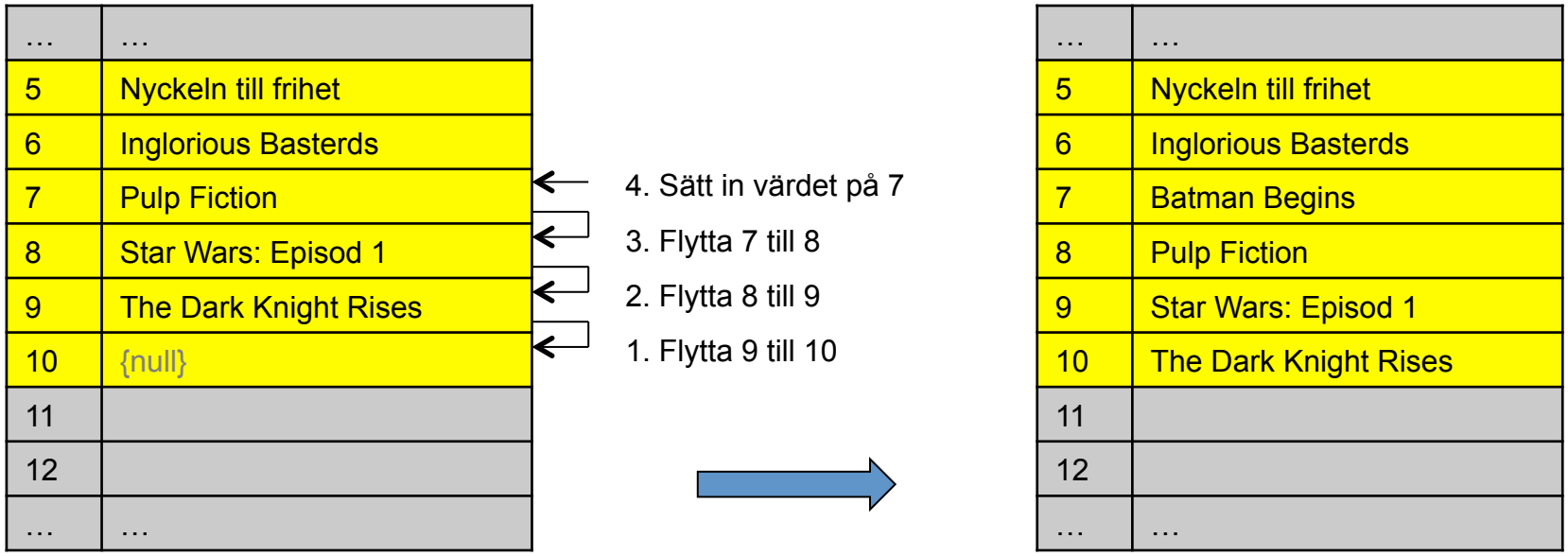
- I de flesta programspråk har första platsen i en array index 0 (kallat zero indexing)
- Varför är detta en bra idé?

- För att hitta ett element i en array beräknas en offset från ankaret
- Om index börjar på 0 finns elementet på plats *n* på minnesadress:
  - anchor_adress + n
- Om index börjar på 1 finns elementet på plats *n* på minnesadress:
  - anchor_adress + n - 1
- Det går snabbare att göra en addition i stället för en addition och en subtraktion

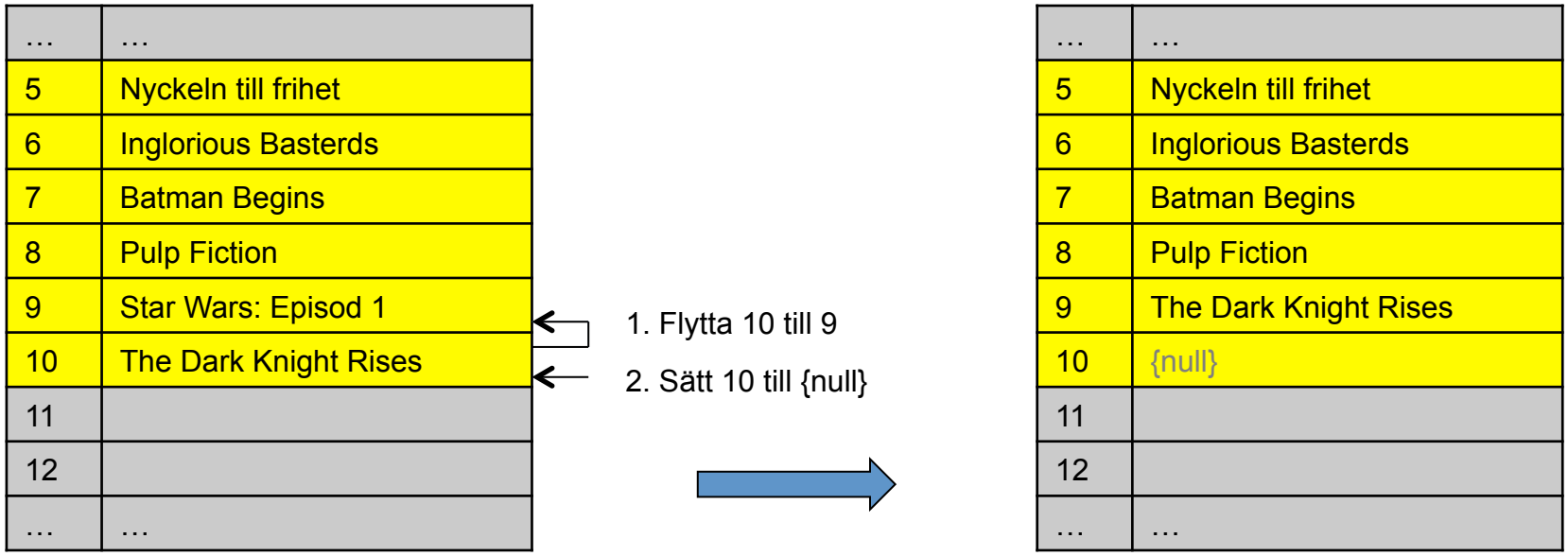
Ändra i listan

- Ta bort ett element i listan fungerar på ungefär samma sätt som att ändra storleken på en array:
  - Kopiera alla element från plats *n+1* till slutet på listan till dess föregående index
  - Sätt att sista platsen i listan är tom (null)
- Lägga till ett element i listan fungerar på likande sätt:
  - Kopiera alla element från plats *n+1* till slutet på listan till dess nästa index
  - Sätt in värdet på plats *n*
- För att lägga till ett element krävs så klart att det finns ledigt minnesutrymme i listan

Lägga till värde i en array



Ta bort värde i en array



Array

- Fördelen med arrayer är att det går snabbt att nå ett element
- Nackdelen är att vi måste kopiera ett antal element varje gång vi stoppar in ett nytt element, tar bort ett element eller ändrar storleken
  - Förutom stoppa in/ta bort i slutet på listan

Länkad lista

- I en länkad lista sparas varje element med sitt värde, och en pekare (minnesadress) till nästa element i listan
- Vi får då en kedja av element
- En länkad lista kräver mer minnesutrymme än en array, då vi behöver spara både värdet och pekare för varje element

Länkad lista i minnet

Nå ett element

- I en array går det snabbt att nå ett element – vi behöver bara beräkna en offset från ankaret så får vi minnesadressen för elementet
- För att nå ett element i en länkad lista måste vi börja på första elementet, och följa pekarna till elementet vi söker
- För att nå element två måste vi alltså göra följande:
  - Gå till element 0 och hämta ut dess pekare
  - Gå till element 1 genom att följa pekaren, och hämta ut dess pekare
  - Gå till element 2 genom att följa pekaren, och hämta ut värdet

Ändra i listan

- För att lägga till ett element i listan måste vi göra följande:
  - Skapa ett nytt element i minnet
  - Sätt pekaren till att peka på element *n*
  - Ändra pekaren för element *n-1* till att peka på det nya elementet
- Ta bort fungerar på liknande sätt:
  - Sätt pekaren på element *n-1* till element *n+1*
  - Ta bort element *n* från minnet

Lägga till värde i en länkad lista

Ta bort värde i en länkad lista

Att tänka på

- Vissa programspråk, till exempel Java, har en garbage collector
- Den frigör automatiskt minne som inte längre används
- Detta görs genom att kontrollera om programmet har en referens till alla objekt i minnet
- Saknas referens, frigörs minnet som ett objekt använder
- Om vi skulle råka sätta pekaren på element 2 till *null*, tas hela listan från plats 3 bort!
- Vi måste med andra ord vara nog med att alltid hålla en referens när vi lägger till/tar bort element i en länkad lista

Länkad lista

- Fördelen med länkade listor är att storleken är dynamisk – ingen kopiering krävs för att ändra den
- Det går också väldigt snabbt att lägga till ett element först i listan, eller ta bort första elementet i listan
- För att plocka bort/lägga till ett element mitt i listan måste vi följa ett antal pekare, vilket dock troligen går snabbare än samma operation i en array (som kräver kopiering)
- Nackdelen är att länkade listor kräver mer minne (värde + pekare för varje element), och att programmet behöver följa ett antal pekare för att nå ett element

Dubbellänkad lista

- För att nå ett element *n* i en länkad lista måste vi följa *n-1* pekare
- Om vi är på element *n* och vill gå tillbaks till element *n-1*, måste vi börja om på 0 och följa alla pekare till element *n-1*
- Det tar också tid att lägga till/ta bort i slutet på en lista, vilket är en ofta använd operation
- För att komma runt detta kan vi använda en dubbellänkad lista – varje element har, precis som tidigare, en pekare till nästa element men även en pekare till föregående element
- Ankarets föregående-pekare går till sista elementet så att vi snabbt kan nå slutet i listan
- Nackdelen med dubbellänkade listor är så klart att de kräver mer minne

Länkad lista eller array?

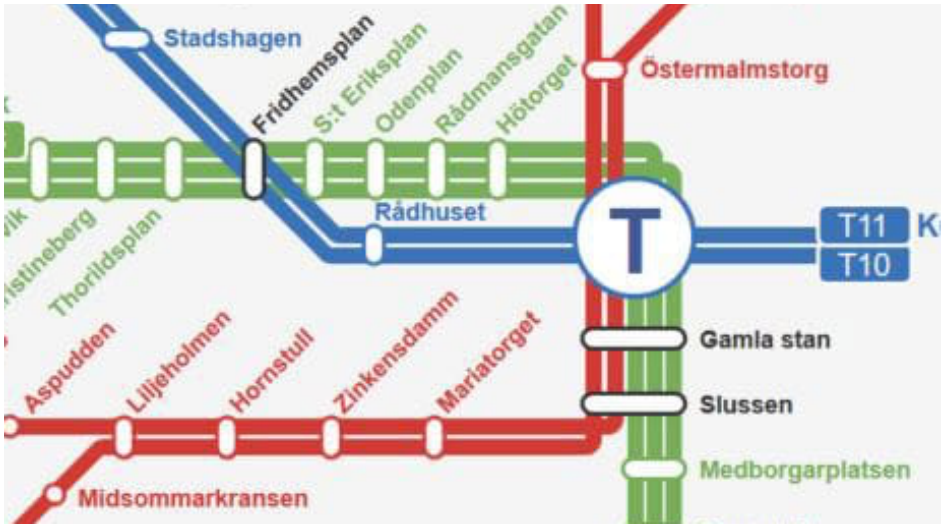
- Vilken teknik vi ska välja beror på vad vi ska använda vår lista till
- Behöver inte listan ändras alls eller väldigt sällan är array bäst
- Om vi ofta lägger till och/eller plockar bort element först i listan har länkad lista en fördel
- Om vi ofta och snabbt behöver nå element på olika positioner i listan är array bäst
- Minneskraven är oftast inget problem eftersom moderna system har mycket minne, och en pekare tar väldigt lite plats

Cache

- De flesta moderna system har förutom internminnet (RAM) också ett cacheminne
- Cacheminnet är väldigt litet jämfört med RAM, men mycket snabbare
- För att processorn ska kunna arbeta med data, kopieras datan först från RAM till cacheminnet
- Vi får en prestandaförbättring om vi kan minimera hur ofta vi behöver kopiera mellan RAM och cache
- Array har bättre cache consistency än en länkad lista, eftersom systemet ofta kopierar ett helt block från RAM till cache
- Därför ger ofta arrayer bäst prestanda, även om det inte alltid känns som att de borde göra det
- För att veta vilken som är bäst måste vi testa båda och mäta prestanda
- För att vi ska kunna se en skillnad krävs dock att vi har väldigt stora datamängder

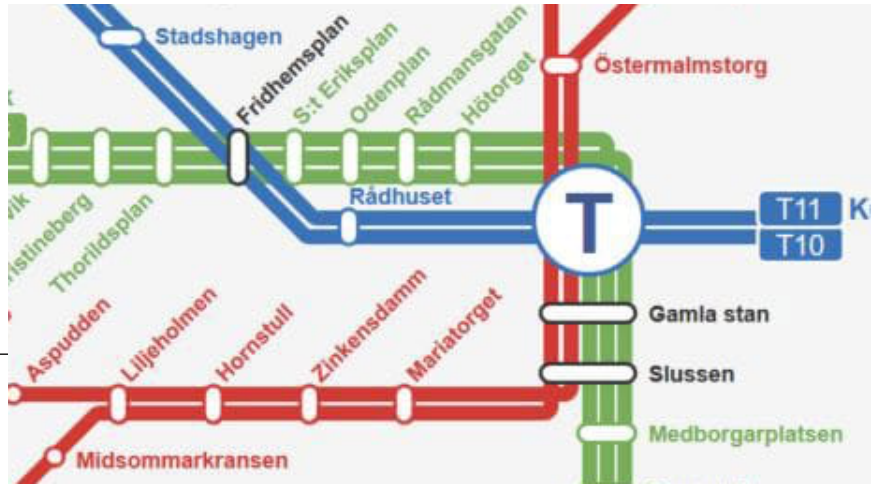
Graf

- Många saker och koncept passar inte riktigt att modellera som listor
- Som exempel kan vi ta Stockholms tunnelbana:



Graf

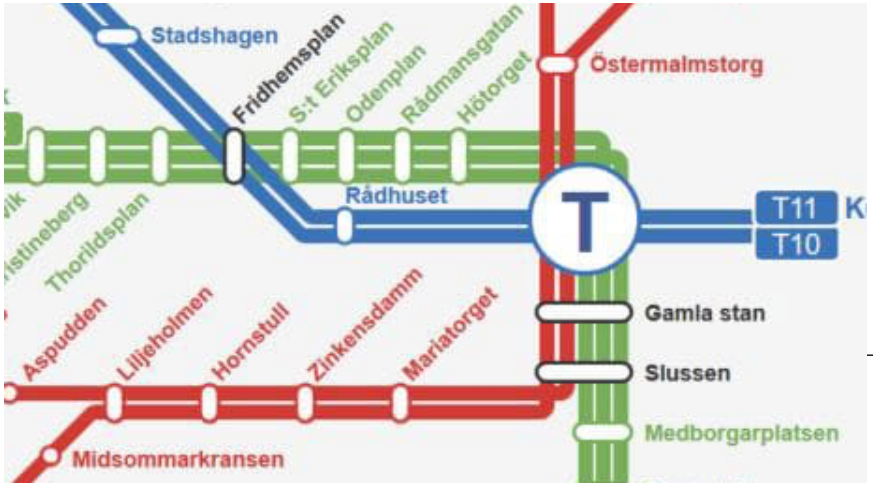
- Varje station trafikeras av röd, grön eller blå linje
- Vissa stationer trafikeras av fler än en färg
- Varje färg har också en eller flera linjer
- Det finns också slutstationer där vi inte kan fortsätta resa
- Varje element har alltså *0…N* pekare till nästa element



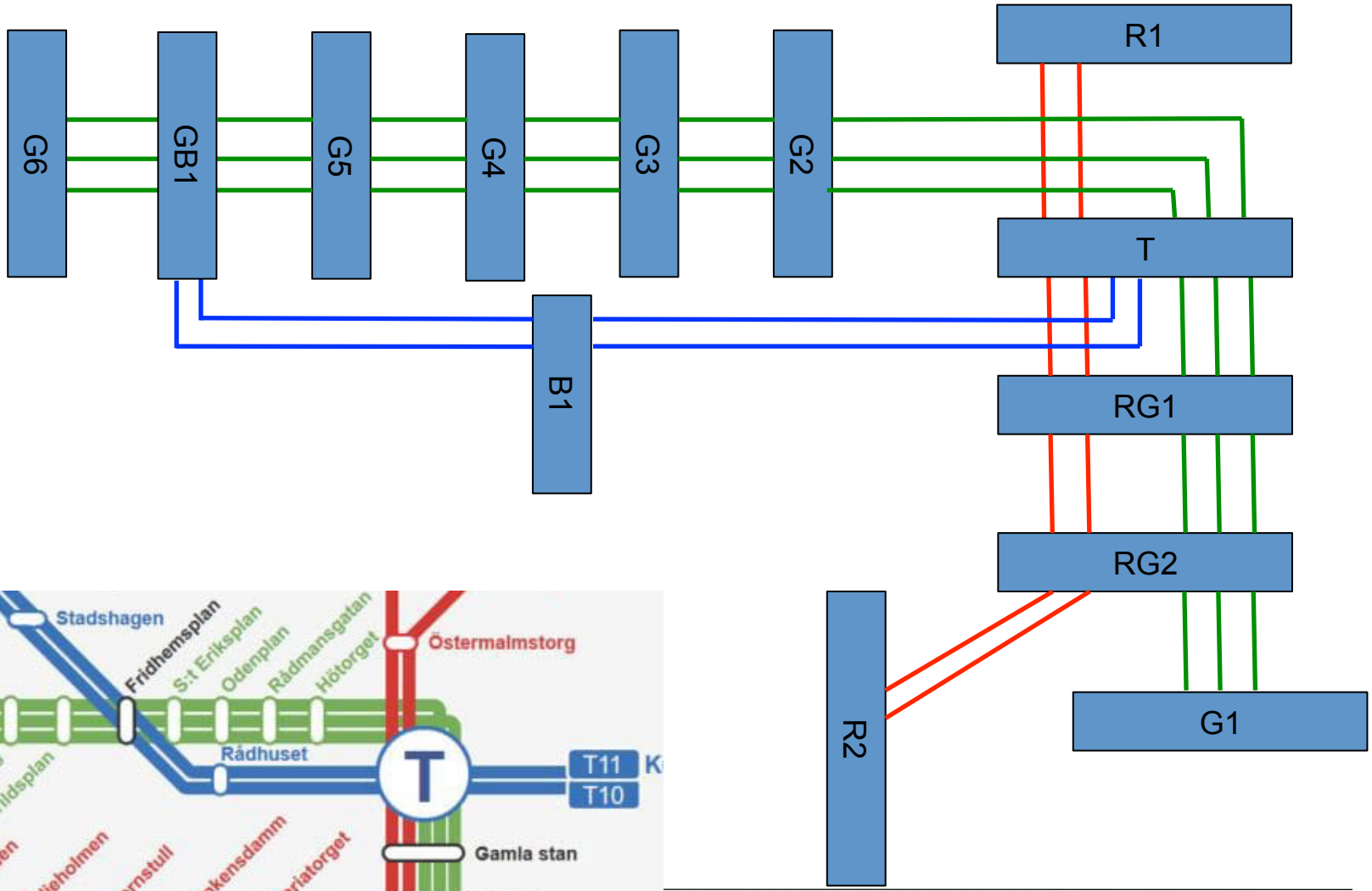
Graf

- För att modellera tunnelbanan är det bättre att använda en graf
- En graf består av ett antal noder, och bågar mellan noderna
- I en riktad graf kan en båge bara följas i en riktning (den vanligaste typen av grafer)

Thorildsplan
Fridhemsplan
S:t Eriksplan
Odenplan
Rådmangsg.
Hötorget
Östermalmstorg
T
Rådhuset.
Gamla Stan
Slussen
Mariatorget
Medborgarpl.



Förenklad namngivning



Graf över tunnelbanan

- En graf beskrivs som G = (V, E)
- V är noderna (nodes/vertices)
- E är bågarna (edges/arcs)
- Den del av tunnelbanan som vi modellerat kan beskrivas som:

  V={T,R1,R2,RG1,RG2,B1,G1,G2,G3,G4,G5, G6,GB1}
  E=({T,RG1},{RG1,RG2},{RG2,G1},{RG2,R2},{T,R1},{T,G2},{G2,G3},
  {G3,G4},{G4,G5},{G5,GB1},{GB1,G6},{T,B1},{B1,GB1}

- Eftersom vi inte namngett alla olika dellinjer anges bara en båge mellan t.ex. RG1 och RG2 i stället för fem

Graf

- Grafer passar bra för att modellera många verkliga problem:
  - Flyglinjer
  - Vägnät
  - Kemiska strukturer
  - Brädet och pjäser i schack
  - Elektroniska kretsar
  - Elnät
  - Sociala nätverk
  - …

Terminologi

- Angränsande (adjacency):
  - Två noder A och B är angränsande om det finns en båge mellan A och B
- Gradtal (degree):
  - Gradtal på en nod A är antalet bågar som är kopplade till A. Vi skiljer mellan in-gradtal (antal noder som är kopplade till A) och ut-gradtal (antal noder som A är kopplad till)
- Ordning (order):
  - Antalet noder i grafen
- Storlek (size):
  - Antalet bågar i grafen

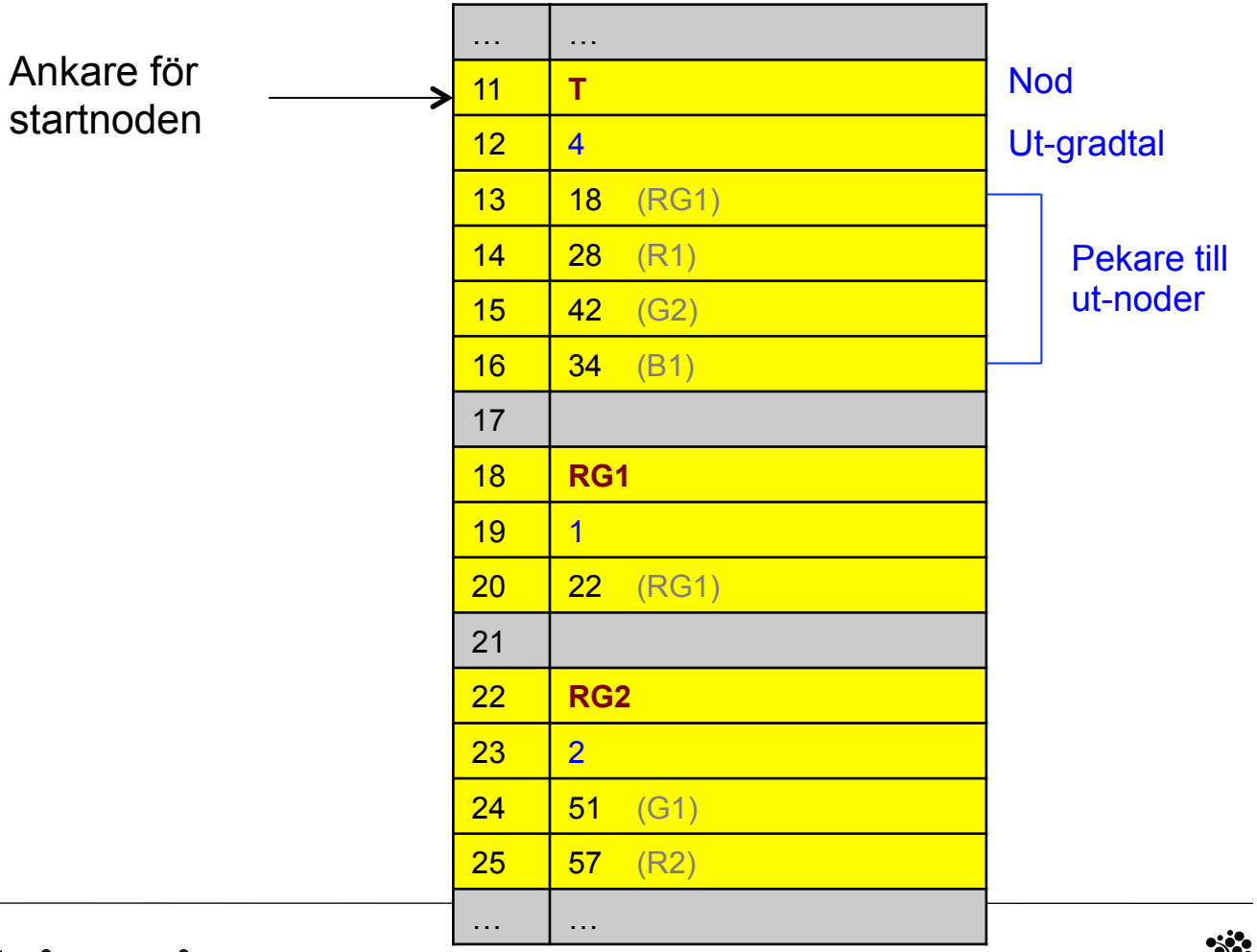
Terminologi

- Väg (path):
  - En sekvens av bågar som knyter samman en sekvens av noder. Beskriver hur vi kan ta oss från en nod A till en annan nod B
- Enkel väg (simple path):
  - En väg där varje både och nod är unik, utom den första och sista noden som får vara samma
- Cykel (cycle):
  - En enkel väg där första och sista noden är samma

Lagring

- Grafer lagras i minnet på liknande sätt som länkade listor
- En länkad lista kan ses som en graf med in- och ut-gradtal på 1

Lagring


Ankare för startnoden
…
…
11
T
12
4
13
18 (RG1)
14
28 (R1)
15
42 (G2)
16
34 (B1)
17
18
RG1
19
1
20
22 (RG1)
21
22
RG2
23
2
24
51 (G1)
25
57 (R2)
…
…
Nod
Ut-gradtal
Pekare till ut-noder

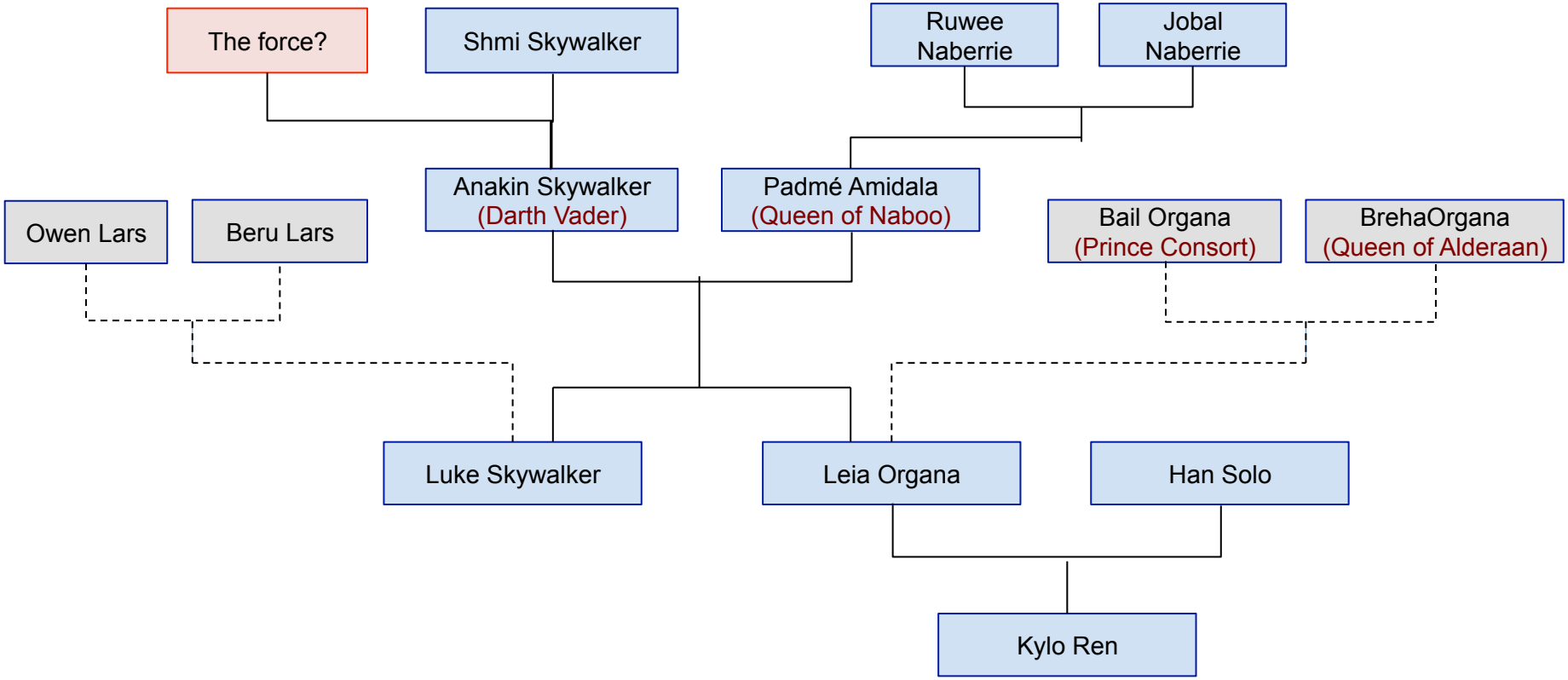
Träd

- Träd är ett specialfall av graf där noderna är organiserade hierarkiskt i nivåer
- Träd är väldigt användbara för många tillämpningar och är ofta förekommande i mjukvaruvärlden
- Exempel:
  - Organisationsschema för företag
  - Biologi – Carl Von Linnés taxonomi
  - Språkanalys (meningsuppbyggnad)
  - Familjeträd
  - Kompilatorer
  - 3D grafik
  - …

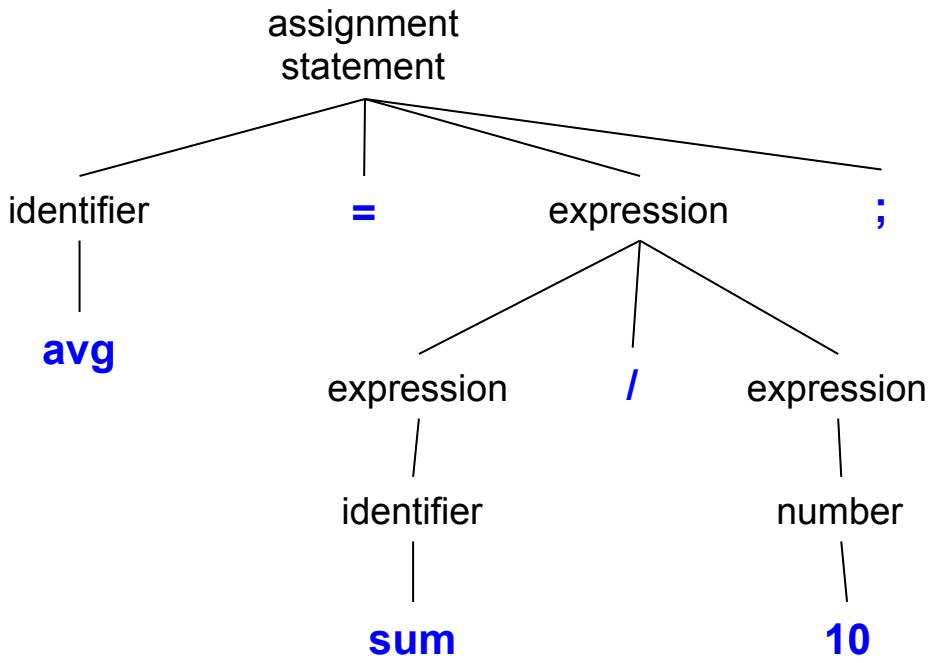
Definition

- Alla träd börjar på en nod utan ingående kopplingar, kallad roten eller rotnoden
- En nod kan endast kopplas till noder i underliggande nivå, vilket medför att inga cykler finns
- Alla noder förutom roten har in-gradtal på 1
  - Det kan förekomma träd med högre in-gradtal
- Det finns en väg från roten till alla andra noder i trädet

Familjeträd



Parse tree (kompilator)



avg = sum / 10;