

F13 - Generella teorier

5DV149 Datastrukturer och algoritmer

Kapitel 9

Niclas Börlin
niclas.borlin@cs.umu.se

2024-05-03 Fre

Innehåll

- ▶ Abstrakta datatyper
 - ▶ **Lista, Cell, Fält, Tabell, Stack, Kö, Träd, Graf, Mängd, Lexikon, Prioritetskö, Heap, Trie, Binärt sökträd, Relation, ...**
- ▶ Vad finns det för **generella** teorier?
- ▶ Vad måste vi ta hänsyn till om vi vill göra en **ny datatyp**?
 - ▶ Exempel på datatyper att fundera på:
 - ▶ Kurs
 - ▶ Dokument
 - ▶ Test
 - ▶ Arkiv
 - ▶ Modell
 - ▶ Biograf

Abstrakta datatyper (ADT)

- ▶ Ett koncept för att kunna diskutera och jämföra olika typer av datastrukturer
- ▶ Hög abstraktionsnivå:
 - ▶ Intresserad av struktur och organisation, inte implementation
- ▶ Operationerna ger datatypen karaktär och specifikationen visar datatypens uttrycksfullhet

Operationskategorier

- ▶ Operationerna kan indelas i olika kategorier:
 1. Konstruktörer
 2. Inspektörer
 3. Modifikationer
 4. Navigatorer
 5. Komparatorer
 6. Destruktörer

1. Konstruktörer

- ▶ **Skapar** och returnerar ett objekt av aktuell ADT
 - ▶ **Grundkonstruktörer** saknar argument av aktuell ADT:
 - ▶ `Stack-Empty()`
 - ▶ `Queue-Empty()`
 - ▶ `Binary-Search-Tree-Make()`
 - ▶ `Array-Create()`
 - ▶ `Link-Create()`
 - ▶ **Vidareutvecklande** konstruktörer tar **ett** argument av aktuell ADT:
 - ▶ `List-Insert(val, pos, List)`
 - ▶ `Stack-Push(val, Stack)`
 - ▶ **Kombinerande** konstruktörer tar **flera** argument av aktuell ADT:
 - ▶ `Set-Union(Set, Set)`

2. Inspektorer

- ▶ Undersöker ett objekts **inre uppbyggnad**:
 - ▶ **Avläser** eller sonderar **elementvärden** eller **strukturella egenskaper**:
 - ▶ `Array-Inspect-value()`
 - ▶ `Stack-Top()`
 - ▶ `Table-Lookup()`
 - ▶ `Set-Choose()`
 - ▶ `Array-Has-Value()`
 - ▶ `Set-Member-Of()`
 - ▶ **Test** av olika extremfall för struktur eller värden:
 - ▶ `Binary-Tree-Has-Left-Child()`
 - ▶ `Stack-Isempty()`

3. Modifikatorer

- ▶ **Ändrar** ett objekts **struktur** och/eller **elementvärden**:
 - ▶ Insättning, borttagning, tilldelning, omstrukturering:
 - ▶ `Array-Set-Value()`
 - ▶ `Table-Remove()`
 - ▶ `Stack-Push()`
 - ▶ `Stack-Pop()`
 - ▶ `Set-Insert()`
- ▶ En del operationer kan räknas **både** som konstruktor och modifikator
 - ▶ `Stack-Push()`

4. Navigatorer

- ▶ Används för att **orientera sig** i ett objekts **struktur**:
 - ▶ Landmärken (kända positioner)
 - ▶ `List-First()`
 - ▶ `List-End()`
 - ▶ `Tree-Root()`
 - ▶ Lokala förflyttningar
 - ▶ `List-Next()`
 - ▶ Traverseringar
 - ▶ `Binary-Tree-Left-Child()`

5. Komparatorer

- ▶ **Jämför** objekt av aktuell ADT med varandra:
 - ▶ `Link-Equal ()`
 - ▶ `Set-Subset ()`

6. Destruktorer

- ▶ **Lämnar tillbaka** resurserna som en datatyp använt
 - ▶ `List-Kill()`
 - ▶ `Array-Kill()`

Gränsytor — exempel

- Vilka kategorier har följande operationer?

```
abstract datatype DList(val)
auxiliary pos
Empty() → DList(val)
Isempy(l: DList(val)) → Bool
First(l: DList(val)) → pos
Next(p: pos, l: DList(val)) → pos
Isend(p: pos, l: DList(val)) → Bool
Inspect(p: pos, l: DList(val)) → val
Insert(v: val, p: pos, l: DList(val))
    → (DList(val), pos)
Remove(p: pos, l: DList(val)) → (DList(val), pos)
Kill(l: DList(val)) → ()
```

Gränsytor — exempel

- Vilka kategorier har följande operationer?

```
abstract datatype DList(val)
auxiliary pos
  Empty() → DList(val)
  Iempty(l: DList(val)) → Bool
  First(l: DList(val)) → pos
  Next(p: pos, l: DList(val)) → pos
  Isend(p: pos, l: DList(val)) → Bool
  Inspect(p: pos, l: DList(val)) → val
  Insert(v: val, p: pos, l: DList(val))
    → (DList(val), pos)
  Remove(p: pos, l: DList(val)) → (DList(val), pos)
  Kill(l: DList(val)) → ()
```

►

	Konstruktor	Inspektor	Modifikator	Navigator	Komparator	Destruktor
Empty						
Iempty						
First						
Next						
Isend						
Inspect						
Insert						
Remove						
Kill						

Gränsytor — exempel

- Vilka kategorier har följande operationer?

```
abstract datatype DList(val)
auxiliary pos
Empty() → DList(val)
Isempy(l: DList(val)) → Bool
First(l: DList(val)) → pos
Next(p: pos, l: DList(val)) → pos
Isend(p: pos, l: DList(val)) → Bool
Inspect(p: pos, l: DList(val)) → val
Insert(v: val, p: pos, l: DList(val))
    → (DList(val), pos)
Remove(p: pos, l: DList(val)) → (DList(val), pos)
Kill(l: DList(val)) → ()
```

►

	Konstruktor	Inspektor	Modifikator	Navigator	Komparator	Destruktor
Empty	X					
Isempy		X				
First				X		
Next				X		
Isend		X				
Inspect		X				
Insert	X		X			
Remove	(X)		X			
Kill						X

Slutna kontra öppna datatyper

- ▶ Kommande exempel handlar om operationer på datatyper
 - ▶ Jämförelser, kopiering, m.m.
- ▶ Det spelar då stor roll om datatypen är **öppen** (navigeringsbar) eller inte
 - ▶ Går det att avläsa ett element **inuti** datatypen?
- ▶ Jämför att beräkna längden på **Lista** respektive **Kö**

Längd på Lista

- Vill vi beräkna längden på Listan är det bara att **traversera** den och **räkna** elementen

```
Algorithm List-length(l: Directed list)  
  // Input: A list  
  // Output: The number of elements in the list  
  len ← 0  
  p ← First(l)  
  while not Isend(p, l) do  
    len ← len + 1  
    p ← Next(p, l)  
  return len
```

- Listan **förändras inte** av operationen

Längd på Kö (1)

- För Kö är enda sättet att se om kön har ett andra elementet att ta bort det första

```
Algorithm Queue-length-bad(q: Queue)  
// Compute the length of a queue  
// This code will destroy the queue  
  
len ← 0  
while not Isempty(q) do  
    // Remove one element from q  
    q ← Dequeue(q)  
    // Increase count  
    len ← len + 1  
  
// Return the length  
return len
```

- Denna naiva lösning kommer att förstöra kön

Längd på Kö (2)

- ▶ Om vi måste återställa in-kön blir operationen mer komplex
- ▶ I princip blir operationen att generera en **kopia** av in-kön och samtidigt utföra den egentliga operationen (räkna element)
- ▶ Vi kan endera **återställa** in-parametern...

```
Algorithm Queue-length(q: Queue)
// Compute the length of a queue
// The input queue is modified in the process, but
// is restored before return

// This will be the replicated queue
r ← Queue-empty()
len ← 0
while not Isempty(q) do
    // Copy first item in queue from q to r
    r ← Enqueue(Front(q), r)
    // Remove from q
    q ← Dequeue(q)
    // Increase count
    len ← len + 1
// Restore input queue
q ← r
// Return length
return len
```

Längd på Kö (3)

- ...eller skriva funktionen så att den **returnerar** en (återställd) kopia av in-kön

```
Algorithm Queue-length(q: Queue)
// Compute the length of a queue. Returns the length
// and a copy of the input queue.

// This will be the replicated queue
r ← Queue-empty()
len ← 0
while not Iempty(q) do
    // Copy first item in queue from q to r
    r ← Enqueue(Front(q), r)
    // Remove from q
    q ← Dequeue(q)
    // Increase count
    len ← len + 1
// Return the length and the replicated queue
return (len, r)
```

Uttrycksfullhet (1)

- ▶ Gränsytans specifikation visar datatypens **uttrycksfullhet**
- ▶ Frågor att fundera kring vid skapandet av en ADT:
 - ▶ Vilken är **värde**mängden?
 - ▶ Vilken **typ** av värden vill jag lagra?
 - ▶ Vilka **interna** resp. **externa** egenskaper har objekten?
 - ▶ Sorterad eller osorterad?
 - ▶ Vad ska man **göra** med objekten?
 - ▶ Specificera en **gränsyta** informellt och formellt
 - ▶ Överväg olika **konstruktions/implementationsmöjligheter**
 - ▶ Lista?
 - ▶ Fält?
 - ▶ Träd?
 - ▶ Tabell?

Uttrycksfullhet (2)

- ▶ Datatypsspecifikationen har två roller:
 - ▶ Beskriva datatypens **egenskaper**
 - ▶ Fungerar som en **regelsamling** för användningen av datatypen
- ▶ Specifikationens uttrycksfullhet kan mätas med tre begrepp:
 - ▶ Nivå 0: (Objektfullständighet) Vi vill kunna **skilja** mellan objekt
 - ▶ Nivå 1: (Algoritmfullständighet) Vi vill kunna **jämföra** objekt
 - ▶ Nivå 2: (Rik gränsyta) Vi vill kunna **kopiera objekt effektivt**

Nivå 0: Objektfullständighet

- ▶ Det ska vara möjligt att **konstruera** och **skilja** mellan **alla** objekt som hör till datatypen
- ▶ Vi måste kunna **inspektera allt** som vi stoppar in i datatypen
- ▶ Vi måste kunna **skilja objekt åt** om vi vet hur dom **borde** skilja sig åt
- ▶ Summering: **Stoppa inte in data du inte kan läsa av!**

Nivå 0, exempel

- ▶ Datatypen **Student** med konstruktorn `Create (name, address)`, men som enda inspektor `Inspect-Name()` uppfyller **inte** nivå 0
 - ▶ Kan inte skilja på två studenter med lika namn men olika adress

Nivå 0, exempel

- ▶ Datatypen **Student** med konstruktorn `Create (name, address)`, men som enda inspektor `Inspect-Name()` uppfyller **inte** nivå 0
 - ▶ Kan inte skilja på två studenter med lika namn men olika adress
- ▶ En **Tabell**-gränsyta med endast funktionerna `Empty()`, `Insert()` och `Max()` (största tabellvärdet) uppfyller **inte** nivå 0
 - ▶ Kan inte skilja på
 - ▶ A: ((Rosor, 46), (Krysantemum, 28))
 - ▶ B: ((Tussilago, 46), (Persilja, 15))

Nivå 0, exempel

- ▶ Datatypen **Student** med konstruktorn `Create (name, address)`, men som enda inspektor `Inspect-Name()` uppfyller **inte** nivå 0
 - ▶ Kan inte skilja på två studenter med lika namn men olika adress
- ▶ En **Tabell**-gränsyta med endast funktionerna `Empty()`, `Insert()` och `Max()` (största tabellvärdet) uppfyller **inte** nivå 0
 - ▶ Kan inte skilja på
 - ▶ A: ((Rosor, 46), (Krysantemum, 28))
 - ▶ B: ((Tussilago, 46), (Persilja, 15))
- ▶ En **Tabell**-gränsyta med endast funktionerna `Empty()`, `Insert()` och `Lookup()` uppfyller **däremot** nivå 0
 - ▶ För A och B ovan så ger `Lookup(A, Rosor)` annat resultat än `Lookup(B, Rosor)`

Nivå 1: Algoritmfullständighet (*Expressive completeness*)

- ▶ Starkare än Nivå 0: objektfullständighet
- ▶ Man ska kunna implementera **alla algoritmer** i denna datatyp
- ▶ Räcker med att visa att man kan implementera ett **likhetstest** mellan två objekt **med hjälp av specifikationens operationer**
- ▶ Alltså:
 - ▶ Nivå 1 = Nivå 0 + likhetstest
 - ▶ Det ska gå att skilja två olika objekt åt även om man **inte** vet vilka skillnader man letar efter

Blank

Nivå 1, Lista

- Går det att **navigera** i datatypen så är likhetstesten ofta enkla

```
Algorithm List-isequal(l1, l2: List)
// Compare two lists. Return True if they are equal,
// otherwise False
p1 ← First(l1) // Position in l1
p2 ← First(l2) // Position in l2

// Iterate over l1
while not Pos-isequal(p1, End(l1), l1) do
    // Is l2 shorter?
    if Pos-isequal(p2, End(l2), l2) then
        return False

    // Are the values different?
    if not Value-isequal(Inspect(p1, l1), Inspect(p2, l2)) then
        return False

    // Advance positions in both lists
    p1 ← Next(p1, l1)
    p2 ← Next(p2, l2)

// Is l2 longer?
if not Pos-isequal(p2, End(l2), l2) then
    return False

// If we reach here, l1 and l2 have equal lengths
// and equal element values
return True
```

- Notera användandet av extern funktion **Value-isequal**

Nivå 1, Kö

- Går det **inte** att navigera blir likhetstesten med komplexa

```
Algorithm Queue-isequal(q1, q2: Queue)
// Compare two queues. Return True if they are equal, otherwise False
equal ← True
// This will become replicas of the original queues
r1 ← Queue-empty(); r2 ← Queue-empty()
// As long as both queues have elements...
while not Iseempty(q1) and not Iseempty(q2) do
    // Extract and remove first elements in the input queues
    v1 ← Front(q1); q1 ← Dequeue(q1)
    v2 ← Front(q2); q2 ← Dequeue(q2)
    // Insert element in both replicas
    r1 ← Enqueue(v1, r1); r2 ← Enqueue(v2, r2)
    // Check if element values are equal
    if not Value-isequal(v1, v2) then
        equal ← False
// Now at least one queue is empty
if Iseempty(q1) != Iseempty(q2) then
    equal ← False
// Transfer remaining elements of q1
while not Iseempty(q1) do
    v ← Front(q1); q ← Dequeue(q1)
    r1 ← Enqueue(v, r1)
// Transfer remaining elements of q2
while not Iseempty(q2) do
    v ← Front(q2); q ← Dequeue(q2)
    r2 ← Enqueue(v, r2)
// Set q1 and q2 to refer to reconstructed queues
q1 ← r1; q2 ← r2

return equal
```

Nivå 1, frågor

- ▶ Uppfyller **Table**: {Empty, Insert, Lookup} nivå 1?

Nivå 1, frågor

- ▶ Uppfyller **Table**: {Empty, Insert, Lookup} nivå 1?
 - ▶ Boken påstår det!

Nivå 1, frågor

- ▶ Uppfyller **Table**: {Empty, Insert, Lookup} nivå 1?
 - ▶ Boken påstår det!
 - ▶ Detta är sant endast om vi gör ett **antagande** angående nyckeltypen, vilket?

Likhet för Tabell

- ▶ Om vi antar att nyckeltypen är **ändlig** och **uppräkningsbar** så uppfyller **Table**: {Empty, Insert, Lookup} nivå 1
- ▶ Hur ser algoritmen för likhetstestet ut?

Likhet för Tabell

- ▶ Om vi antar att nyckeltypen är **ändlig** och **uppräkningsbar** så uppfyller **Table**: {Empty, Insert, Lookup} nivå 1
- ▶ Hur ser algoritmen för likhetstestet ut?

```
▶ Algorithm Table-isequal(A, B: Table(key, val))  
  // Compare two tables A and B. Return True if  
  // they are equal, otherwise False  
  for each possible value x in key type do  
    (b1, v1)  $\leftarrow$  Lookup(x, A)  
    (b2, v2)  $\leftarrow$  Lookup(x, B)  
    if b1 != b2 or Value-isequal(v1, v2) then  
      return False  
  return True
```

- ▶ Denna algoritm är **ineffektiv**
 - ▶ Komplexiteten beror av antalet möjliga värden M för **nyckeltypen**
 - ▶ Ex. för 32-bitars heltal blir $M \approx 4 \cdot 10^9$
 - ▶ Beror ej på antalet element n i tabellen

Nivå 2: Rik gränsyta (*Expressive richness*)

- ▶ Exemplet visar att även om man teoretiskt uppfyller nivå 1 så kan vissa algoritmer bli hopplöst **ineffektiva**
- ▶ Krav för nivå 2: Man ska med hjälp av gränsytan kunna implementera speciella **analysfunktioner** som uppfyller följande:
 - ▶ Objektet ska kunna **rekonstrueras** både till värde och struktur med enbart komposition av analysfunktionerna
 - ▶ Analysfunktionerna får varken innehålla **iteration** eller **rekursion** i sin definition
- ▶ I praktiken räcker det att visa att man kan konstruera en **effektiv kopiering** av datatypen
- ▶ Nivå 2: Nivå 1 + “effektiv kopiering”

Rik gränsyta — exempel

- ▶ **Stack**-specifikationen har en rik gränsyta
 - ▶ `Iempty` kan avgöra om stacken är `Empty` eller konstruerad med `Push(v, s)` för något `v` och `s`
 - ▶ `Top` läser av `v` och `Pop` ger `s`
 - ▶ För vilken stack som helst kan ändliga kompositioner av dessa analysfunktioner
 - ▶ plocka ut vart och ett av elementen i stacken
 - ▶ hitta strukturen, ordningen på dem
 - ▶ Utifrån detta kan stacken återskapas
- ▶ Samma resonemang gäller för **Kö**

Algorithm: Effektiv kopiering av Lista

```
Algorithm List-copy(l: List)
// Returns a copy of the input list.
// The input list is unchanged.

// Output list
c ← List-empty()

// Traverse each element in l
p ← First(l)
// Position in output list to insert in
q ← First(c)

while not Pos-isequal(p, End(l), l) do
  v ← Inspect(p, l)
  // Insert element in output list
  (c, q) ← Insert(v, q, c)
  // Advance output position to insert at the end
  q ← Next(q, c)
  // Advance in input list
  p = Next(p, l)

// Return the copy
return c
```

Algorithm: Effektiv kopiering av Kö

```
Algorithm Queue-copy(q: Queue)
// Returns a copy of the input queue. The input queue
// is modified during the process but is restored
// at return.

// This will become a replica of the original queue
r ← Queue-empty()
// This will be the proper copy of the queue
c ← Queue-empty()

while not Iseempty(q) do
    // Extract and remove first element in the input queue
    v ← Front(q); q ← Dequeue(q)
    // Insert element in both queues
    r ← Enqueue(v, r)
    c ← Enqueue(v, c)

// Restore input queue
q ← r
// Return the copy
return c
```

Algorithm: Effektiv kopiering av Stack

```
Algorithm Stack-copy(s: Stack)
// Returns a copy of the input stack. The input stack
// is deconstructed during the process but is restored
// at return.

// This will become a replica of the original stack
r ← Stack-empty()
// This will be the proper copy of the stack
c ← Stack-empty()
// This is a temporary stack
t ← Stack-empty()

while not Iseempty(s) do
    // Extract and remove top element from the input stack
    v ← Top(s); s ← Pop(s)
    // Push it on temporary stack
    t ← Push(v, t)

// Now the temporary stack is a copy, but reversed
// Unreverse it
while not Iseempty(t) do
    // Extract and remove top element from the temporary stack
    v ← Top(t); t ← Pop(t)
    // Push it on both output stacks
    r ← Push(v, r)
    c ← Push(v, c)

// Restore input stack
s ← r
// Return the copy
return c
```

Algorithm: Effektiv kopiering av Tabell

- Om vi utökar **Tabell** med funktionen **Choose**(t) som returnerar ett godtyckligt nyckel-värde-par så kan vi kopiera effektivt

```
Algorithm Table-copy(t: Table)
// Returns a copy of the input table

// Replica
r ← Table-empty()
// Copy
c ← Table-empty()

while not Iseempty(t) do
    // Extract and remove an arbitrary key-value pair
    (k, v) ← Choose(t)
    t ← Remove(k, t)
    // Insert pair in both tables
    r ← Insert(k, v, r)
    c ← Insert(k, v, c)

// Restore input Table
t ← r
// Return the table copy
return c
```

Praktisk uttrycksfullhet

- ▶ Vi har **teoretiska** mått på uttrycksfullhet:
 - ▶ Nivå 0-2: Objektfullständighet, algoritmfullständighet och rik gränsyta
- ▶ Måste man alltid uppfylla **nivå 2**?
 - ▶ Kan man **nöja sig** med nivå 1 (eller 0)?
- ▶ **Räcker det** med att uppfylla nivå 2?
 - ▶ Ibland **opraktiskt**
 - ▶ Saknar t.ex. utskrifter, längdfunktioner, kopiering
- ▶ Hur skapar man en gränsyta **i praktiken**?

Att utforma en gränsyta till en ADT (1)

- ▶ Utgå från den mentala modellen:
 - ▶ Vilka **data** vill vi kunna lagra i objektet?
 - ▶ Vad vill vi kunna **göra** med objektet?
- ▶ Applicerar de teoretiska begreppen
 - ▶ Vill vi kunna skilja mellan objekt (nivå 1)?
 - ▶ Vill vi kunna kopiera objekt (nivå 2)?

Att utforma en gränsyta till en ADT (2)

- ▶ Ofta blir målet att:
 - ▶ Uppfylla nivå 0 (annars kan vi inte plocka ut allt data vi stoppar in)
 - ▶ Uppfylla nivå 1 (annars finns det algoritmer som inte kan implementeras)
 - ▶ Operationerna är **primitiva**, dvs. inte kan implementeras av övrigare, enklare operationer i gränsytan
 - ▶ Operationerna är **oberoende**, dvs. nivå 1 uppfylls inte om någon operation tas bort
- ▶ Detta ger en **stram yta** med få operationer
- ▶ Om vi får dålig prestanda kan vi alltid senare definiera **extra operationer** utifrån operationerna i grundgränsytan

Specifikation för mängd

- Vilka funktioner behövs för en stram gränsyta?

abstract datatype `Set (val)`

`Empty ()` \rightarrow `Set (val)`

`Single (v: val)` \rightarrow `Set (val)`

`Insert (v: val, s: Set (val))` \rightarrow `Set (val)`

`Union (s: Set (val), t: Set (val))` \rightarrow `Set (val)`

`Intersection (s: Set (val), t: Set (val))` \rightarrow `Set (val)`

`Difference (s: Set (val), t: Set (val))` \rightarrow `Set (val)`

`Isempy (s: Set (val))` \rightarrow `Bool`

`Member-of (v: val, s: Set (val))` \rightarrow `Bool`

`Choose (s: Set (val))` \rightarrow `val`

`Remove (v: val, s: Set (val))` \rightarrow `Set (val)`

`Equal (s: Set (val), t: Set (val))` \rightarrow `Bool`

`Subset (s: Set (val), t: Set (val))` \rightarrow `Bool`

`Kill (s: Set (val))` \rightarrow `()`

Specifikation för mängd

- Vilka funktioner behövs för en stram gränsyta?

abstract datatype `Set(val)`

`Empty()` \rightarrow `Set(val)`

`Isempty(s: Set(val))` \rightarrow `Bool`

`Insert(v: val, s: Set(val))` \rightarrow `Set(val)`

`Member-of(v: val, s: Set(val))` \rightarrow `Bool`

`Choose(s: Set(val))` \rightarrow `val`

`Remove(v: val, s: Set(val))` \rightarrow `Set(val)`

`Kill(s: Set(val))` \rightarrow `()`

Fördelar med en stram gränsyta

- ▶ **Utbytbarhet** (mellan implementationer):
 - ▶ Man kan börja med **enkla implementationer** och sedan byta ut mot allt effektivare
- ▶ **Portabilitet** mellan miljöer:
 - ▶ Mindre problem att flytta en datatyp med **få** operationer
- ▶ **Integritet** mot komplicerande/saboterande tillägg:
 - ▶ Mindre risk för att operationer läggs till som
 - ▶ **strider mot grandidén** med datatypen,
 - ▶ bara fungerar med aktuell **implementation**,
 - ▶ **saboterar** för andra operationer.
 - ▶ *One datatype to rule them all*
 - ▶ *Kan-vara-bra-att-ha-sjukan*
 - ▶ Bättre skapa en **ny datatyp** med tillägget
- ▶ **Säkerhet**
 - ▶ log4j

Programspråksstöd för ADTs

- ▶ Många språk ger mycket litet eller inget stöd alls
- ▶ Då krävs:
 - ▶ Konventioner
 - ▶ Namngivning
 - ▶ Operationsval
 - ▶ God dokumentation av olika val som görs
 - ▶ Disciplin
 - ▶ Inte gå in och peta i interna strukturer!

Log4Shell (CVE-2021-44228)

- ▶ Från beskrivningen:
 - ▶ *An attacker who can control log messages... can execute arbitrary code loaded from LDAP servers...*
- ▶ Påverkade system: Amazon, Cloudflare, iCloud, Minecraft (Java Edition), Steam, Tencent QQ, Twitter
- ▶ Log4j är ett system för att logga händelser i Java
 - ▶ Tillägg för att kunna ladda Java-objekt orsakade sårbarheten

