

## F04 — Riktad lista, Kö, algoritmer, pseudokod

5DV149 Datastrukturer och algoritmer

Kapitel 2.1-2.2, 3.4, 4.1-4.2, 8

Niclas Börlin

[niclas.borlin@cs.umu.se](mailto:niclas.borlin@cs.umu.se)

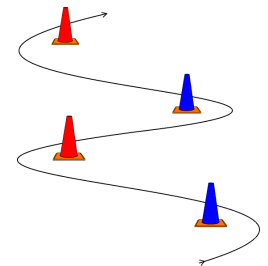
2024-04-05 Fre

- ▶ Riktad lista
- ▶ Kö
- ▶ Algoritmer
- ▶ Pseudokod
- ▶ Läsanvisningar: Kap 2.1-2.2, 3.4, 4.1-4.2, 8

## Riktad lista

## Riktad lista

- ▶ Modell: Slalombana
- ▶ Specialisering av Lista
  - ▶ Förflyttning bara i en riktning — framåt
- ▶ I Lista finns [Previous](#)- och [End](#)-operationer så vi kan navigera åt båda hållen
- ▶ Datatypen Riktad lista har en [Isend](#)-operation i stället.



## Gränsyta till Riktad lista (*Directed List*)

```
abstract datatype DList(val)
auxiliary pos
Empty() → DList(val)
IsEmpty(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)) → ()
```

## Riktad lista vs. Lista

```
abstract datatype List(val)
auxiliary pos
Empty() → List(val)
IsEmpty(l: List(val)) → Bool
First(l: List(val)) → pos
End(l: List(val)) → pos
Next(p: pos, l: List(val)) → pos
Previous(p: pos, l: List(val)) → pos
Pos-isequal(p1, p2: pos, l: List(val)) → Bool
Inspect(p: pos, l: List(val)) → val
Insert(v: val, p: pos, l: List(val))
    → (List(val), pos)
Remove(p: pos, l: List(val)) → (List(val), pos)
Kill(l: List(val)) → ()

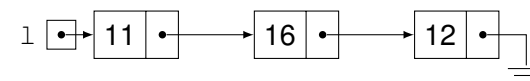
abstract datatype DList(val)
auxiliary pos
Empty() → DList(val)
IsEmpty(l: DList(val)) → Bool
First(l: DList(val)) → pos
Isend(p: pos, l: DList(val)) → Bool
Next(p: pos, l: DList(val)) → pos
Inspect(p: pos, l: DList(val)) → val
Insert(v: val, p: pos, l: DList(val))
    → (DList(val), pos)
Remove(p: pos,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

## Riktad lista, förtydliganden

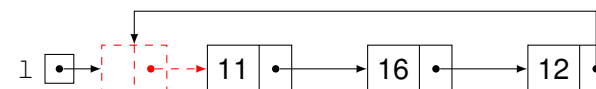
- **Insert** och **Remove** har samma logik som för Lista
  - **Insert**(v, p, l) sätter in värdet v i listan l på positionen **omedelbart före** p och returnerar den nya listan samt positionen för det **nyinsatta** elementet
  - **Remove**(p, l) tar bort elementet på positionen p i listan l och returnerar den nya listan samt positionen **omedelbart efter** det borttagna elementet

## Riktad lista, konstruktionsalternativ

- Fält
- Dubbellänkad lista (ignorera bakåtlänkarna)
- Enkellänkad Lista av 1-Cell utan huvud:



- Använt i flera C-exempel på föreläsningarna.
- Enkellänkad Lista av 1-Cell med 1-Cell-huvud:

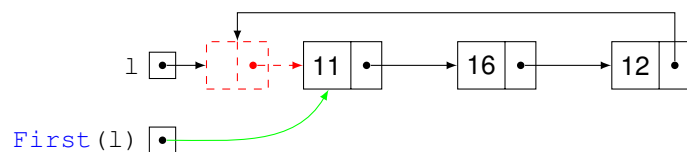


- Används i kodbasen.
- Enkellänkad mer ekonomisk än dubbellänkad

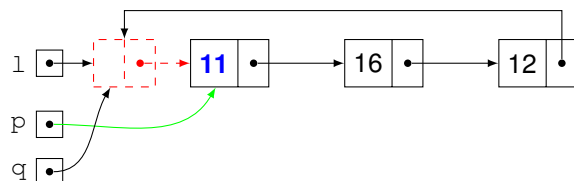
## Problem vid insättning (1)

- Naturligt låta positionen representeras av en pekare till "rätt" element

- `First(l)` returnerar `l->head->next`:



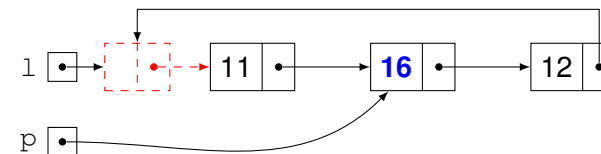
- `Inspect(p, l)` returnerar `p->val`:



- `Isend(q, l)` returnerar `q == l->head`

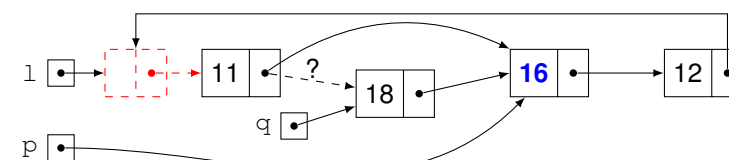
## Problem vid insättning (2)

- Antag att positionen `p` refererar till elementet med värde 16 (`Inspect(p, l)` skulle returnera 16):



- Antag vidare att vi vill sätta in värdet 18 före elementet med värdet 16

- Efter allokering av nytt element, före inlänkning



- Vi behöver komma åt länken i föregående cell!

## Lösning på insättning: Alternativ 1 och 2

- Alternativ 1: Kopiering

- Insättning före elementet på position `p`:

1. Skapa en ny cell
2. Länka in den efter `p`
3. **Kopiera** `p`'s värde till den nya cellen
4. Sätt `p`'s värde till `v`

- Nackdel: Värdet måste kopieras

- Alternativ 2: Sök från början

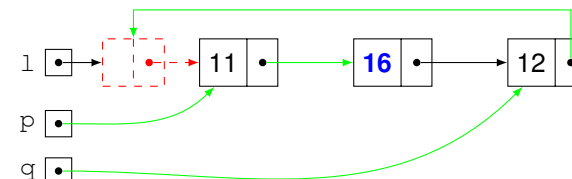
- Traversera från huvudet till elementet före
- Nackdel: Insättning blir **långsam**:  $O(n)$

- Alternativ 3: Byt definitionen av "här"!

## Lösning på insättning: Alternativ 3 (1)

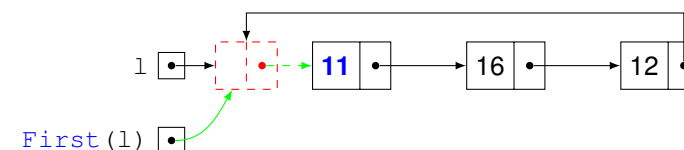
- Representera positionen som en pekare till **föregående** element!

- `Inspect(p, l)` returnerar `p->next->val` i stället för `p->val`:



- `Isend(q, l)` returnerar `q->next == l->head`

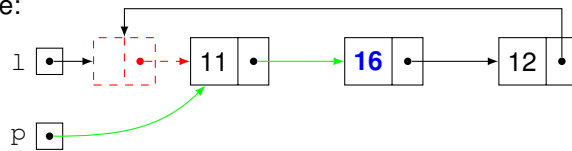
- `First(l)` returnerar `l->head` i stället för `l->head->next`:



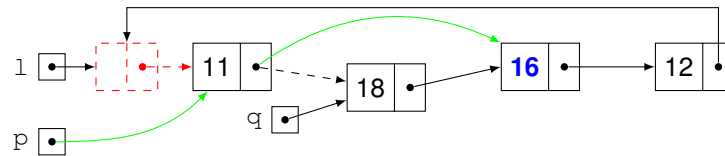
## Lösning på insättning: Alternativ 3 (2)

### ► Insättning:

#### ► Före:



#### ► Efter allokering av nytt element, före inlänkning:

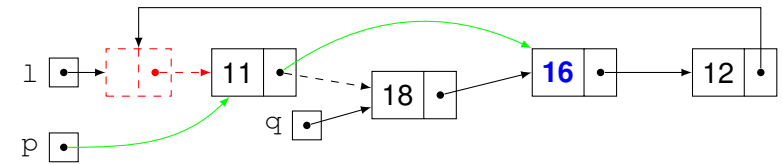


►  $p \rightarrow \text{next} = q$  löser problemet

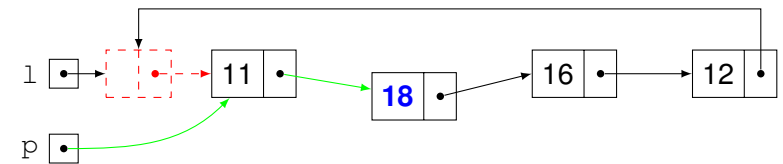
Blank

## Lösning på insättning: Alternativ 3 (3)

### ► Före inlänkning:



### ► Efter inlänkning:



►  $p$  refererar nu till det nyligen insatta elementet, så  $p$  är positionen som ska returneras

Blank

## Kö

## Kö

### ► Modell: Kö.



- Specialisering av Lista
  - Begränsningar på operationerna
- Generisk datatyp (polytyp)
- Sammansatt — lagrar element
  - Homogen
- Linjärt ordnad struktur
- FIFO — *First In, First Out*
  - Insättning i **slutet** av kön
  - Borttagning i **början** av kön

## Gränsyta till Kö

```
abstract datatype Queue(val)
  Empty() → Queue(val)
  Isempty(q: Queue(val)) → Bool
  Enqueue(v: val, q: Queue(val)) → Queue(val)
  Front(q: Queue(val)) → val
  Dequeue(q: Queue(val)) → Queue(val)
  Kill(q: Queue(val)) → ()
```

## Informell specifikation — Kö

- `Empty()` — konstruerar en tom kö
- `Isempty(q)` — testar om kön är tom
- `Enqueue(v, q)` — sätter in ett element med värdet `v` sist i kön `q`
- `Front(q)` — returnerar värdet av det första elementet i kön (förutsatt att kön inte är tom)
- `Dequeue(q)` — returnerar kön med det första elementet borttaget (förutsatt att kön inte är tom)

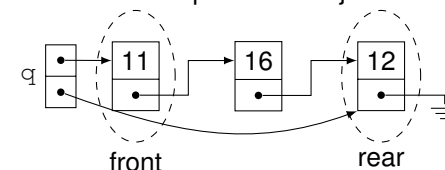
## Kö vs. Stack

```
abstract datatype Queue(val)
  Empty() → Queue(val)
  Isempty(q: Queue(val)) → Bool
  Enqueue(v: val, q: Queue(val)) → Queue(val)
  Front(q: Queue(val)) → val
  Dequeue(q: Queue(val)) → Queue(val)
  Kill(q: Queue(val)) → ()

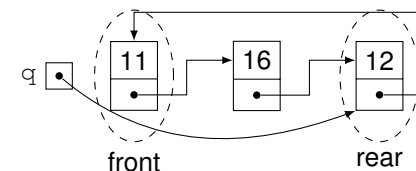
abstract datatype Stack(val)
  Empty() → Stack(val)
  Isempty(s: Stack(val)) → Bool
  Push(v: val, s: Stack(val)) → Stack(val)
  Top(s: Stack(val)) → val
  Pop(s: Stack(val)) → Stack(val)
  Kill(s: Stack(val)) → ()
```

## Kö konstruera som Lista

- ▶ Fronten på kön = början på listan
- ▶ Riktad lista med 1-celler
  - ▶ Lista 1: Rak lista: Fronten på kön = början av listan



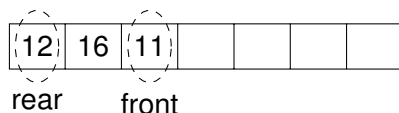
- ▶ Lista 2: Cirkulär lista: Länken i slutet av listan pekar på början av kön



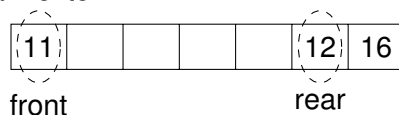
- ▶ Alla kö-operationer går att göra  $O(1)$
- ▶ Kodbasen använder internt Lista

## Kö konstruerad som Fält

- ▶ Fält 1: Rak vektor:



- ▶ Fält 2: Cirkulär vektor:



- ▶ Fördelar:
  - ▶ Fält 2: **Snabb** avläsning, borttagning, insättning
- ▶ Nackdelar:
  - ▶ Maximal storlek
  - ▶ Outnyttjat utrymme
  - ▶ Fält 1: Enqueue är **dyrt** ( $O(n)$ ) pga. omflyttningar
  - ▶ Fält 2, detalj: Måste lämna en position tom för att kunna skilja mellan en **tom** och **full** kö

## Kö, tillämpningar

- ▶ Buffert:
  - ▶ Routrar
  - ▶ Skrivarkö
  - ▶ Telefonkö
- ▶ Bredden-först-traversering

## Vad är en algoritm?

### Algoritmer, psuedokod

- ▶ **Instruktion** som man följer för att lösa ett **givet** problem på ett **strukturerat** sätt (jfr recept eller IKEA-byggbeskrivning)
  - ▶ Indata
  - ▶ Verktyg
  - ▶ Procedur (algoritm)
  - ▶ Resultat (bullar)
- ▶ En *algoritm* är en **stegvis** beskrivning av en **ändlig** process

## Egenskaper för en algoritm

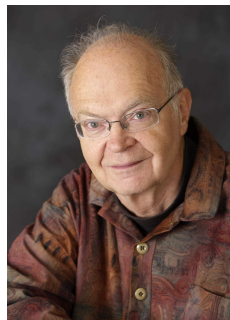
- ▶ **Texten** som beskriver algoritmen har en **fix** (ändlig) storlek
  - ▶ Instruktionen har samma storlek oavsett hur många bullar som bakas
- ▶ **Processen** kan **variera** i storlek
  - ▶ Antalet bullar som bakas kan variera
- ▶ En algoritm kan ha olika **kornighet**
  - ▶ Grovkornig:
    1. Bland mjöl och jäst i en bunke
  - ▶ Finkornig:
    1. Ta fram en bunke som rymmer 5 liter
    2. Ta fram ett decilitermått
    3. Mät upp 5 dl mjöl
    4. Håll mjölet i bunken

## Exempel

- ▶ Antag att vi har en lista på alla anställda på ett företag: Namn, Pnr, lön, etc.
- ▶ Vi vill räkna ut den totala lönekostnaden för företaget
  1. Skriv ner talet 0
  2. För varje anställd i listan
    - 2.1 Addera personens lön till det skrivna talet och skriv ner det nya resultatet
  3. När man nått slutet på listan är det nedskrivna talet den totala lönekostnaden

## Krav på algoritmer

- ▶ Ändlighet
  - ▶ Algoritmen måste sluta (terminera)
- ▶ Bestämndhet
  - ▶ Varje steg måste vara entydigt
- ▶ Indata
  - ▶ Måste ha noll eller flera indata
- ▶ Utdata
  - ▶ Måste ha ett eller flera utdata
- ▶ Effektivitet/genomförbarhet
  - ▶ Varje steg i algoritmen måste gå att utföra på ändlig tid



Donald Knuth (1938 –)  
*The Art of Computer Programming*,  
volumes 1 (1968), 2 (1969), 3 (1973),  
4A (2011)

## Beräkningsbarhet (mer om detta nästa vecka)

- ▶ Reglerna för algoritmer beskriver inte tillräckligt tydligt vad man kan och inte kan göra i form av algoritmer — vad som är **beräkningsbart**
- ▶ Definition av beräkningsbarhet (Alan Turing, 1936)
  - ▶ Ett problem är beräkningsbart om och endast om det finns en *a-machine* (automatic machine)<sup>1</sup> som löser problemet

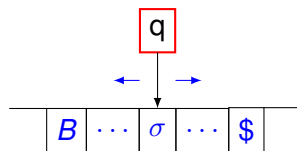


Alan Turing (1912 – 1954)  
Turing, A.M. (1936), "On Computable Numbers, with an Application to the Entscheidungsproblem",  
*Proceedings of the London Mathematical Society*, 2 42: 230-65, 1937.

<sup>1</sup>senare döpt till Turing-maskin

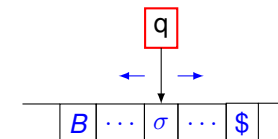
## Turing-maskin (1)

- ▶ En Turing-maskin är en abstrakt, simpel modell av en beräkningsmaskin och består av:
  1. En **centralenhet** som kan befinna sig i ett **ändligt** antal olika **tillstånd** (*states*).
  2. Ett **oändligt** långt **band** indelat i **celler** som kan innehålla **symboler**. Symbolerna kommer från ett **ändligt** alfabet.
  3. Ett **läs/skriv-huvud**.
  4. En **drivanordning** för bandet.



## Turing-maskin (2)

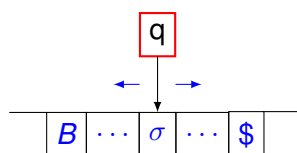
- ▶ En Turing-maskin kan utföra 4 operationer:
  1. **Avläsa** symbolen i cellen som är mitt för läs-/ skriv-huvudet (den **aktuella** cellen).
  2. (Radera och) **skriva en ny symbol** i den aktuella cellen.
  3. **Flytta bandet** en cell-längd framåt eller bakåt.
  4. **Stanna maskinen**.
- ▶ Turing-maskinen jobbar i diskreta beräkningssteg.
  - ▶ I varje steg utförs en operation.
  - ▶ Vilken operation Turing-maskinen utför bestäms av en övergångsfunktion (*transition function* eller *action table*).





## Turing-maskin (3)

- Maskinen startar med ett **starttillstånd** och en **given position** på remsan.
- Maskinen avslutar beräkningen när ett **sluttillstånd** uppnås
  - Beräkningen kan misslyckas genom att maskinen aldrig uppnår sluttillståndet.
- Trots sin enkelhet så kan **varje dator-algoritm översättas till en Turing-maskin**.



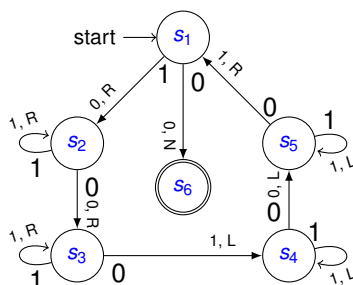
## Turing-maskin, exempel (1)

- <https://sv.wikipedia.org/wiki/Turingmaskin>
- Tillstånd: s1, s2, s3, s4, s5, s6.
- Alfabet: 0, 1.
- Tomma symbolen: 0 (får finnas oändligt många gånger på remsan).
- Indatasymboler: 1.
- Starttillstånd: s1.
- Sluttillstånd: s6.
- Maskinen startar med läshuvudet vid den första ettan (den längst till vänster).

S	R	W	N	M	Kommentar
s1	0	0	s6	N	Ingen (mer) etta att kopiera. Klar!
s1	1	0	s2	R	Kopiera denna etta till näst-nästa nolla högerut, lämna en nolla som marker
s2	1	1	s2	R	Leta vidare högerut efter första nollan.
s2	0	0	s3	R	Första nollan hittad. Gå vidare till s3, som hittar andra.
s3	1	1	s3	R	Leta vidare högerut efter andra nollan.
s3	0	1	s4	L	Andra nollan hittad, skriv en etta och gå tillbaka två nollor.
s4	1	1	s4	L	Leta vidare vänsterut efter första nollan på tillbakavägen.
s4	0	0	s5	L	Första nollan hittad. Gå vidare till s5 som hittar andra.
s5	1	1	s5	L	Leta vidare vänsterut efter andra nollan (den som s1 lämnade som markeri
s5	0	1	s1	R	Nollan hittad. Skriv tillbaka den etta som s1 skrev över, och börja sedan om

## Turing-maskin, exempel (2)

S	R	W	N	M	Kommentar
s1	0	0	s6	N	Ingen (mer) etta att kopiera. Klar!
s1	1	0	s2	R	Kopiera denna etta till näst-nästa nolla högerut, lämna en nolla som markering för til
s2	1	1	s2	R	Leta vidare högerut efter första nollan.
s2	0	0	s3	R	Första nollan hittad. Gå vidare till s3, som hittar andra.
s3	1	1	s3	R	Leta vidare högerut efter andra nollan.
s3	0	1	s4	L	Andra nollan hittad, skriv en etta och gå tillbaka två nollor.
s4	1	1	s4	L	Leta vidare vänsterut efter första nollan på tillbakavägen.
s4	0	0	s5	L	Första nollan hittad. Gå vidare till s5 som hittar andra.
s5	1	1	s5	L	Leta vidare vänsterut efter andra nollan (den som s1 lämnade som markering).
s5	0	1	s1	R	Nollan hittad. Skriv tillbaka den etta som s1 skrev över, och börja sedan om från bör



## Turing-maskin, exempel (3)

Steg	Tillstånd	Remsa
1	s1	<b>1</b> 1000
2	s2	0 <b>1</b> 000
3	s2	01 <b>0</b> 00
4	s3	010 <b>0</b> 0
5	s4	0101 <b>0</b>
6	s5	01010 <b>0</b>
7	s5	<b>0</b> 1010
8	s1	<b>1</b> 1010
9	s2	10 <b>0</b> 10
10	s3	100 <b>1</b> 0
11	s3	1001 <b>0</b>
12	s4	1001 <b>1</b>
13	s4	100 <b>1</b> 1
14	s5	1001 <b>1</b>
15	s1	<b>1</b> 1011
16	s6	-stopp-

## Hur beskriver vi algoritmer?

### Pseudokod

- ▶ Vi använder oss av pseudokod för att beskriva algoritmer
  - ▶ Vi behöver ett språk som:
    - ▶ Kan förstås av en människa.
    - ▶ Är strukturerat och formellt
    - ▶ Mindre formellt än programmeringsspråk
  - ▶ Mix av naturligt språk och programmeringsspråk
  - ▶ Influenser från matematisk notation
  - ▶ Det finns inget universellt språk utan många dialekter
- ▶ Pseudokod döljer mycket av programspråkens designval, dvs. pseudokoden är **oberoende** av programspråk
- ▶ Man ska kunna fokusera på hur man **löser problemet** och inte hur lösningen ska **implementeras**

### Pseudokod

- ▶  $\leftarrow$  används för tilldelning.
- ▶  $=$  används för likhetsrelation.
- ▶ Funktionsdeklarationer:
  - ▶ **Algorithm** name(param1, param2)
- ▶ Beslutsstrukturer:
  - ▶ if ... then ... [else ...]
- ▶ Anrop:
  - ▶ algorithmName ( args )
- ▶ Villkorsloopar:
  - ▶ while ... do ...
  - ▶ repeat ... until ...
- ▶ Räkneloopar:
  - ▶ for ... do ...
- ▶ Fältindexering:
  - ▶ A[i]
- ▶ Returnera värden:
  - ▶ return value

### Pseudokod, Exempel

```
Algorithm Array-mean(v: Array, n: Int)
// Input: An Array v storing n integers
// Output: The average of the n elements in v
sum  $\leftarrow$  v[0]
for i  $\leftarrow$  1 to n-1 do
    sum  $\leftarrow$  sum + v[i]
return sum/n
```

## Implementationer av pseudokoden

```
Algorithm Array-mean(v: Array, n: Int)
// Input: An Array v storing n integers
// Output: The average of the n elements in v
sum ← v[0]
for i ← 1 to n-1 do
    sum ← sum + v[i]
return sum/n
```

C

```
double arrayMean(double *v, int n)
{
    double sum = v[0];
    for (int i=1; i < n; i++)
        sum += v[i];
    return sum/n;
}
```

Python

```
def arrayMean(v,n):
    sum = v[0]
    for i in range(1,n):
        sum = sum + v[i]
    return sum/n
```

## Exempel

- ▶ Skriv en pseudokod för att räkna ut längden på en lista
  - ▶ Använd operationerna för Lista:s ADT

```
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
```

## Kopiering av sammansatt datatyp

- ▶ OBS! Kopiering av en **sammansatt** datatyp, t.ex. Lista, Fält, Kö, etc., är **inte** definierat!
  - ▶ Undantaget är om kopieringsoperationen är **explicit** definierad i datatypens gränssnitt
  - ▶ För en kö  $q_2$ , om tilldelningen  $q_1 \leftarrow q_2$  är definierad så brukar det betyda att  $q_1$  och  $q_2$  refererar till **samma** kö!
- ▶ Vill man kopiera en sammansatt datatyp måste man själv skriva koden med hjälp av datatypens gränssnitt
- ▶ För exempel, se övningarna i gruppövning 1

## Algoritm-mönster för lista

- ▶ Traversering
  - ▶ Besök systematiskt alla element i objektet
- ▶ Sökning
  - ▶ Sök efter det första elementet som uppfyller ett bestämt villkor
- ▶ Filtrering
  - ▶ Filtrera ut (behåll) alla element som uppfyller ett bestämt villkor
- ▶ Reduktion
  - ▶ Beräkna en funktion av objektets elementvärden
    - ▶ Ex. summera alla tal i en lista
- ▶ Avbildning (mappning)
  - ▶ Transformera varje elementvärde i en datastruktur
    - ▶ Ex. multiplicera alla talen i en lista med 4

## Traversering

### ► Besök systematiskt alla element i objektet

```
Algorithm List-traverse(l: Directed list,
                      Do-stuff: Function)
// Input: A directed list and a function to apply
// to each element in the list. The function
// should accept element values. The input list
// might be modified.
// Output: None.

// Start at the beginning of the list
p ← First(l)
while not Isend(p, l) do
    // Apply function to each element
    Do-stuff(Inspect(p, l))
    // Advance to next element
    p ← Next(p, l)
```

## Sökning

### ► Sök efter ett element som uppfyller ett bestämt villkor

```
Algorithm List-peek(l: Directed list,
                  Match: Function)
// Input: A list and a match function. The function
// should return True if the element value
// matches what the caller wants.
// Output: True/False + pos for the first matching
// element, if any.

// Start at the beginning of the list
p ← First(l)
while not Isend(p, l) do
    // Does the element value match what we want?
    if Match(Inspect(p, l)) then
        // Yes, return True and the position
        return (True, p)
    else
        // No, advance in the list
        p ← Next(p, l)
// No match found, return False
return (False, None)
```

## Filtrering

### ► Filtrera ut alla element som uppfyller ett bestämt villkor

```
Algorithm List-filter(l: Directed list, Match: Function)
// Input: A list and a match function. The function
// should return True if the element value
// matches what the caller wants.
// Output: A list with all matching elements.
// NOTE: The output list is in the reverse order.

// Start with an empty output list
matched ← Empty()
p ← First(l)
while not Isend(p, l) do
    // Does the element value match?
    val ← Inspect(p, l)
    if Match(val) then
        // Yes, insert it first in the output list
        matched ← Insert(val, First(matched), matched)
    // Advance in the input list
    p ← Next(p, l)
// Return the list of matched element values
return matched
```

## Avbildning (mappning)

### ► Transformera varje elementvärde i en datastruktur

```
Algorithm List-map(l: Directed list, Map: Function)
// Input: A list and a mapping function. The function
// should accept an element value and return a
// transformed value.
// Output: A list with transformed elements in
// the same order as in the input list.

// Start with an empty output list
// Keep track of last position in m, i.e. position
// where to insert elements
m ← Empty()
q ← First(m)

p ← First(l)
while not Isend(p, l) do
    v ← Map(Inspect(p, l))
    (m, q) ← Insert(v, q, m)
    q ← Next(q, m)
    p ← Next(p, l)
return m
```

## ► Beräkna en funktion av objektets elementvärden

```

Algorithm List-reduce(l: Directed list,
                    Combine: Function)
// Input: A list and a function. The function
// should accept two element values and return
// a combination of the values
// Output: The result of the function applied to
// all elements in the list
// NOTE: The function should be commutative

p ← First(l)
res ← Inspect(p, l)
while not Isend(p, l) do
    res ← Combine(res, Inspect(p, l))
    p ← Next(p, l)
return res
    
```

## ► Skriv en algoritm som delar upp en lista så att udda tal hamnar i en ny lista och jämna tal i en annan

## Algoritm 1

```

Algorithm List-split1(l: Directed list)
// Input: A list holding integers
// Output: Two lists holding the odd and even numbers
// of the input list, respectively, in unchanged
// order
odd ← List-empty()
ipo ← First(odd) // Insert position in odd list
even ← List-empty()
ipe ← First(even) // Insert position in even list
p ← First(l) // Position in input list
while not Isend(p, l) do
    v ← Inspect(p, l)
    if Is-odd(v) then
        // Insert and advance in odd list
        (odd, ipo) ← Insert(v, ipo, odd)
        ipo ← Next(ipo, odd)
    else
        // Insert and advance in even list
        (even, ipe) ← Insert(v, ipe, even)
        ipe ← Next(ipe, even)
    // Advance in input list
    p ← Next(p, l)
return (odd, even)
    
```

## Algoritm 2

```

Algorithm List-split2(l: Directed list)
// Input: A list holding integers
// Output: One lists holding the odd numbers and
// the original list, modified to hold
// the even numbers only
odd ← List-empty()
ipo ← First(odd) // Insert position in odd list
p ← First(l) // Position in input list
while not Isend(p, l) do
    v ← Inspect(p, l)
    if Is-odd(v) then
        // Insert and advance in odd list
        (odd, ipo) ← Insert(v, ipo, odd)
        ipo ← Next(ipo, odd)
        // Remove from input list
        // Returned p will refer to next unprocessed element
        (l, p) ← Remove(p, l)
    else
        // Advance to next unprocessed element in input list
        p ← Next(p, l)
return (odd, l)
    
```