Forelesning 2

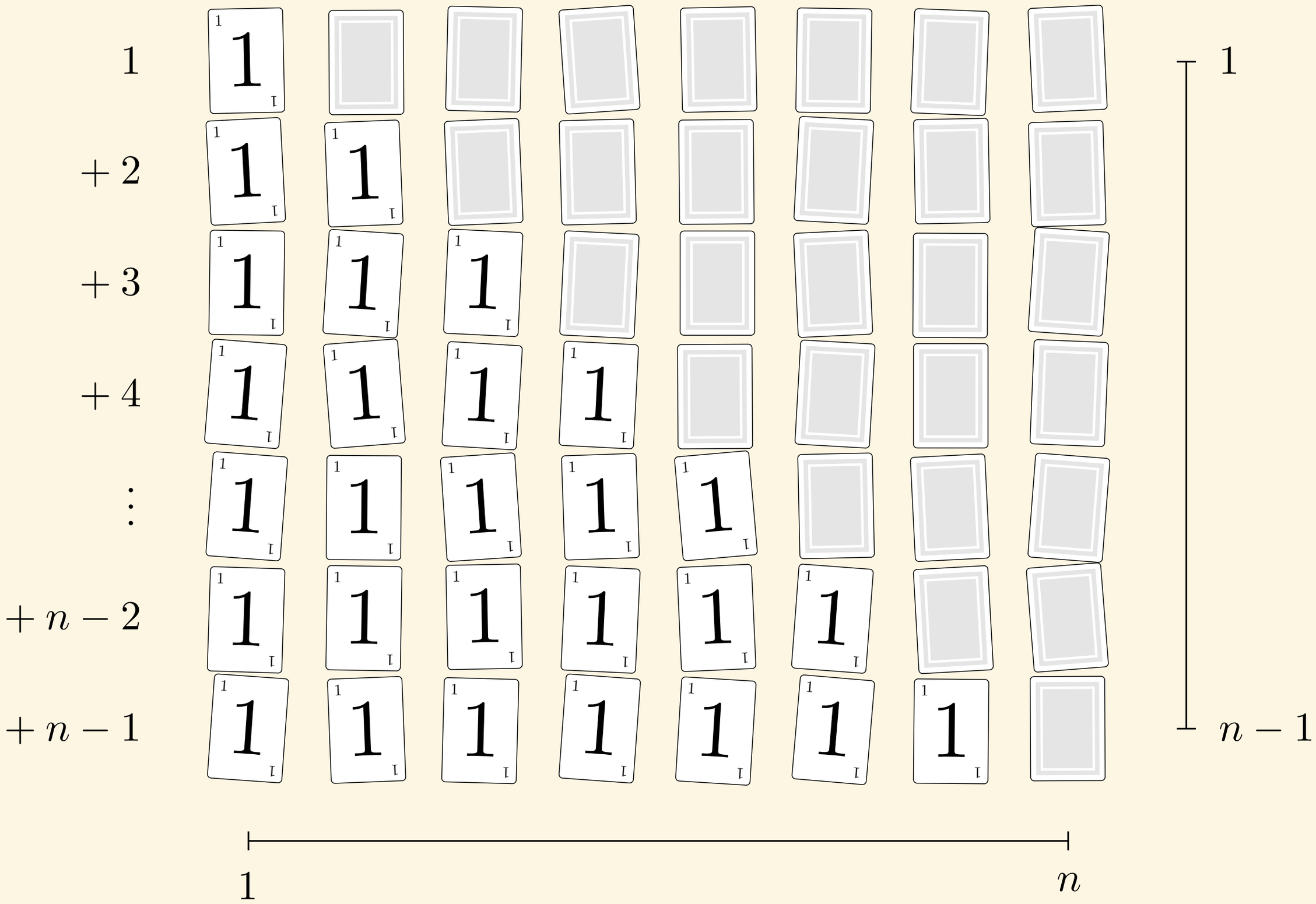
Datastrukturer

0:4

To kritiske summer

$$1+2+3+4+5$$

mlh@ntnu.no



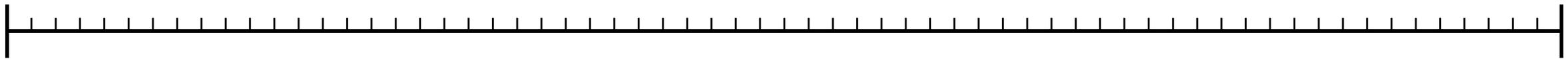
$$\sum_{i=0}^{n-1} i = \frac{n \cdot (n-1)}{2}$$

antall $\times$ gjennomsnitt av første og siste

mlh@ntnu.no

$$1+2+4+8+16$$

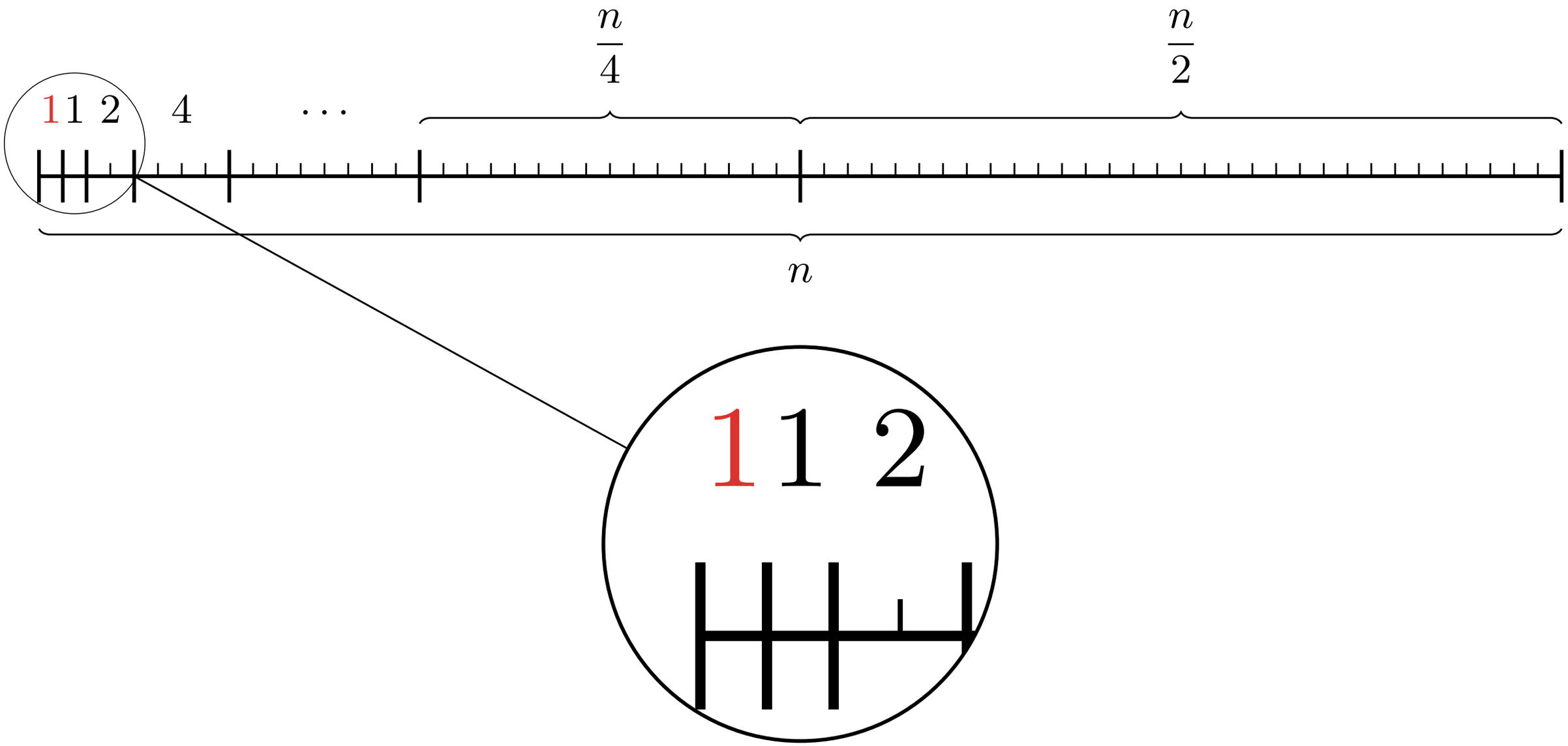
Ett av Xeno sine berømte paradokser: Du kan aldri komme deg fra A til B, fordi først må du komme halvveis, og før det, så må du komme halvveis til halveien, etc.



Hvis vi bruker reelle tall og ser på grenseverdien av n/2 + n/4 + n/8 + … så blir jo svaret bare n. Men hva om vi har begrenset oppløsning, og bare kan bruke heltall?

Vi kan anta at n = 2^h for et eller annet ikke-negativt heltall h.

mlh@ntnu.no



Alternativ huskeregel 1: Utslagsturnering med n deltakere.

I runde 1 trenger du n/2 matcher. I runde 2 trenger du n/4, etc., helt til du til slutt har én match mellom de to siste. Antall matcher er 1 + 2 + 4 + … + n/2.

Og hvor mange matcher er det? I hver match går én deltaker ut av turneringen, helt til du sitter igjen med bare vinneren, som aldri blir slått ut. Altså n – 1 matcher.

Med andre ord: 1 + 2 + 4 + … + n/2 = n–1.

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Alternativ huskeregel 2: Tenk på et tall i totallssystemet som bare består av ettall. Dvs., 11111111…111. Hvilken verdi har dette tallet?

Vel, hvis det har h siffer, er det summen av de h første toerpotensene, 1 + 2 + 4 + … + 2^(h-1).

Men hva skjer om du legger til 1? Da får du plutselig 100000000…000 – ett siffer ekstra, og tallet er lik neste toerpotens. Så 11111111…111 er én mindre enn neste toerpotens.

Med andre ord: 1 + 2 + 5 + … + 2^(h-1) = 2^h – 1.

$$1+2+4+\cdots+\frac{n}{4}+\frac{n}{2}=n-1$$

1. Stakker og køer

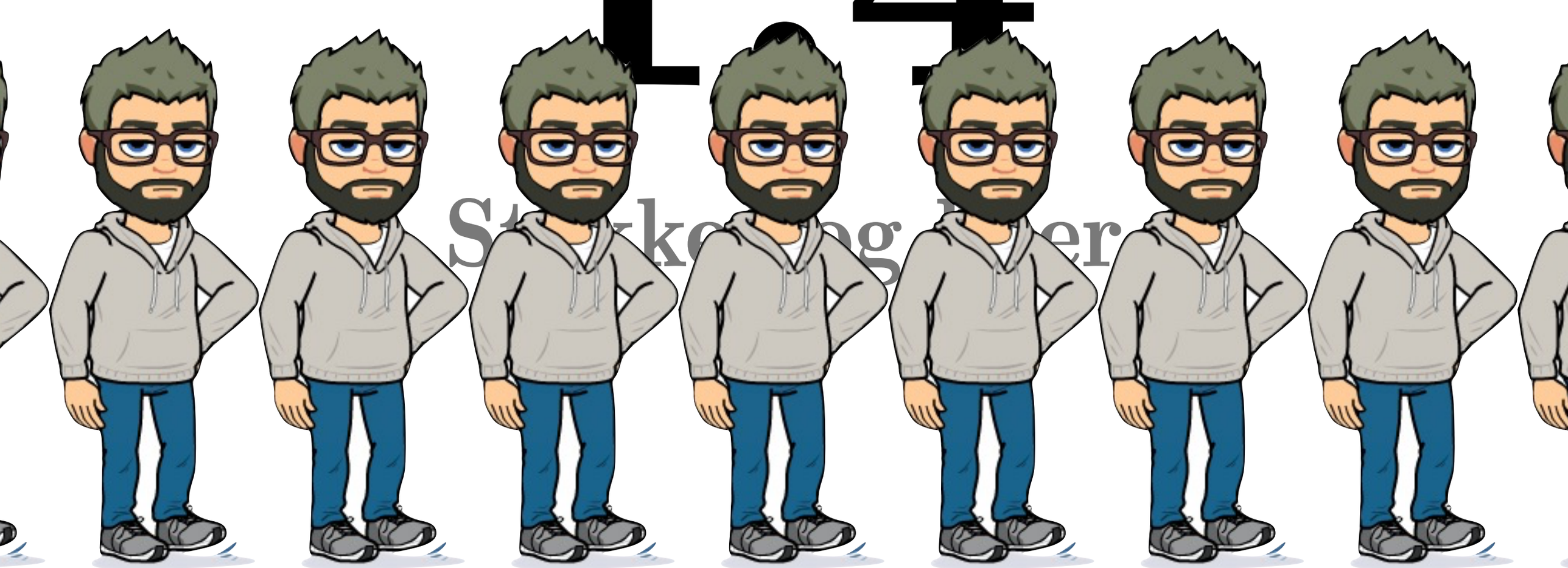
2. Lenkede lister

3. Hashtabeller

4. Dynamiske tabeller

10.3: Selvstudium

1:4



Stakker

Kun adgang øverst

Stakk vs stack. Stakk som i «høystakk» – mer passende navn hadde kanskje vært «stabel».

Stack-Empty(S)

Er stakken tom?

```
STACK-EMPTY(S)
1  if S.top == 0
```

top er øverste indeks, dvs. antall elementer

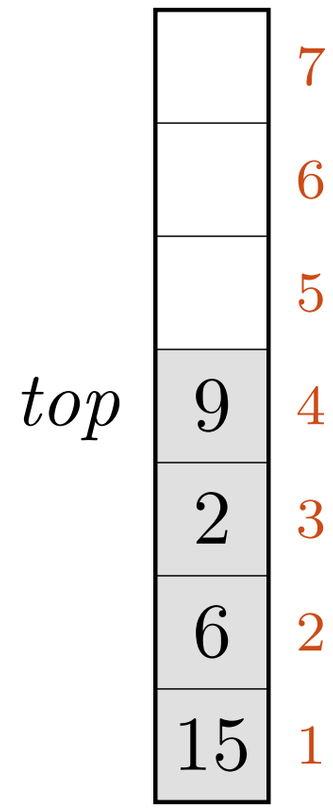
```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
```

Ingen elementer, så stakken er tom

```
STACK-EMPTY(S)
1  if S.top == 0
2        return TRUE
3  else return FALSE
```

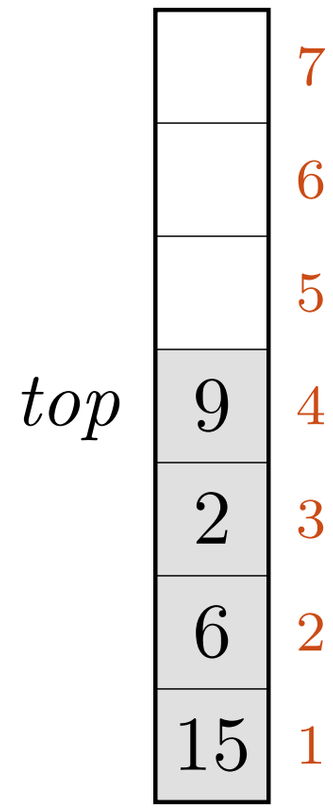
Vi har elementer, så stakken er ikke tom

```
STACK-EMPTY(S)
1  if S.top == 0
2        return TRUE
3  else return FALSE
```



```
STACK-EMPTY(S)
1  if S.top == 0
2        return TRUE
3  else return FALSE
```

→ FALSE



Push(S, x)

Legg et element på toppen

```
PUSH(S, x)
1  S.top = S.top + 1
```

Øk antallet med én

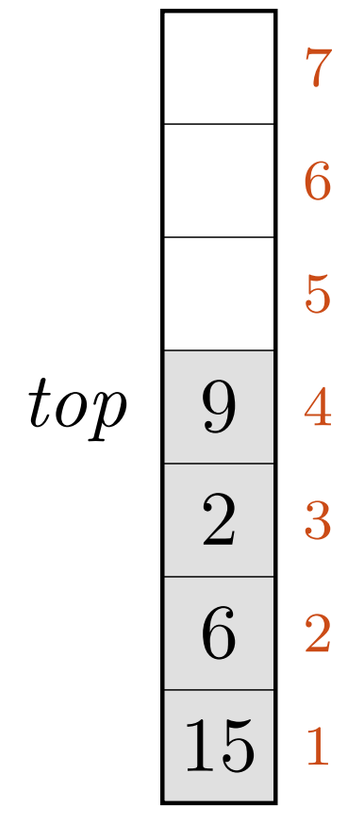
$$
\begin{array}{l}
\textsc{Push}(\mathrm{S}, x) \\
1 \quad \mathrm{S}.top = \mathrm{S}.top + 1 \\
2 \quad \mathrm{S}[\mathrm{S}.top] = x
\end{array}
$$

Øverste indeks refererer nå til x

PUSH(S, x)

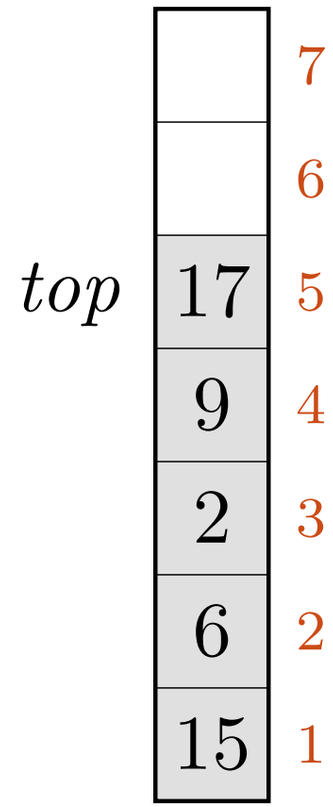
```
1  S.top = S.top + 1
2  S[S.top] = x
```

$x = 17$



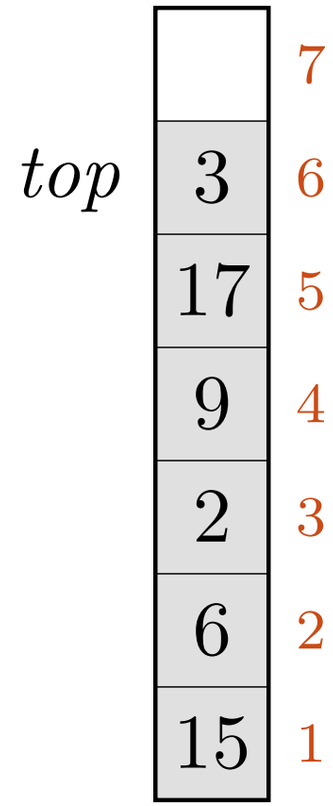
```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

$x = 17$



```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

$x = 3$



Pop(S)

Fjern ett element fra toppen

mlh@ntnu.no

```
POP(S)
1  if STACK-EMPTY(S)
```

mlh@ntnu.no

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
```

Kan ikke fjerne noe fra en tom stakk

mlh@ntnu.no

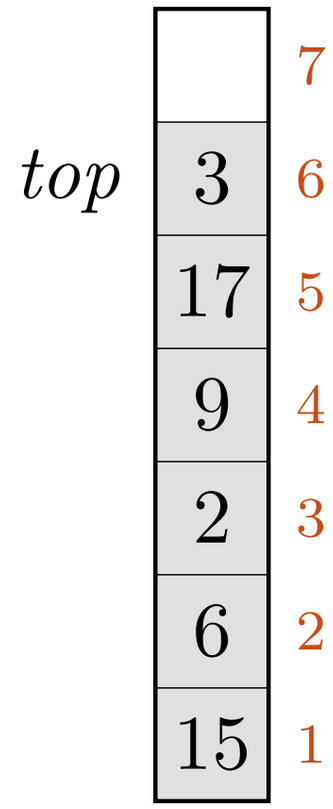
```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
```

Redusér antallet med én

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

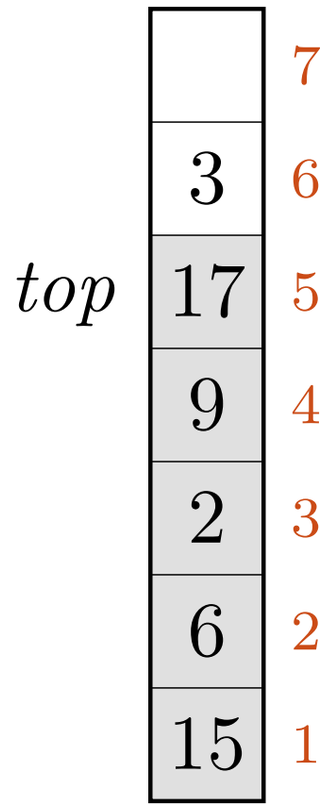
Returnér den gamle toppen (nå utenfor stakken)

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```



```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

→ 3



Køer

Først inn, først ut

Kan implementeres på flere måter (inkl. lenkede lister, f.eks.); her bruker vi et såkalt «ringbuffer».

$\textsc{Enqueue}(\mathrm{Q}, x)$

Legg til et element bakerst i køen

$$\begin{array}{l}\textsc{Enqueue}(\mathrm{Q}, x)\\ 1 \quad \mathrm{Q}[\mathrm{Q}.tail] = x\end{array}$$

Q[1] er først i køen og Q[$tail$] er den ledige plassen bak

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
```

Vi må forskyve *tail*, men har kommet til slutten!

$\textsc{Enqueue}(\mathrm{Q}, x)$
1 $\mathrm{Q}[\mathrm{Q}.tail] = x$
2 **if** $\mathrm{Q}.tail == \mathrm{Q}.length$
3 $\quad\quad \mathrm{Q}.tail = 1$

Tabellen «går i ring», så neste posisjon er 1

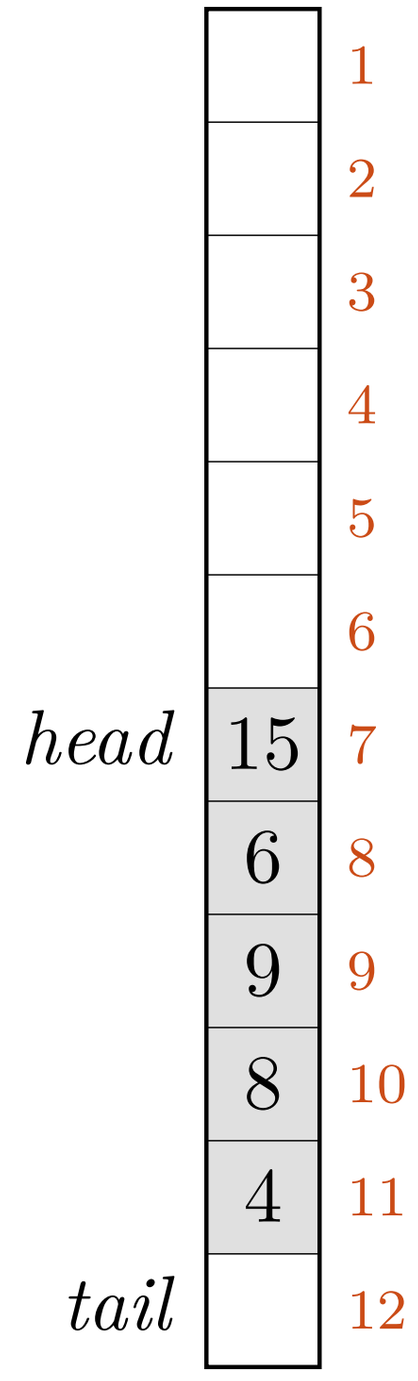
```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

Har vi ledig plass, er det jo ikke noe problem

ENQUEUE(Q, x)

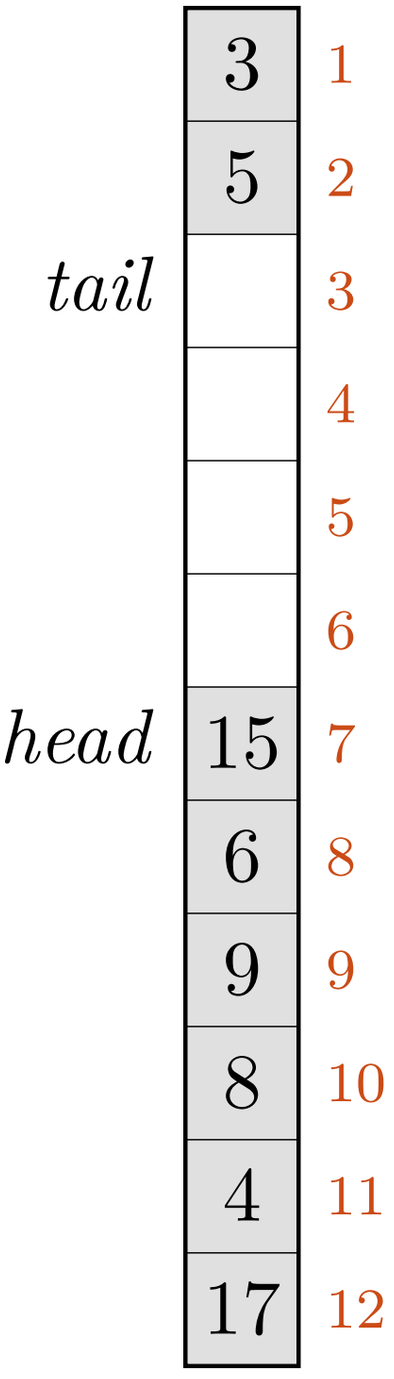
```
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

$x = 17$



ENQUEUE(Q, x)
1 Q[Q.$tail$] = x
2 **if** Q.$tail$ == Q.$length$
3 Q.$tail$ = 1
4 **else** Q.$tail$ = Q.$tail$ + 1

$x = 5$



Dequeue(Q)

Den første i køen går ut

mlh@ntnu.no

```
DEQUEUE(Q)
1  x = Q[Q.head]
```

x er den som står først i køen

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
```

Vi må forskyve *head*, men har kommet til slutten!

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3      Q.head = 1
```

Tabellen «går i ring», så neste posisjon er 1

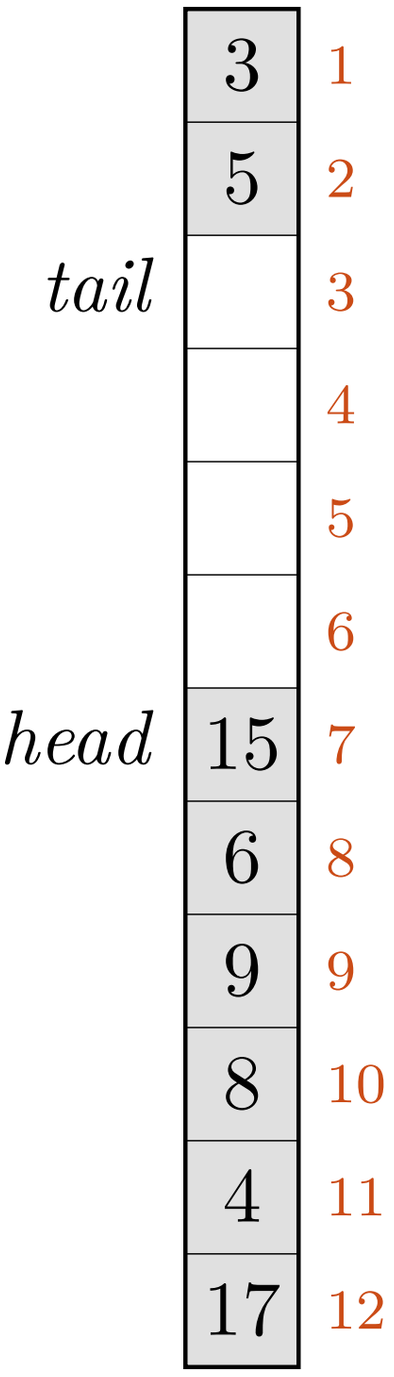
```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
```

Har vi ledig plass, er det jo ikke noe problem

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

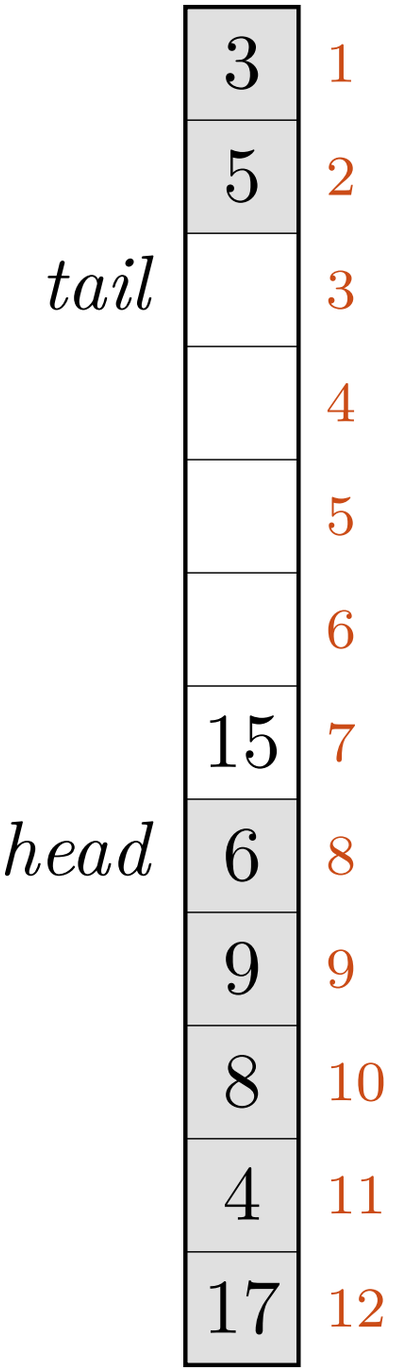
x = –



```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

→ 15

$x = 15$



2.4

Lenkede lister

- Består av «noder», som peker på neste (og kanskje forrige)
- Tar lineær tid å slå opp på en gitt posisjon
- Tar konstant tid å sette inn/slette elementer

LIST-SEARCH(L, k)

L liste
k nøkkelverdi

Finn en node med nøkkel k

```
LIST-SEARCH(L, k)
1  x = L.head
```

L liste
k nøkkelverdi
x nodepeker

Vi begynner med den første noden

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
```

L liste
k nøkkelverdi
x nodepeker

Har vi nådd slutten eller funnet k?

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
```

L liste
k nøkkelverdi
x nodepeker

Hvis ikke, så går vi til neste node

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3        x = x.next
4  return x
```

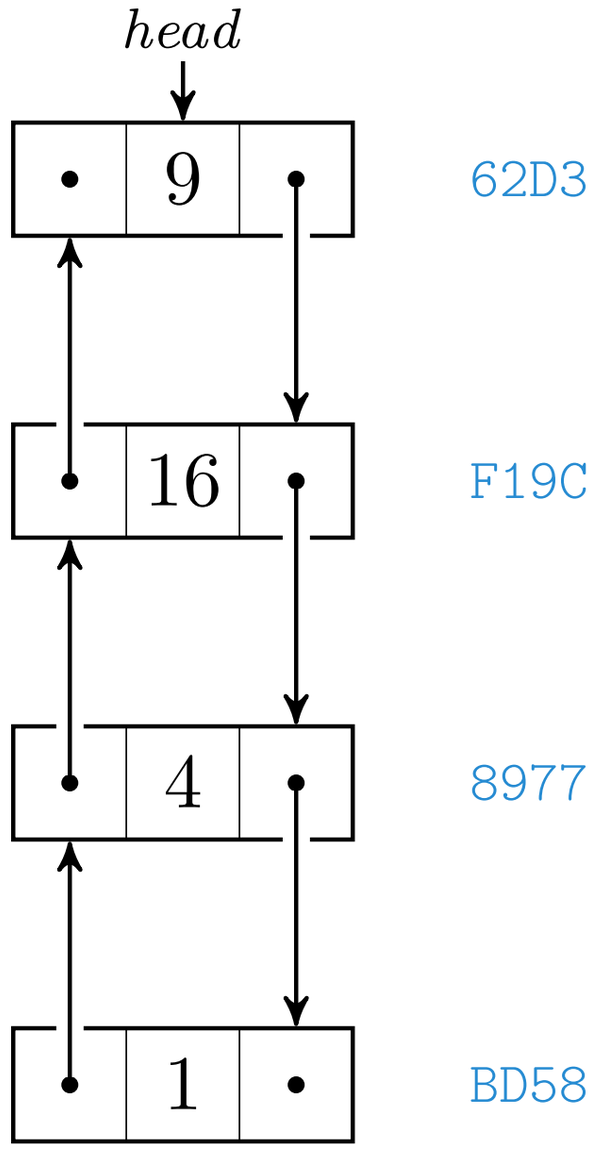
L liste
k nøkkelverdi
x nodepeker

Her er enten x == NIL eller $x.key == k$

LIST-SEARCH(L, k)

```
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

$k, x \;=\; 4, -$

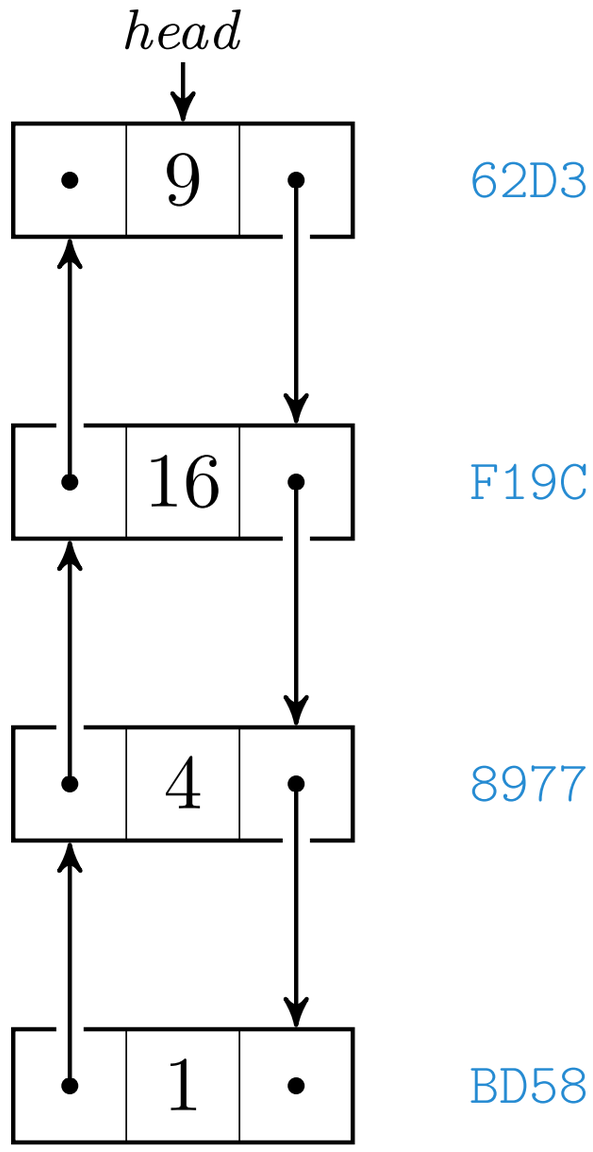


De blå tallene ved siden av nodene er ment å være minne-adresser. En peker er egentlig bare en tallvariabel som inneholder en slik adresse.

LIST-SEARCH(L, k)
1 $x = \text{L}.head$
2 **while** $x \neq$ NIL and $x.key \neq k$
3 $\quad x = x.next$
4 **return** x

→ NIL

$k, x = 7, \text{NIL}$



LIST-INSERT(L, x)

L liste
x peker, ny node

Sett inn noden x som nytt hode i lista L

LIST-INSERT(L, x)
1 $x.next = \mathrm{L}.head$

L liste
x peker, ny node

Forrige hode kommer nå etter x

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
```

L liste
x peker, ny node

(Med mindre lista var tom . . .)

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
```

L liste
x peker, ny node

Det forrige hodet har nå x som forgjenger

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
```

L liste
x peker, ny node

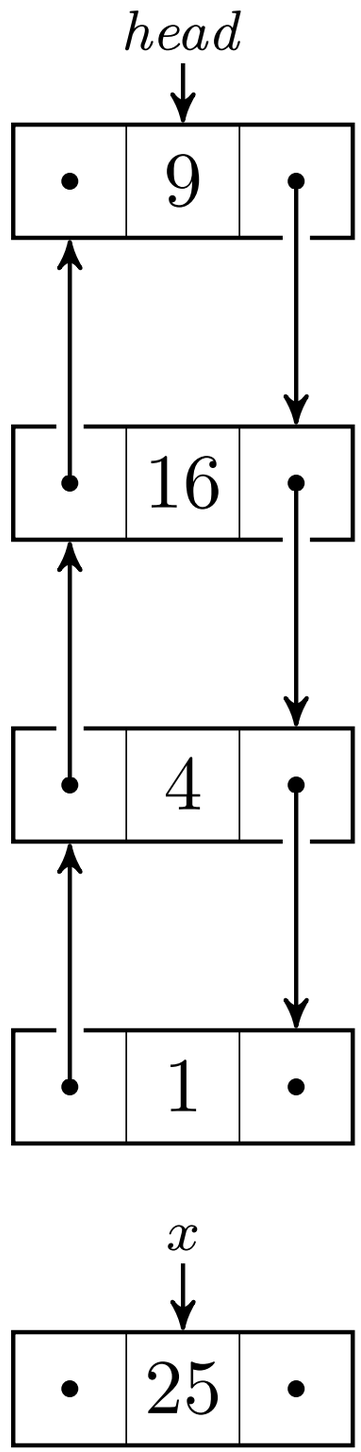
«Kryssreferanser» er på plass: Sett inn noden!

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

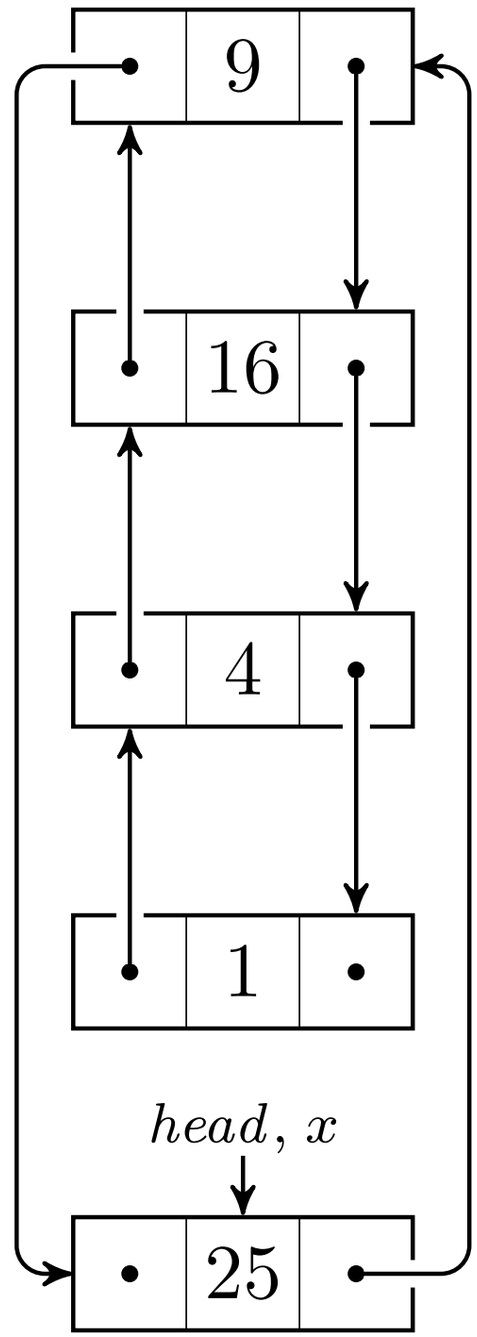
L liste
x peker, ny node

(. . . og husk at x ikke har noen forgjenger)

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3        L.head.prev = x
4  L.head = x
5  x.prev = NIL
```



```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3        L.head.prev = x
4  L.head = x
5  x.prev = NIL
```



List-Delete(L, x)

L liste
x skal fjernes

Få nodene før og etter til å referere til hverandre, og «hoppe over» x, som fortsatt peker på disse nodene, men som ikke *blir pekt på* av noen noder i lista lenger.

Fjern noden x fra lista L

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
```

L liste
x skal fjernes

(Med mindre x står først . . .)

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
```

L liste
x skal fjernes

Forgjengeren arver x sin etterkommer

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
```

L liste
x skal fjernes

Hvis x sto først, står etterkommeren nå først

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
```

L liste
x skal fjernes

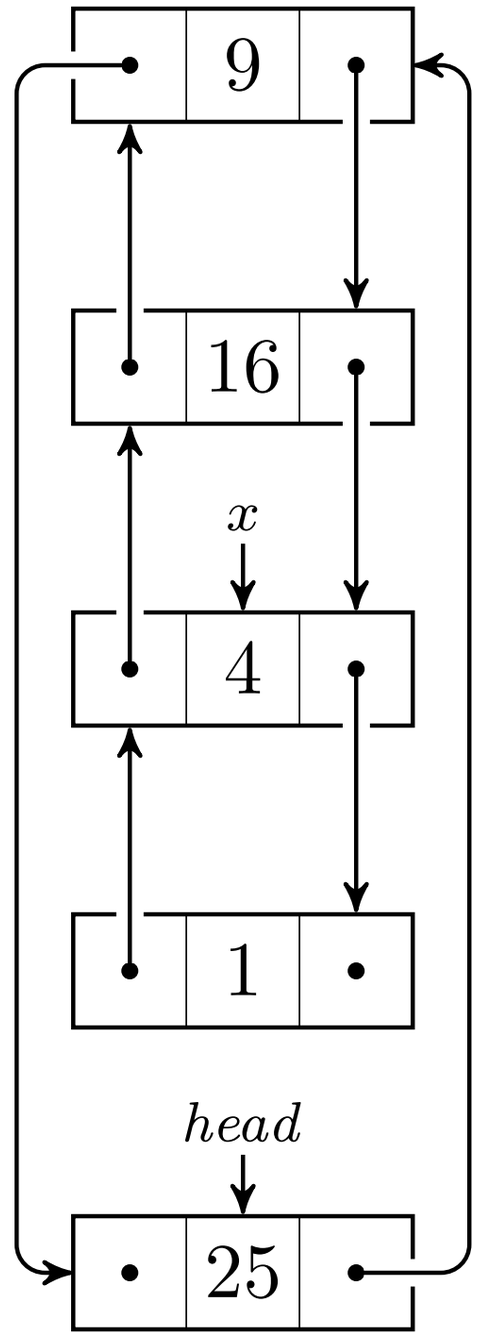
(Med mindre x står sist . . .)

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

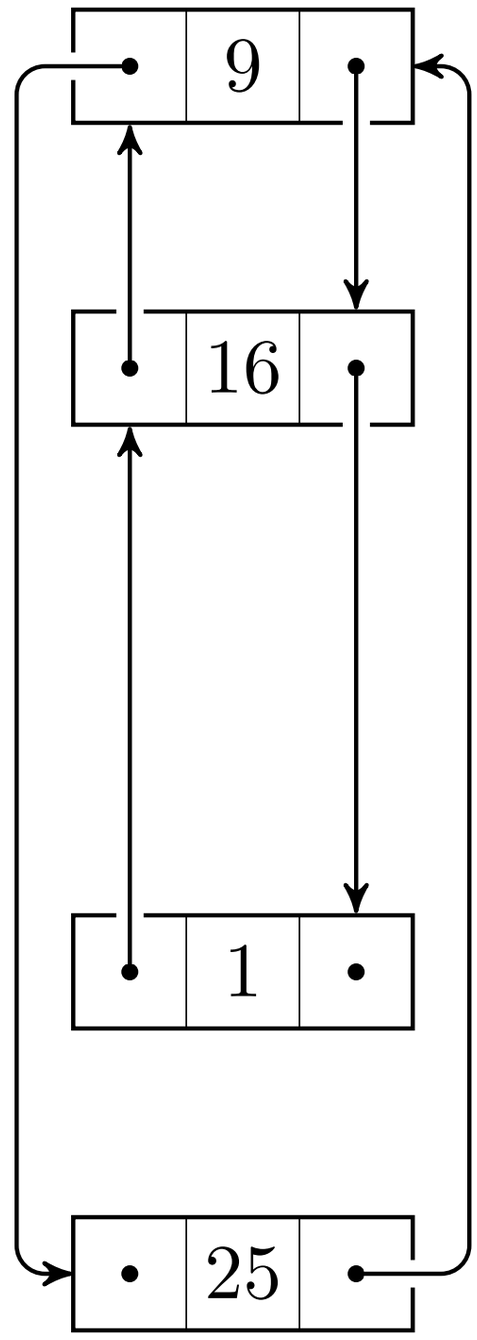
L liste
x skal fjernes

Etterkommeren arver x sin forgjenger

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```



```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```



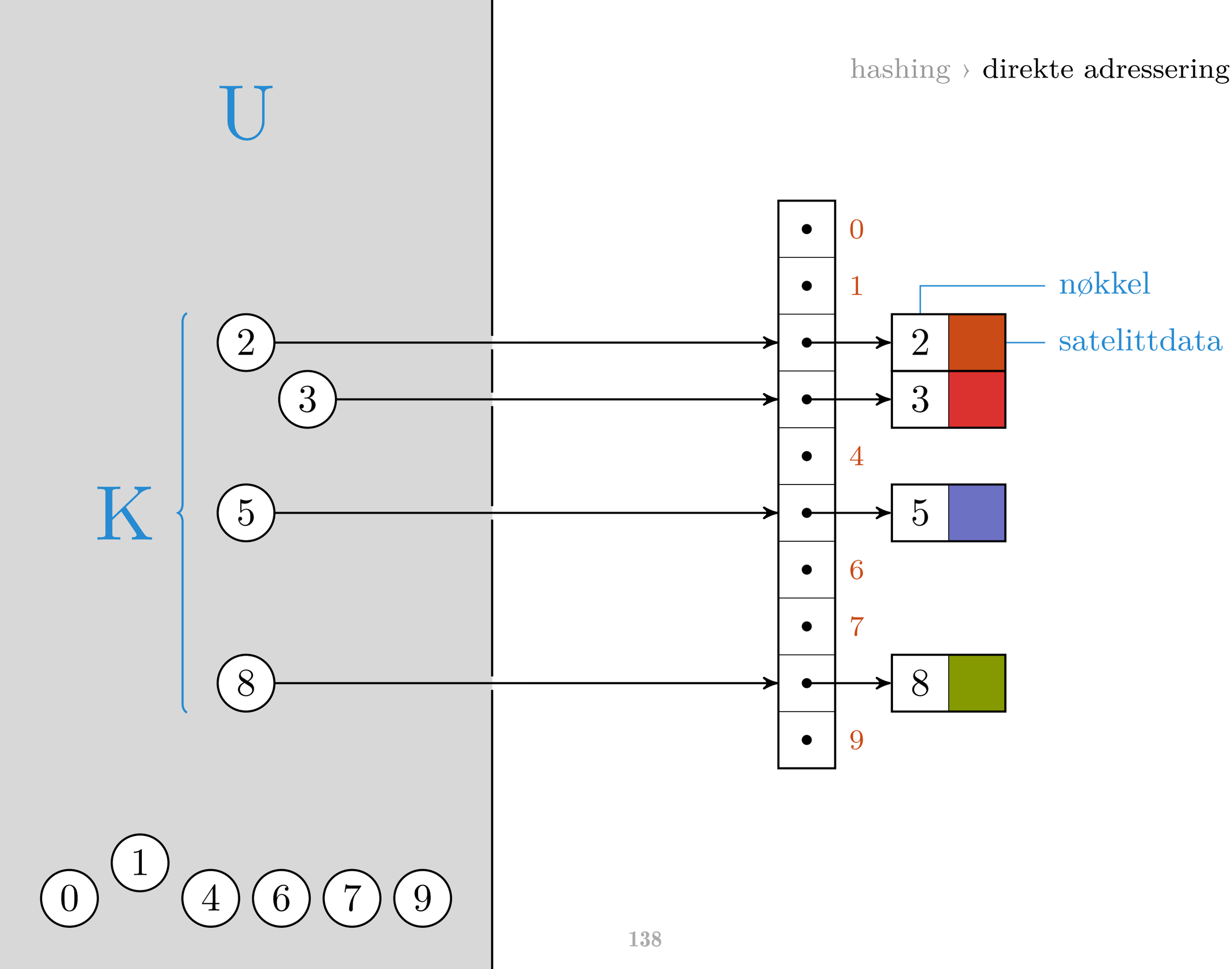
3:4

Hashtabeller

Direkte adressering

Nøkkel = indeks

Egentlig bare en «myk start» på hashing. Vi har en verdi k som vi bruker som nøkkel i en oppslagstabell. Direkte adressering er å bare bruke k som indeks, direkte.



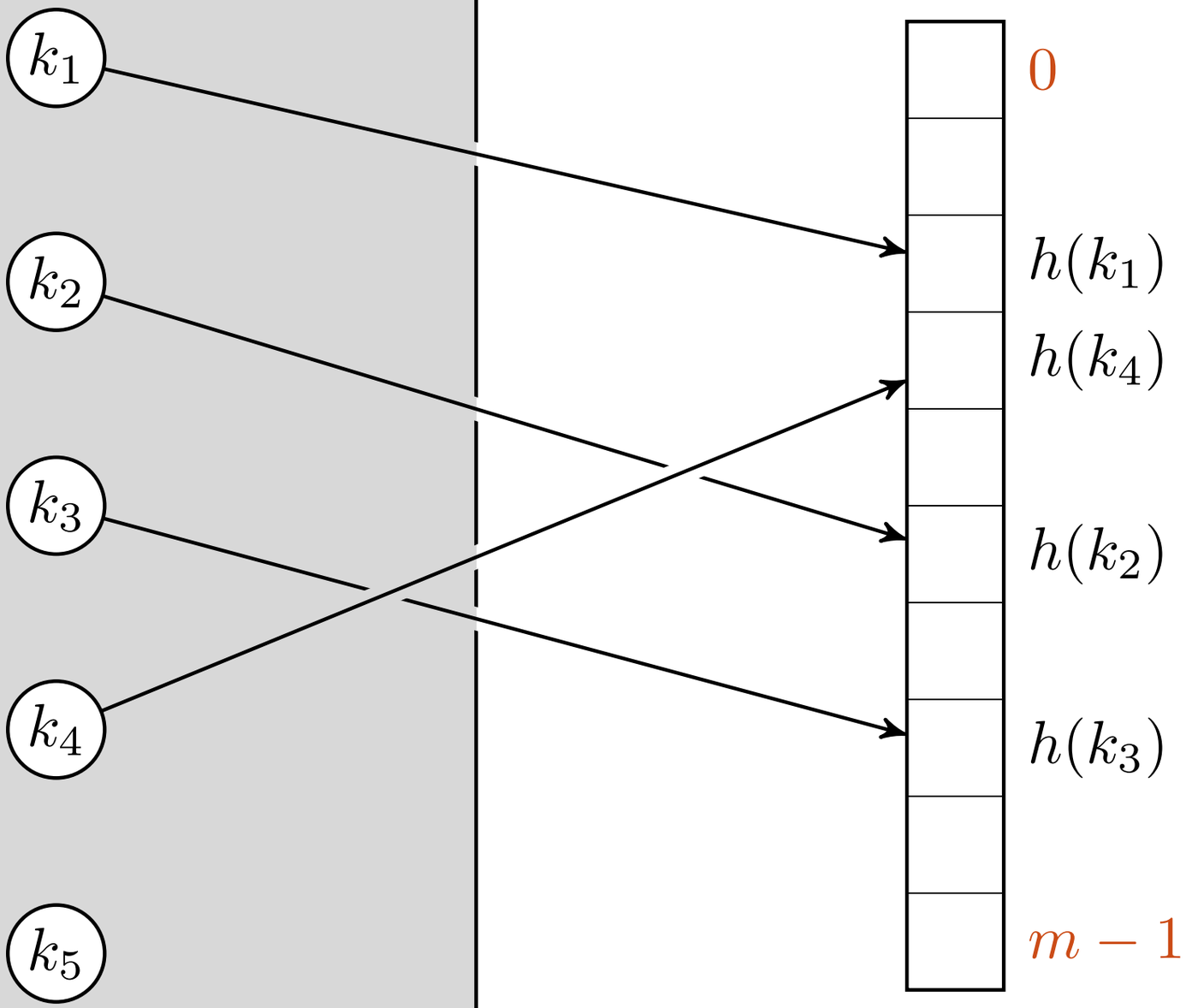
Vi ønsker å tillate store nøkler uten å ha store tabeller. Vi kan da «knøvle» nøkkelen til å bli en akseptabel, tilsynelatende tilfeldig, indeks.

Hashtabeller

Modifisert nøkkel er indeks

Obs: Input til en hashfunksjon kan være en vilkårlig bitstreng, tolket som et tall. Vi kan godt hashe andre ting som strenger og mer kompliserte objekter.

Vi stapper objektene inn i hashfunksjonen og får en gyldig indeks ut.



Hashing: Regn ut en indeks fra nøkkelverdien.

$$h(k) = \lfloor km \rfloor \quad 0 \leq k < 1$$

$$h(k) = k \bmod m$$

$$h(k) = \lfloor m(k\mathrm{A} \bmod 1) \rfloor$$

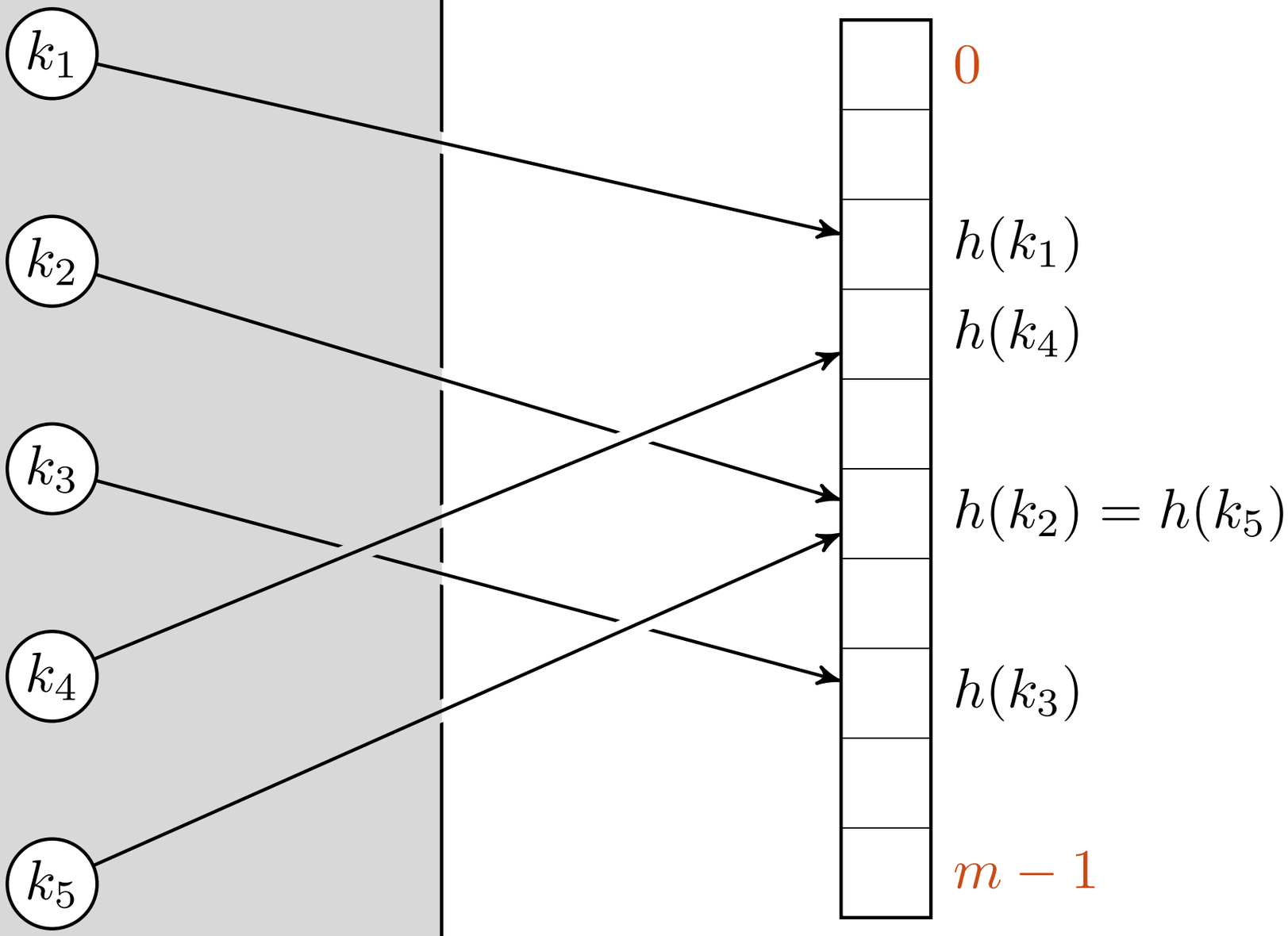
$$0 < \mathrm{A} < 1$$

f.eks.

Strenger e.l.?

Regn ut et heltall

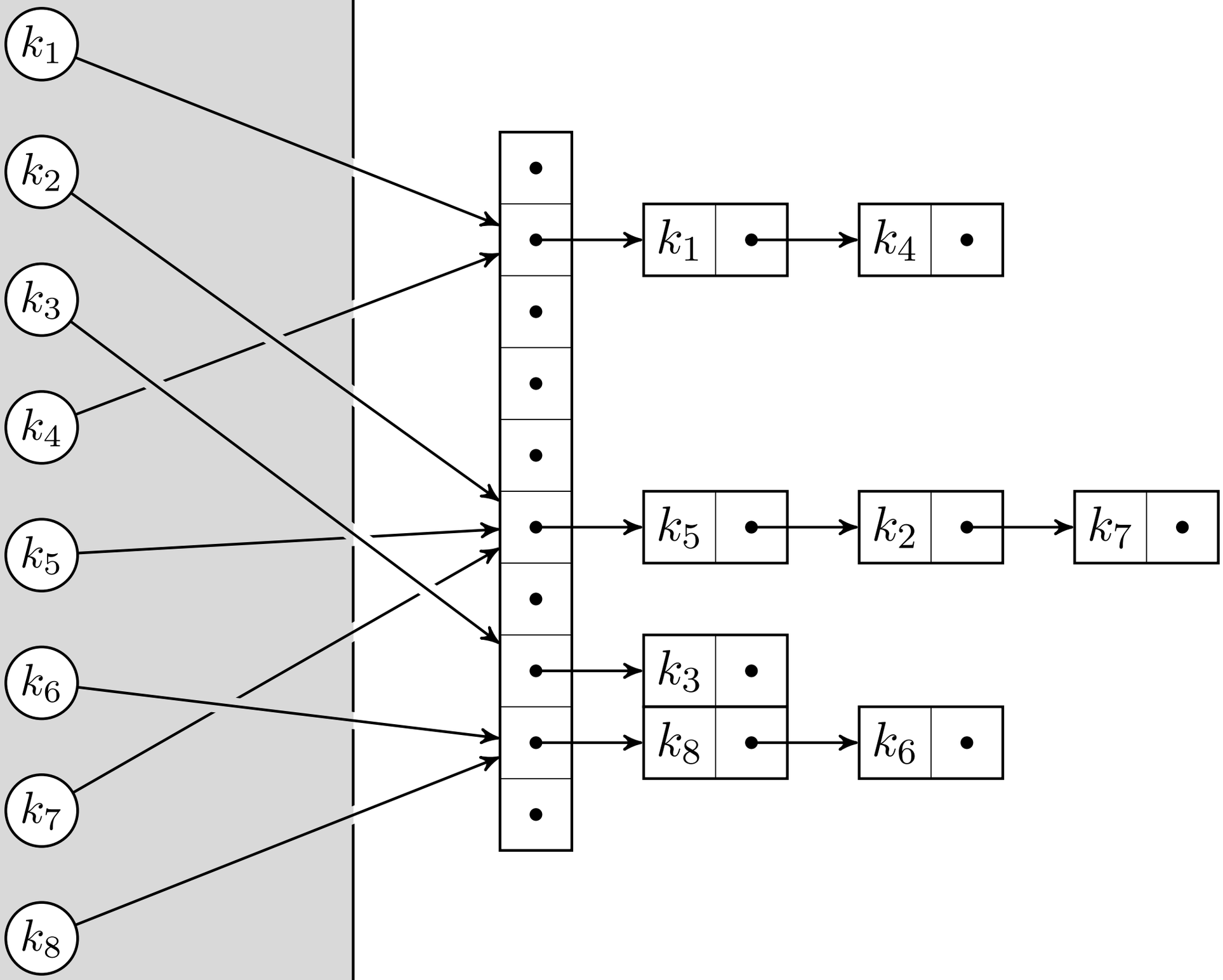
F.eks. behandle tegn som siffer



Kjeding (chaining)

Hver posisjon har en liste

Om to verdier hasher til samme indeks, så har vi en kollisjon; for å ta vare på begge verdiene, kan vi ha en lenket liste (f.eks.) i hver celle i tabellen.



- Mange kollisjoner: Lineært lange lister
- Søk vil ta lineær tid
- Anta lineært stor tabell
- Anta jevn, «tilfeldig» fordeling
- Konstant forventet kjøretid!

Statisk datasett?

Lag custom hashfunksjon!

Kan da garantere konstant kjøretid

4:4

Dynamiske tabeller

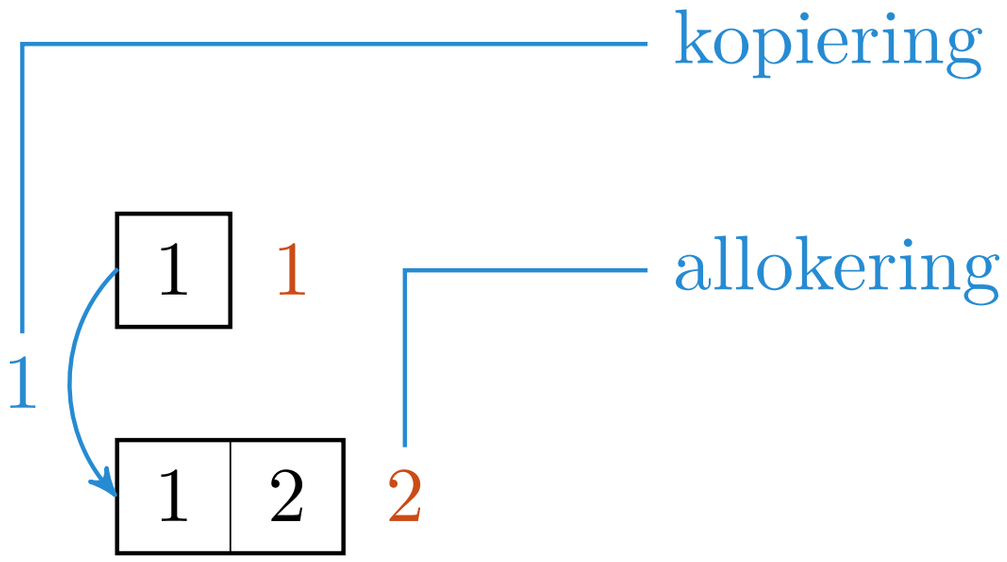
- **Hva om en hashtabell blir for full?**
- **Eller hva med en stakk eller kø?**
- **Vi kan allokere nytt minne og kopiere**
- **Men det tar jo lineær tid ...**
- **... så vi vil gjøre det sjelden!**
- **Vi tar i, og allokerer mye minne**

Amortisert arbeid

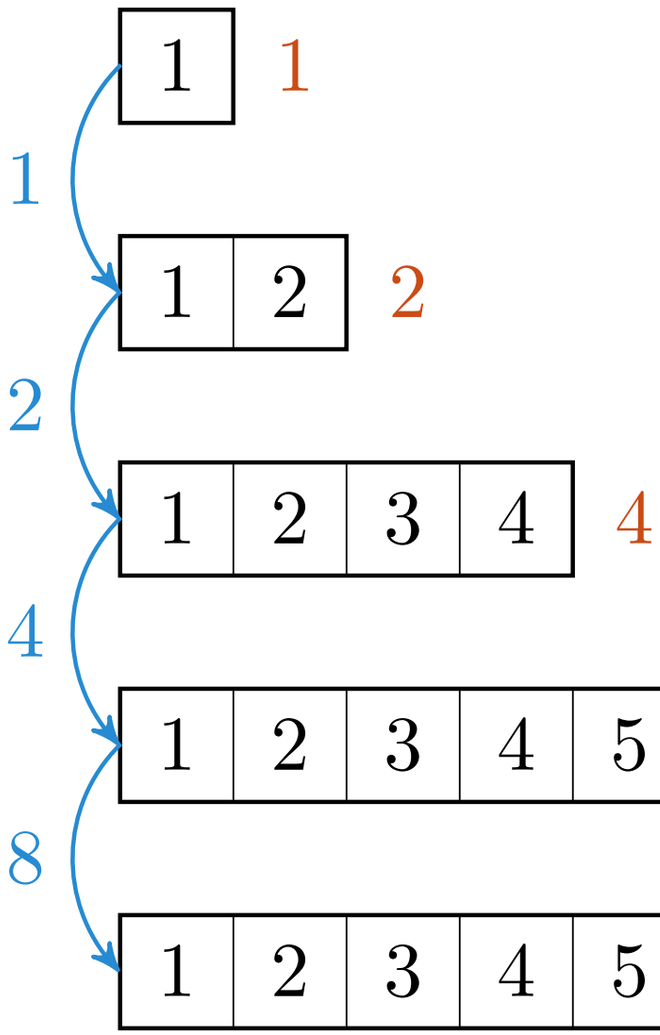
- Kjøretid for én enkelt operasjon: Ikke altid informativt
- Se på gjennomsnitt per operasjon etter at mange har blitt utført!

Husk denne!

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$



Vi bør jo ta med deallokering av den forrige tabellen òg – men det endrer ikke det asymptotiske resultatet.



$$\sum_{i=0}^{h-1} 2^i + \sum_{i=0}^{h} 2^i = (2^h - 1) + (2^{h+1} - 1)$$

$$= (n - 1) + (2n - 1)$$

$$= 3n - 2 = 46$$

$$h = \log_2 n = 4$$

$$\text{Ekstra arbeid per insetting: } \frac{3n-2}{n} = \Theta(1)$$

Snitt-kjøretid 1 & 2

Avg-case og amortisering

Average-case: Forventet kjøretid.

Avg-case

Snitt over instanser

Amortisering

Snitt over operasjoner

1. Stakker og køer

2. Lenkede lister

3. Hashtabeller

4. Dynamiske tabeller

Neste gang: Splitt og hersk (divide & conquer)!

Husk sum

mene!