Forelesning 2a

For å unngå grunnleggende kjøretidsfeller er det viktig å kunne organisere og strukturere data fornuftig. Her skal vi se på hvordan enkle strukturer kan implementeres i praksis, og hva vi vinner på å bruke dem i algoritmene våre.

Støttelitteratur

- ☐ Innledningen til del III
- ☐ 10. Elementary data structures: Innl., 10.1 og 10.2
- ☐ 11. Hash tables: Tom. 11.3.1
- ☐ 16. Amortized analysis: Innl., 16.1 og s. 460–463 (tom. *at most 3*)

Læringsmål

Fra pensumheftet

- [B_1] Forstå *tabeller*, *matriser*, *stakker* og *køer*
- [B_2] Forstå *lenkede lister*
- [B_3] Forstå *hashtabeller*
- [B_4] Kjenne *perfekt hashing*; O(1) søk for statiske data
- [B_5] Forstå tabelldobling

mlh@ntnu.no

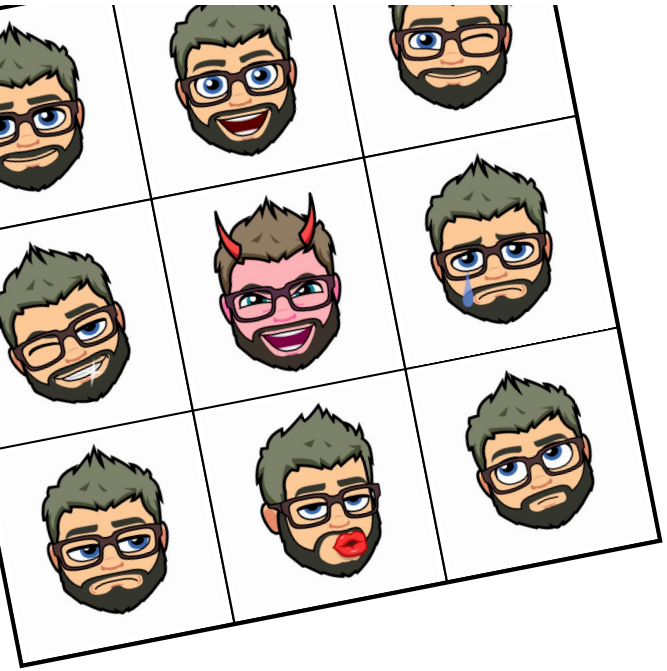
Forelesningen filmes

s.ntnu.no/video-opptak

Forelesning 2

Datastrukturer

1. **Tabeller og matriser**
2. **Stakker og køer**
3. **Lenkede lister**
4. **Hashtabeller**
5. **Dynamiske tabeller**



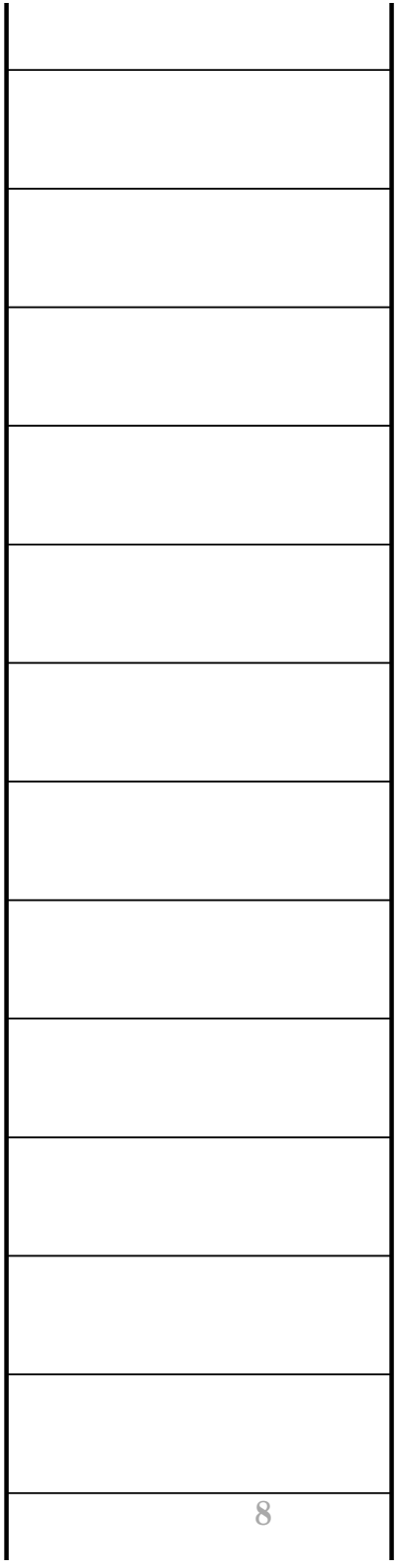
1:5

Tabeller og matriser

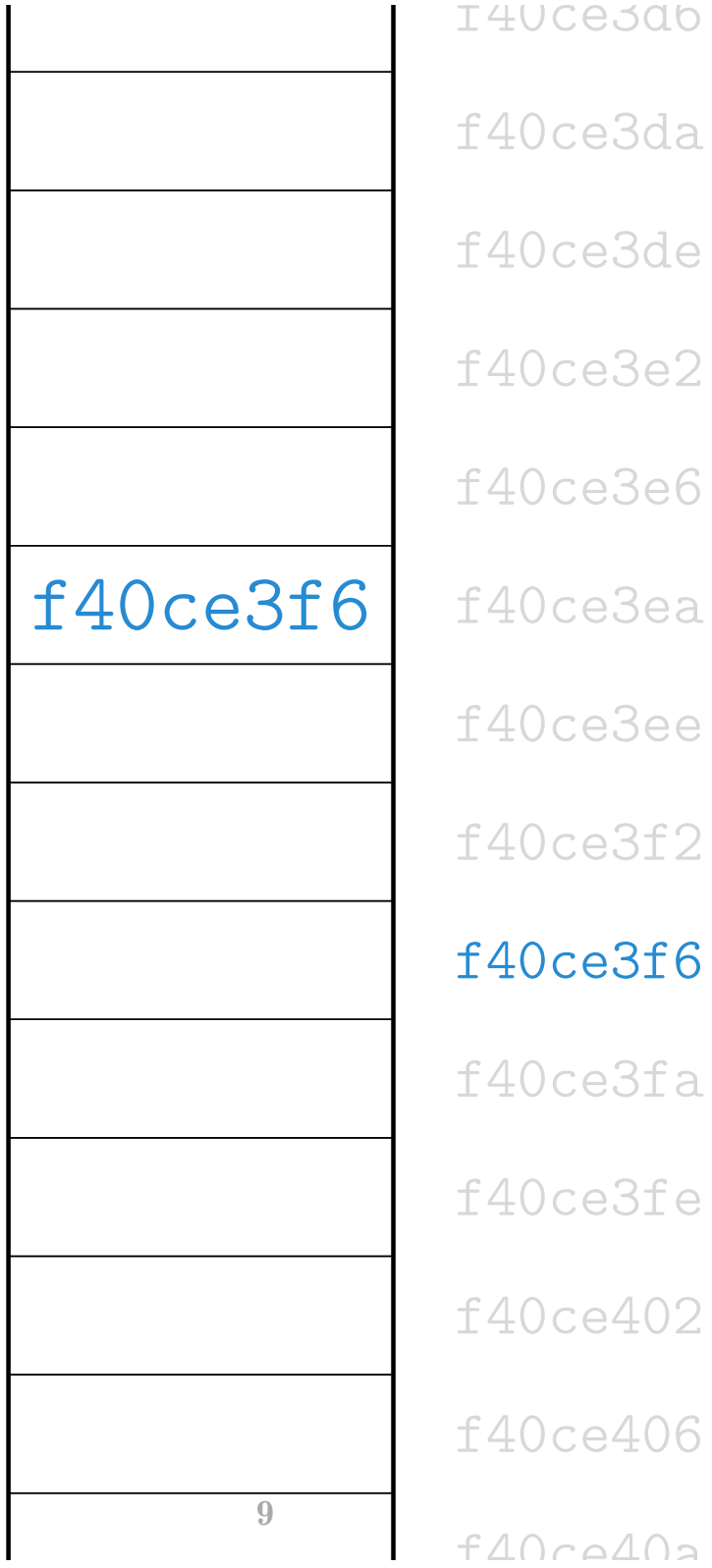
Hver byte (ev. hvert ord) i minnet har en adresse.

En peker er en slik adresse.

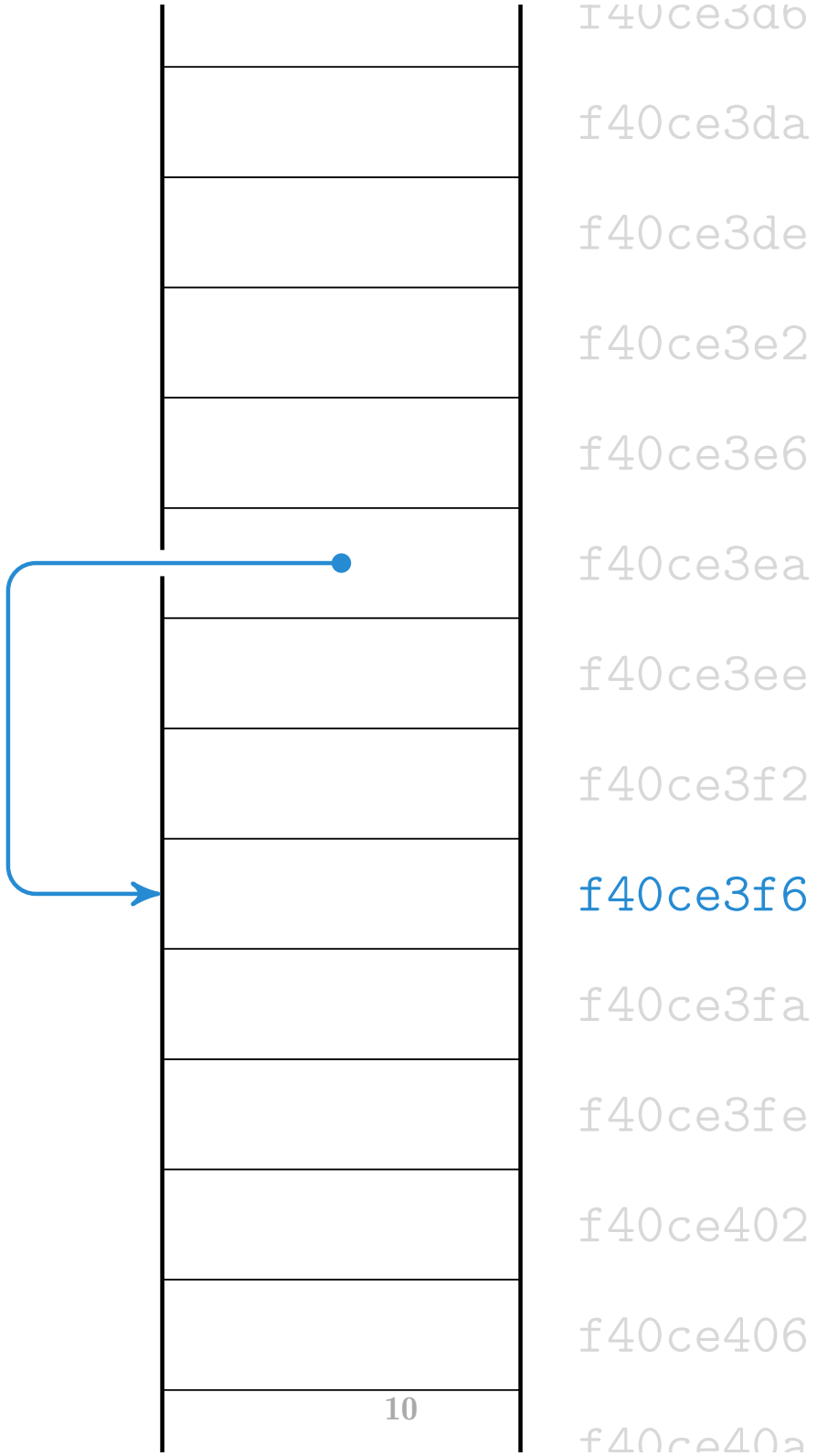
Her har jeg gruppert minnet i ord (words), altså flere bytes (typisk 4 eller 8, avhengig av maskin).



En variabel (et ord) som inneholder et heltall – som her tolkes som adressen til en *annen* variabel/et annet ord.

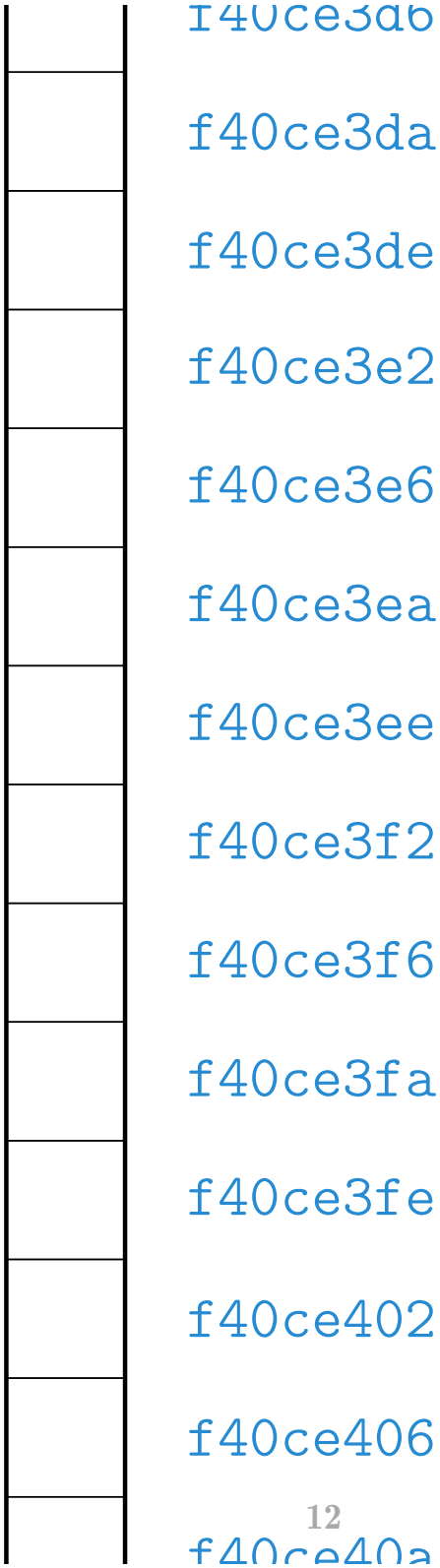


Samme peker, bare illustrert på en annen måte.

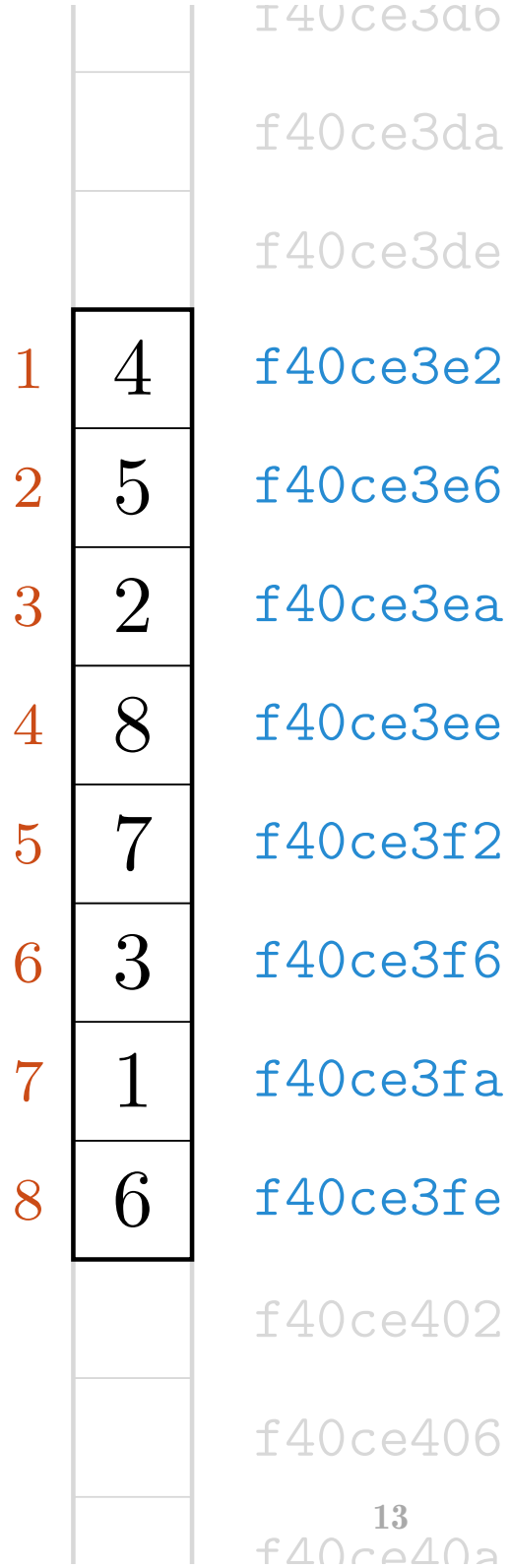


mlh@ntnu.no

Éndimensjonale tabeller (arrays) representeres gjerne av en sammenhengende minneblokk.

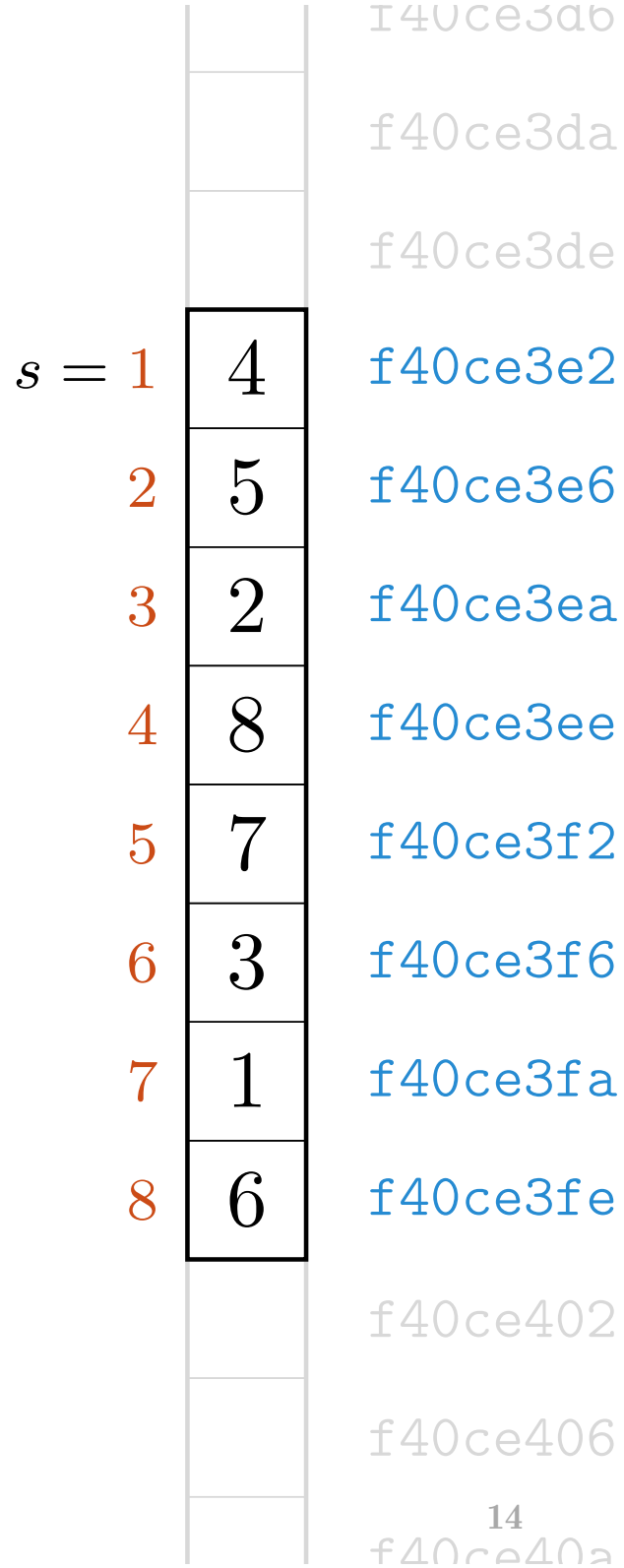


Minnet



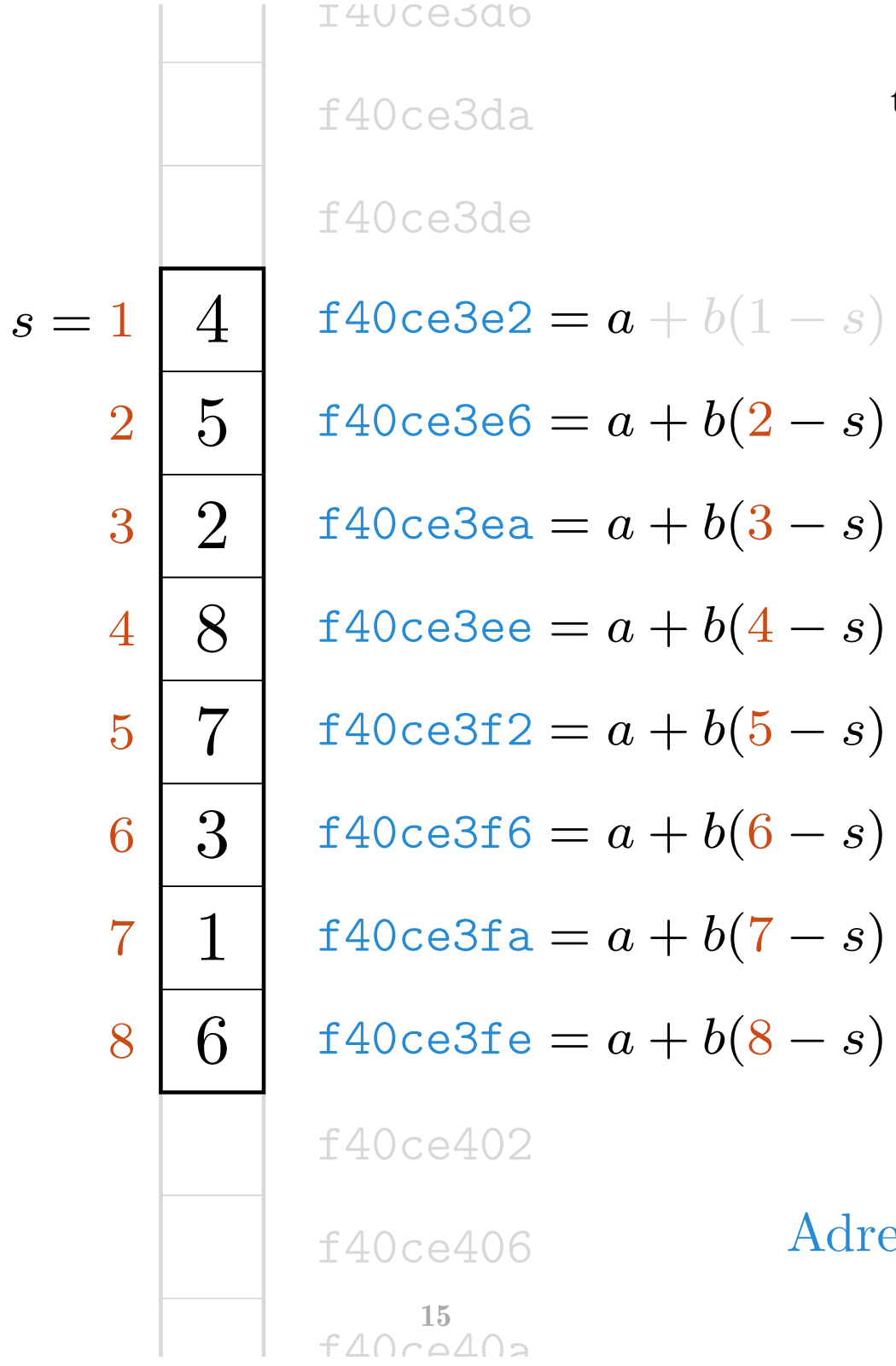
Tabell

mlh@ntnu.no



Startindeks

mlh@ntnu.no



Adresser

mlh@ntnu.no

		f40ce3d6
		f40ce3da
		f40ce3de
$s = 1$	4	f40ce3e2 $= a + b(1 - s)$
2	5	f40ce3e6 $= a + b(2 - s)$
3	2	f40ce3ea $= a + b(3 - s)$
4	8	f40ce3ee $= a + b(4 - s)$
5	7	f40ce3f2 $= a + b(5 - s)$
6	3	f40ce3f6 $= a + b(6 - s)$
7	1	f40ce3fa $= a + b(7 - s)$
8	6	f40ce3fe $= a + b(8 - s)$
		f40ce402
		f40ce406
		f40ce40a

$(b = 4)$

mlh@ntnu.no

Todimensjonale tabeller, eller matriser, representeres gjerne ved hjelp av én eller flere endimensjonale tabeller.

«Row-major order»

	1	2	3
1	1	2	3
2	4	5	6

Matrise

1	1
2	2
3	3
4	4
5	5
6	6

Radvis lagring

mlh@ntnu.no

	1	2	3
1	1	2	3
2	4	5	6

$\mathrm{M}[i, j]$

1	1
2	2
3	3
4	4
5	5
6	6

$\mathrm{A}[\,3(i-1)+(j-1)+1\,]$

«Column-major order»

	1	2	3
1	1	2	3
2	4	5	6

Matrise

1	1
2	4
3	2
4	5
5	3
6	6

Kolonnevis lagring

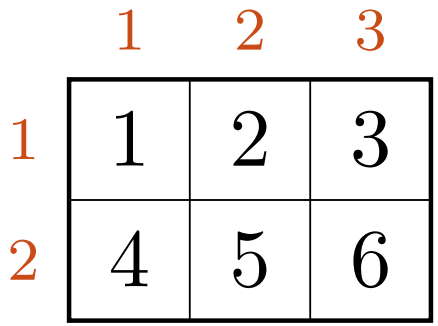
mlh@ntnu.no

	1	2	3
1	1	2	3
2	4	5	6

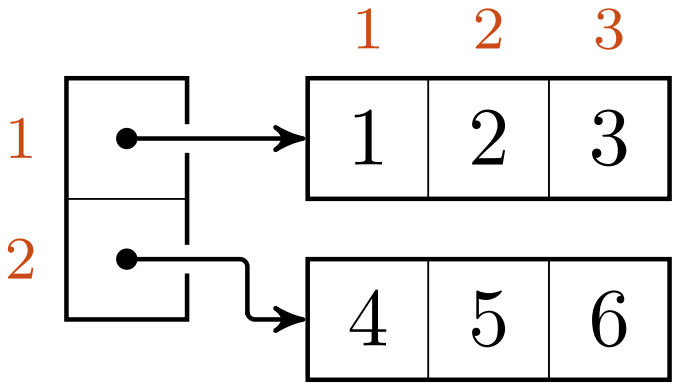
$\mathrm{M}[i, j]$

1	1
2	4
3	2
4	5
5	3
6	6

$\mathrm{A}[\,2(j-1)+(i-1)+1\,]$

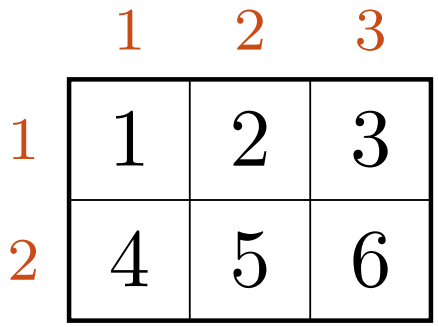


Matrise

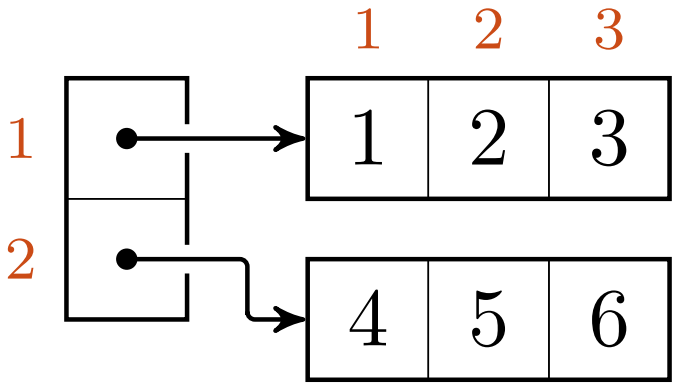


Tabell per rad

En slik implementasjon kan også brukes til «jagged/ragged arrays» der radene har ulik lengde.



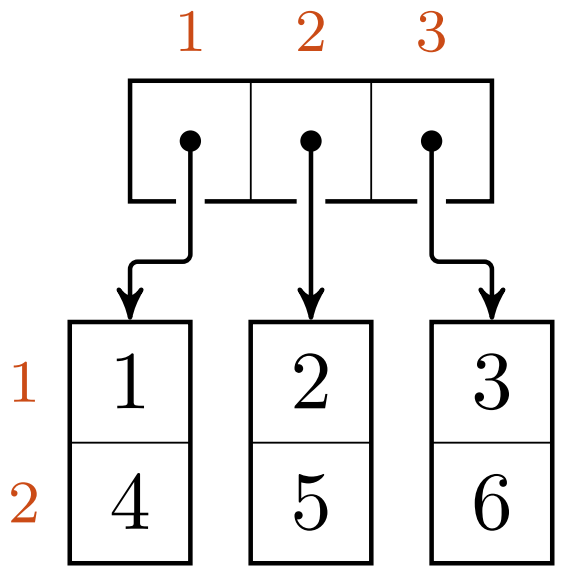
Matrise



(Peker = startadresse)

	1	2	3
1	1	2	3
2	4	5	6

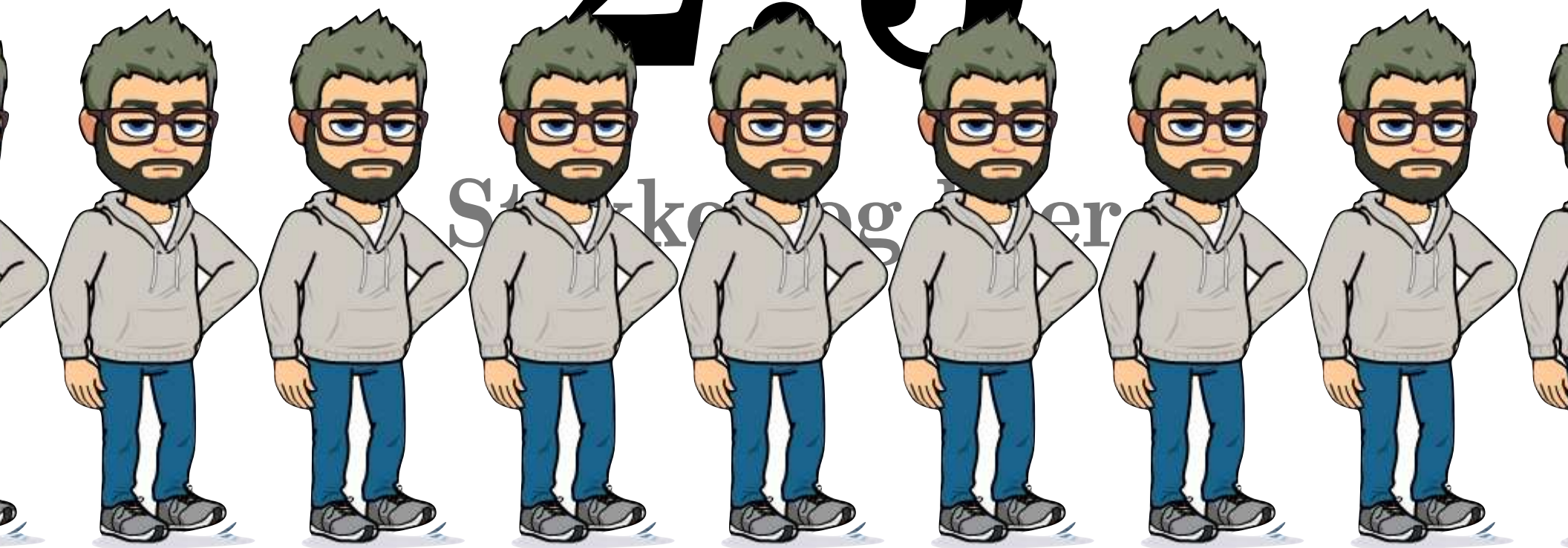
Matrise



Tabell per kolonne

Disse strategiene kan også utvides til flerdimensjonale tabeller.

2:5



LIFO: Last in, first out.

Stakker

Kun adgang øverst

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

STACK-EMPTY(S)

Er stakken tom?

mlh@ntnu.no

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

top er øverste indeks, dvs. antall elementer

mlh@ntnu.no

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

Ingen elementer, så stakken er tom

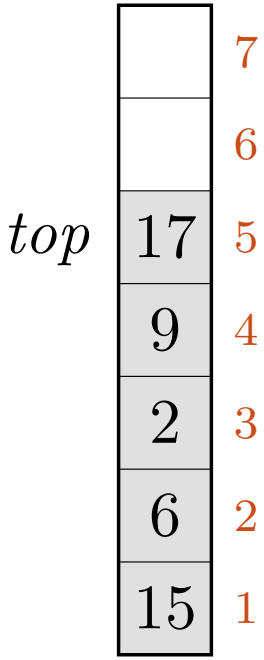
mlh@ntnu.no

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

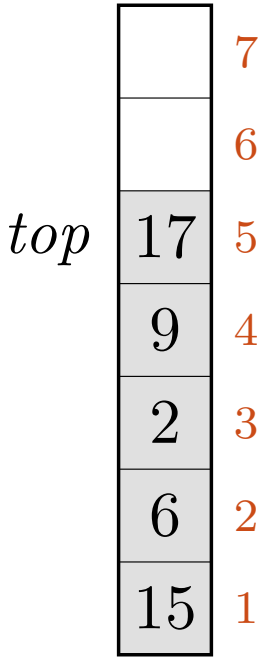
Vi har elementer, så stakken er ikke tom

mlh@ntnu.no

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

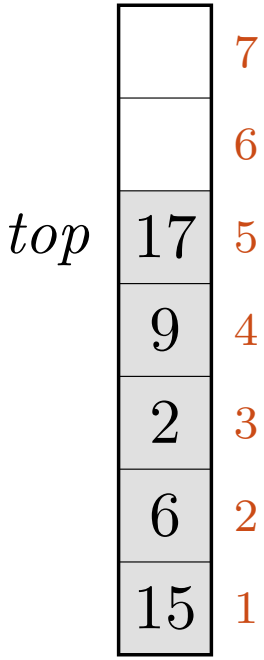


```
STACK-EMPTY(S)
1  if S.top == 0
2        return TRUE
3  else return FALSE
```



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

→ FALSE



PUSH(S, x)

S stakk
x nytt element

Legg et element på toppen

mlh@ntnu.no

PUSH(S, x)
1 **if** S.top == S.$size$

S stakk
x nytt element

Er stakken alt full?

```
PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
```

S stakk
x nytt element

Da feiler PUSH

mlh@ntnu.no

```
PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
3  else S.top = S.top + 1
```

S stakk
x nytt element

Ellers: Øk antallet med én

mlh@ntnu.no

```
PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
3  else S.top = S.top + 1
4       S[S.top] = x
```

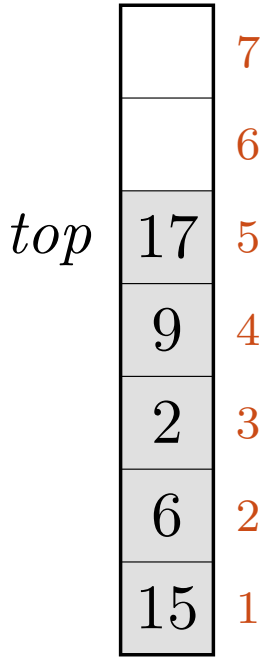
S stakk
x nytt element

Øverste indeks refererer nå til x

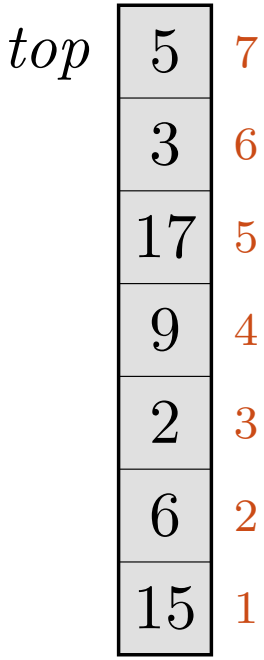
mlh@ntnu.no

```
PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
3  else S.top = S.top + 1
4       S[S.top] = x
```

$x = 3$



```
PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
3  else S.top = S.top + 1
4       S[S.top] = x
```



$x = 9$

```
ERROR: overflow
```

Pop(S)

Fjern ett element fra toppen

mlh@ntnu.no

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

```
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
```

Reduser antallet med én

mlh@ntnu.no

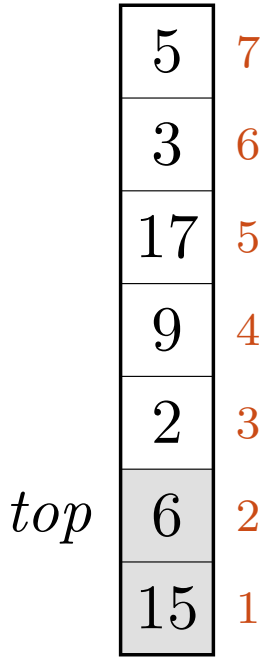
```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

Returner den gamle toppen (nå utenfor stakken)

mlh@ntnu.no

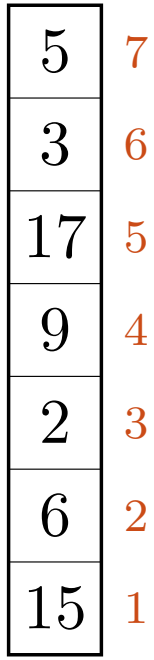
```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

S.*top* = 2



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

S.*top* = 0




```
ERROR: underflow
```

Køer

Først inn, først ut

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

ENQUEUE(Q, x)

Legg til et element bakerst i køen

mlh@ntnu.no

$\textsc{Enqueue}(\mathrm{Q}, x)$
1 $\quad \mathrm{Q}[\mathrm{Q}.\mathit{tail}] = x$

$\mathrm{Q}[\mathit{head}]$ er først i køen og $\mathrm{Q}[\mathit{tail}]$ er den ledige plassen bak

mlh@ntnu.no

ENQUEUE(Q, x)

1 $Q[Q.tail] = x$

2 **if** $Q.tail == Q.size$

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

mlh@ntnu.no

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

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

mlh@ntnu.no

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

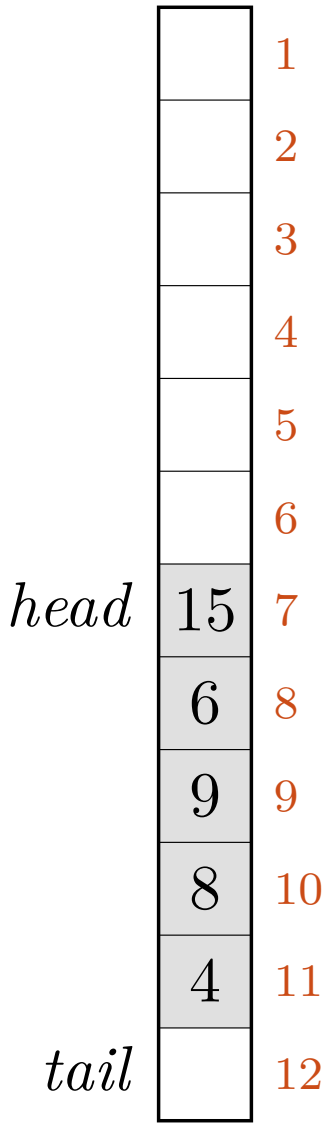
Har vi ledig plass, er det jo ikke noe problem

mlh@ntnu.no

ENQUEUE(Q, x)

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

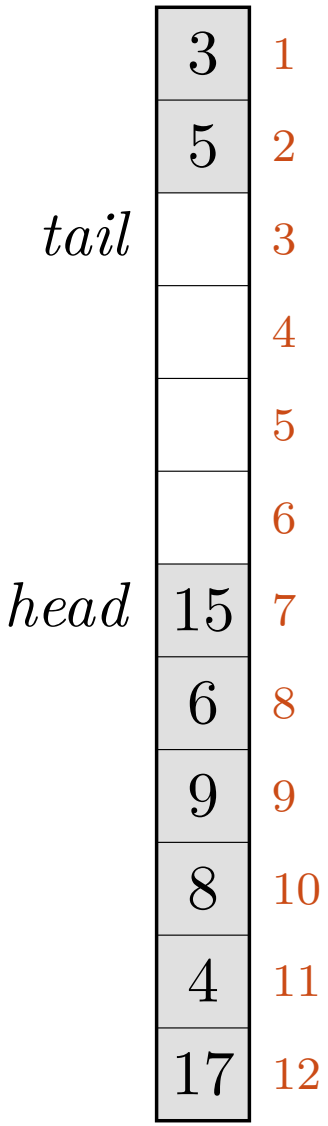
$x = 17$



ENQUEUE(Q, x)

```
1  Q[Q.tail] = x
2  if Q.tail == Q.size
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 $x = \mathrm{Q}[\mathrm{Q}.head]$

x er den som står først i køen

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

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

mlh@ntnu.no

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

Har vi ledig plass, er det jo ikke noe problem

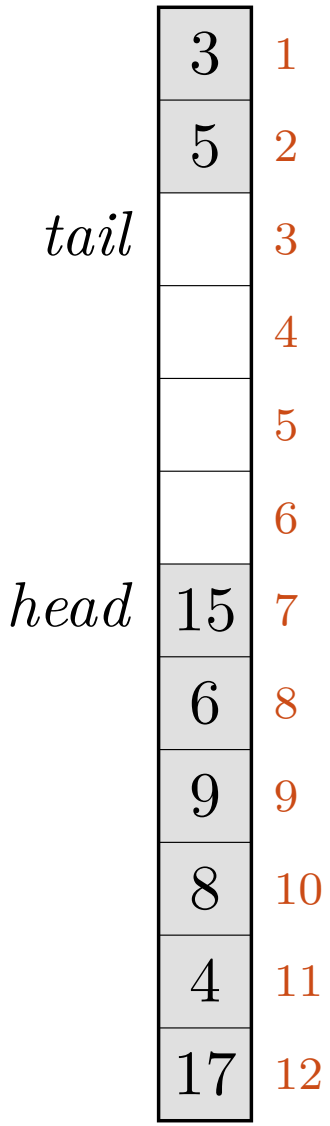
mlh@ntnu.no

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


mlh@ntnu.no

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

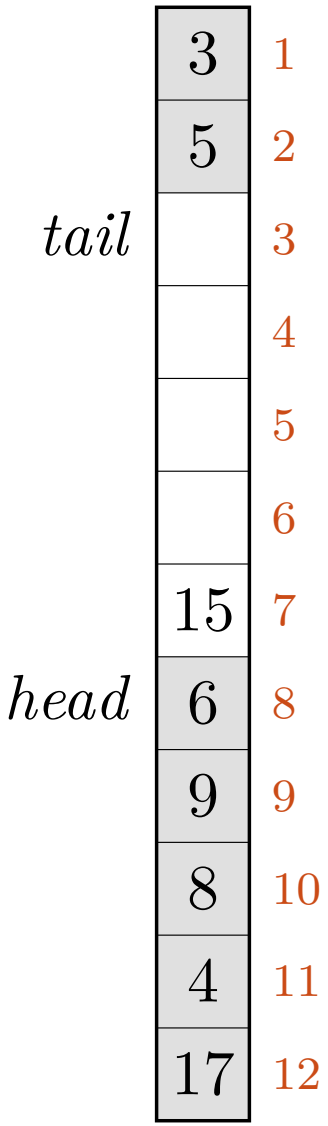
x = –



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

→ 15

$x = 15$



Dynamiske mengder

Et lite mellomspill

Ikke en bestemt datastruktur, men en type funksjonalitet.

Holder styr på en mengde elementer som har en nøkkel, og kanskje annen info.

Den andre informasjonen kan enten være såkalt satellittdata, som ignoreres av implementasjonen – eller det kan være data knyttet direkte til implementasjonen, som f.eks. pekere til andre elementer i mengden.

Mengde av objekter med nøkkelverdi.

Om nøklene er unike, kan vi se på det som en representasjon av en mengde nøkler.

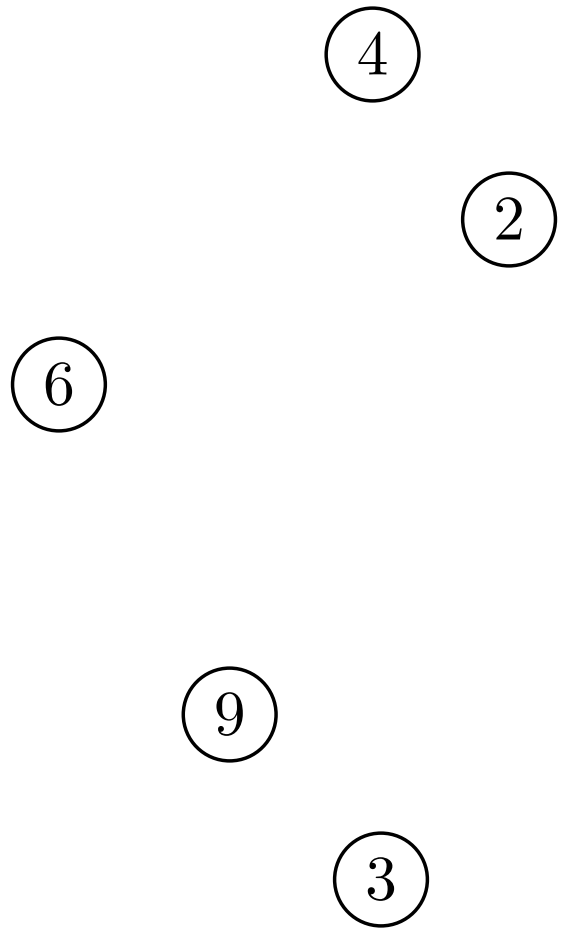
Enkleste variant: Kan bare søke etter nøkler, og legge til og fjerne elementer.

Om nøklene har en ordning, har vi også noen andre operasjoner.

1 SEARCH(S, k)
2 INSERT(S, x)
3 MINIMUM(S)
4 SUCCESSOR(S, x)
5 DELETE(S, x)

Om vi ikke finner noe element, returneres NIL. (Det gjelder også Minimum, Successor, etc.)

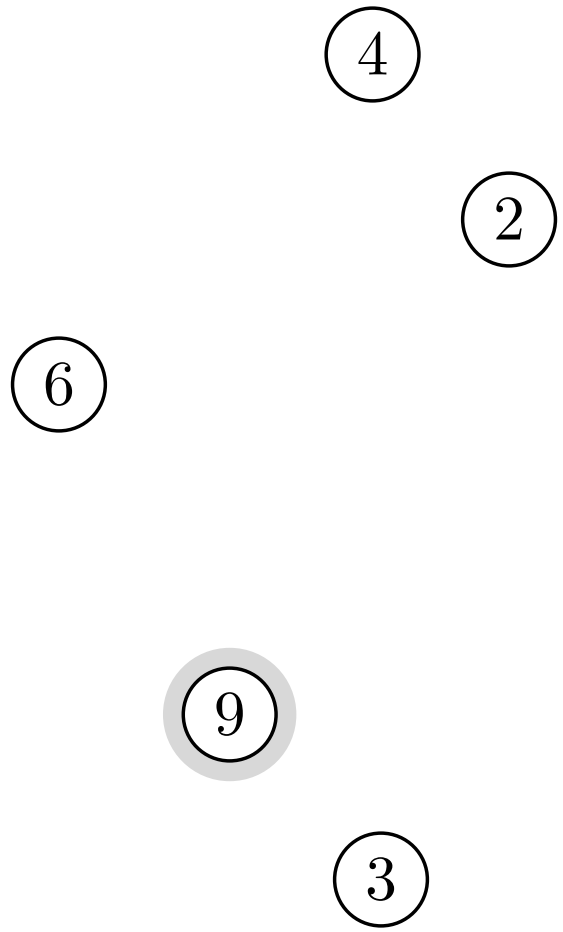
$k, x.key = 9, 5$



Finn y med $y.key = k$

1 SEARCH(S, k)
2 INSERT(S, x)
3 MINIMUM(S)
4 SUCCESSOR(S, x)
5 DELETE(S, x)

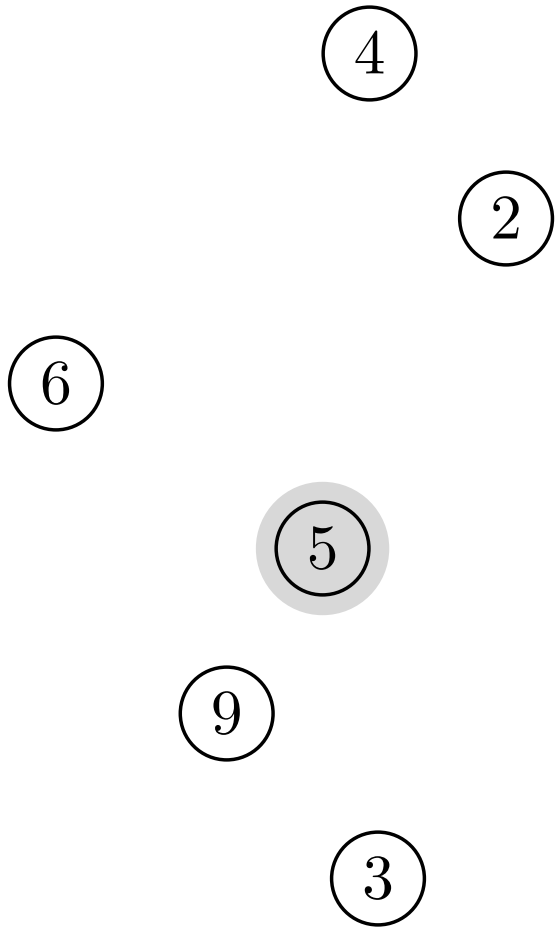
$k, x.key = 9, 5$



Sett inn x

Her støttes også Maximum(S).

1 SEARCH(S, k)
2 INSERT(S, x)
3 MINIMUM(S)
4 SUCCESSOR(S, x)
5 DELETE(S, x)

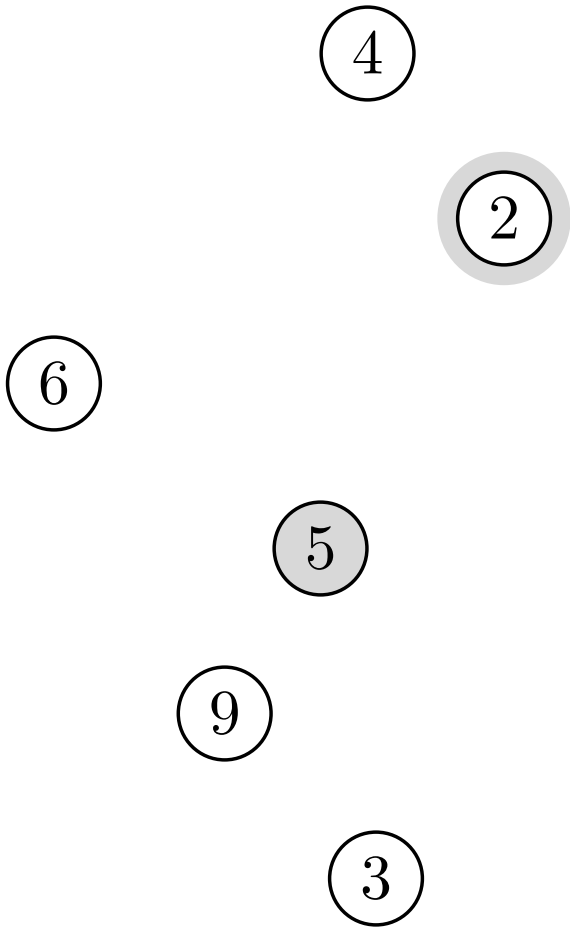


$k, x.key = 9, 5$

Finn y med minst $y.key$

Her støttes også Predecessor(S, x)

1 SEARCH(S, k)
2 INSERT(S, x)
3 MINIMUM(S)
4 SUCCESSOR(S, x)
5 DELETE(S, x)

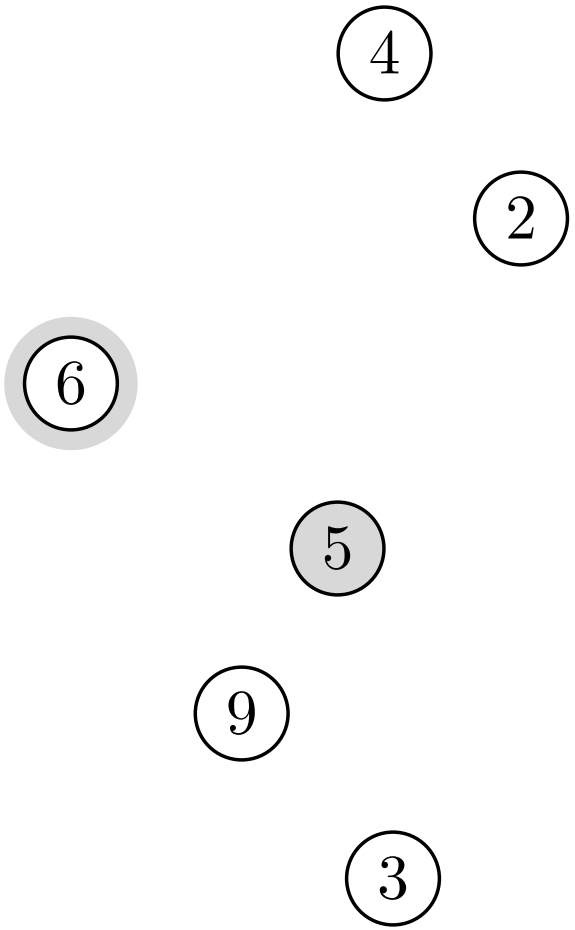


$k, x.key = 9, 5$

Neste etter x

1 SEARCH(S, k)
2 INSERT(S, x)
3 MINIMUM(S)
4 SUCCESSOR(S, x)
5 DELETE(S, x)

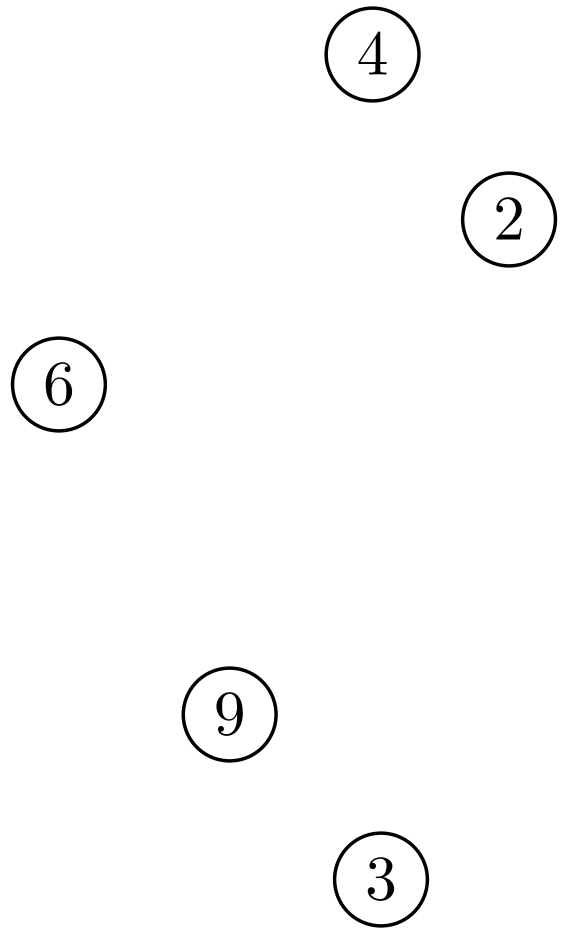
$k, x.key = 9, 5$



Slett x

1 SEARCH(S, k)
2 INSERT(S, x)
3 MINIMUM(S)
4 SUCCESSOR(S, x)
5 DELETE(S, x)

$k, x.key = 9, 5$



Problemstørrelsen her er vanligvis antall elementer i mengden.

Usortert tabell:

Innsetting og sletting i konstant tid, men resten blir lineært.

Sortert tabell:

Innsetting og sletting blir lineært, søk blir logaritmisk og resten blir konstant.

Søk i logaritmisk tid i en sortert tabell kommer vi tilbake til i neste forelesning.

Vi vil forbedre dette på ulike vis.

Noen eksempler i dag; flere i forelesning 5, om trær.

3:5

Lenkede lister

3:5

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

mlh@ntnu.no

LIST-SEARCH(L, k)
1 $x = \text{L}.head$

L liste
k nøkkelverdi
x nodepeker

Vi begynner med den første noden

LIST-SEARCH(L, k)

1 $x = \text{L}.head$

2 **while** $x \neq$ NIL and $x.key \neq k$

L liste
k nøkkelverdi
x nodepeker

Har vi nådd slutten eller funnet k?

mlh@ntnu.no

```
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 $x.key == k$

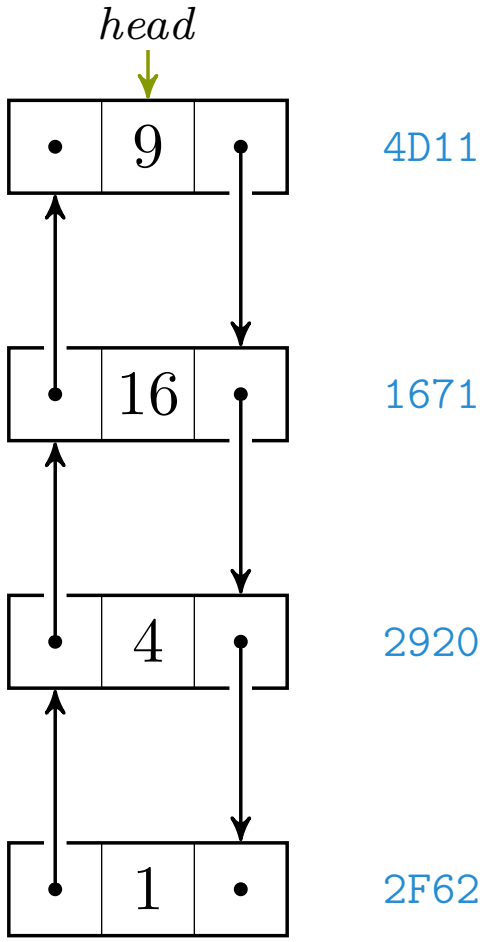
mlh@ntnu.no

LIST-SEARCH(L, k)

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

L.$head$, k, x = 4D11, 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.

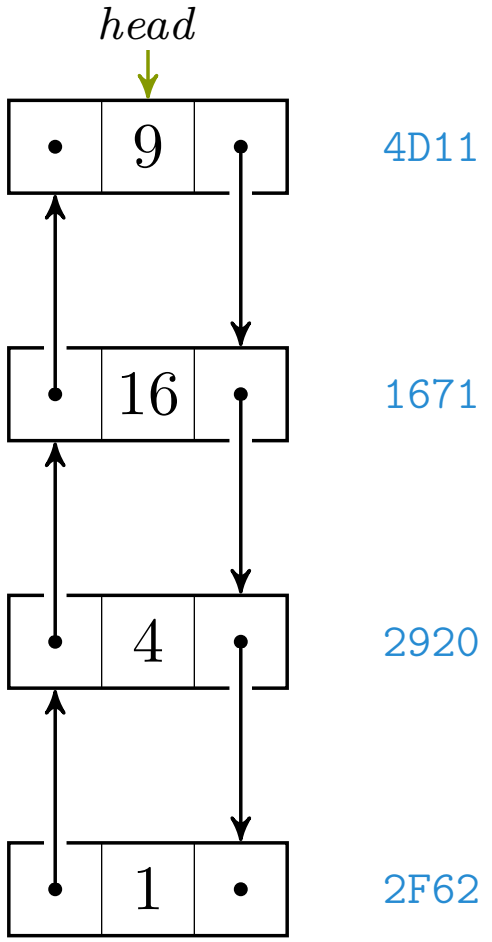


LIST-SEARCH(L, k)

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

→ NIL

L.$head, k, x$ = 4D11, 7, NIL



LIST-PREPEND(L, x)

L liste
x nytt hode

Sett noden x inn først i L

mlh@ntnu.no

LIST-PREPEND(L, x)

1 $x.next = \text{L}.head$

L liste
x nytt hode

Forrige hode kommer nå etter x

LIST-PREPEND(L, x)

1 $x.next = \text{L}.head$

2 $x.prev = \text{NIL}$

L liste

x nytt hode

Det er ingen noder før x

mlh@ntnu.no

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

L liste
x nytt hode

Forrige hode, om noe, har x som forgjenger

mlh@ntnu.no

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

L liste
x nytt hode

Forrige hode, om noe, har x som forgjenger

mlh@ntnu.no

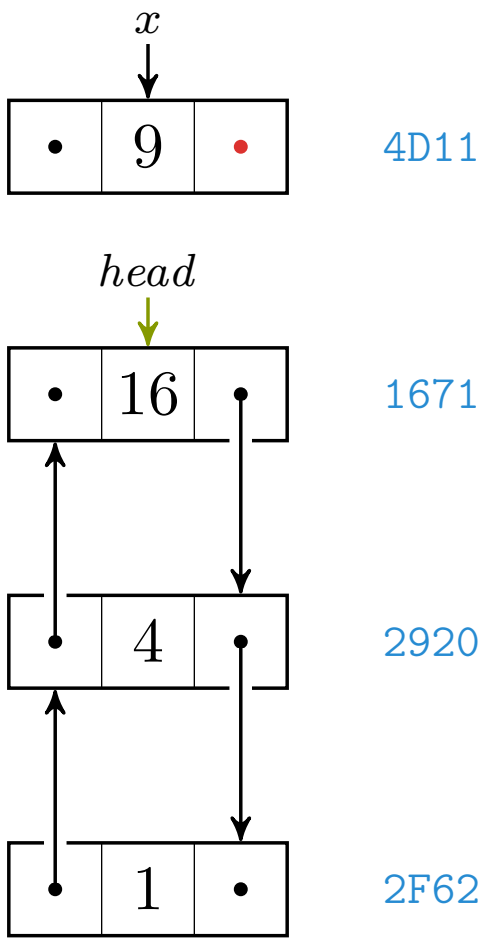
```
LIST-PREPEND(L, x)
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4        L.head.prev = x
5  L.head = x
```

L liste
x nytt hode

«Kryssreferanser» er på plass: Sett inn noden!

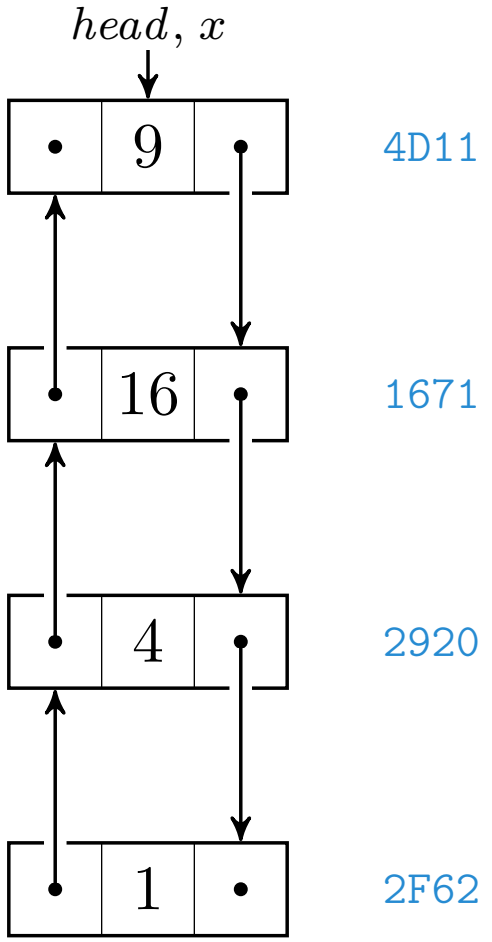
LIST-PREPEND(L, x)

```
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4        L.head.prev = x
5  L.head = x
```



L.$head, x$ = 1671, 4D11

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



L.*head*, x = 4D11, 4D11

LIST-INSERT(x, y)

x peker, ny node
y peker, forrige

Sett inn noden x etter noden y

mlh@ntnu.no

LIST-INSERT(x, y)

1 $x.next = y.next$

x peker, ny node
y peker, forrige

Noden etter y (eller NIL) havner etter x

LIST-INSERT(x, y)

1 $x.next = y.next$

2 $x.prev = y$

x peker, ny node

y peker, forrige

Forgjengeren til x vil bli y

mlh@ntnu.no

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

x peker, ny node
y peker, forrige

Var det en node etter y?

mlh@ntnu.no

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

x peker, ny node
y peker, forrige

Den får x som forgjenger

mlh@ntnu.no

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

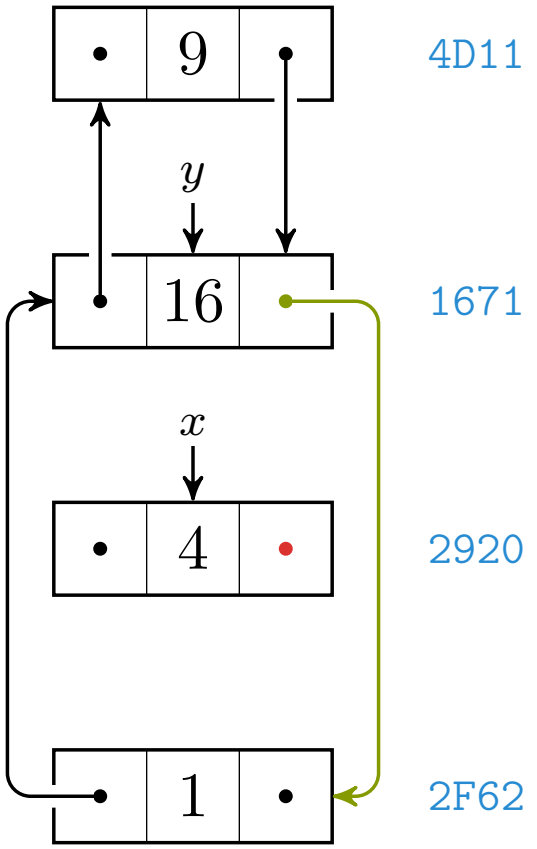
x peker, ny node
y peker, forrige

Og til slutt setter vi faktisk x etter y

mlh@ntnu.no

LIST-INSERT(x, y)

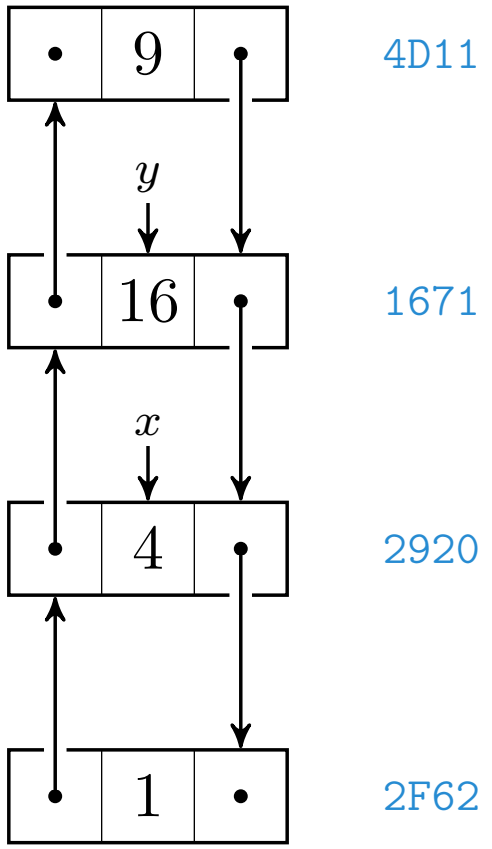
```
1  x.next = y.next
2  x.prev = y
3  if y.next ≠ NIL
4        y.next.prev = x
5  y.next = x
```



x, y = 2920, 1671

LIST-INSERT(x, y)

1 $x.next = y.next$
2 $x.prev = y$
3 **if** $y.next \neq$ NIL
4 $\quad y.next.prev = x$
5 $y.next = x$



$x, y = 2920, 1671$

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

mlh@ntnu.no

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

L liste
x skal fjernes

(Med mindre x står først . . .)

mlh@ntnu.no

```
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

mlh@ntnu.no

```
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

mlh@ntnu.no

```
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 . . .)

mlh@ntnu.no

```
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
```

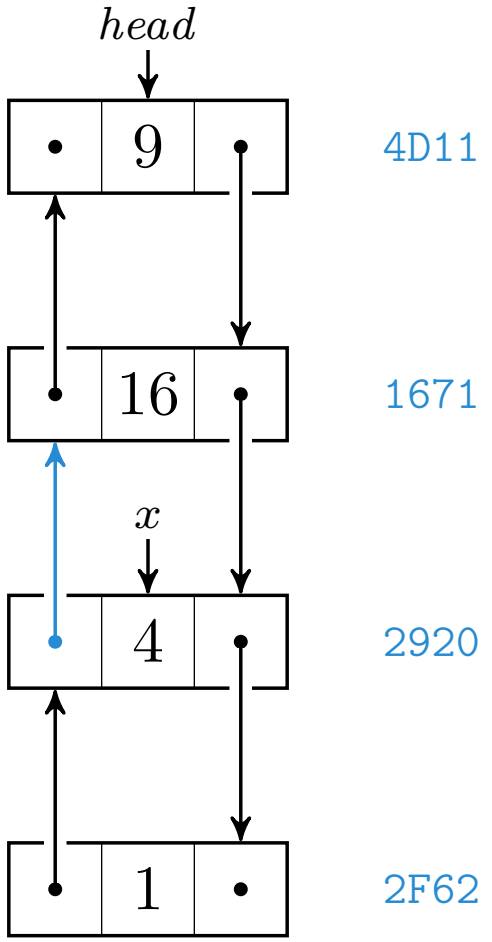
L liste
x skal fjernes

Etterkommeren arver x sin forgjenger

mlh@ntnu.no

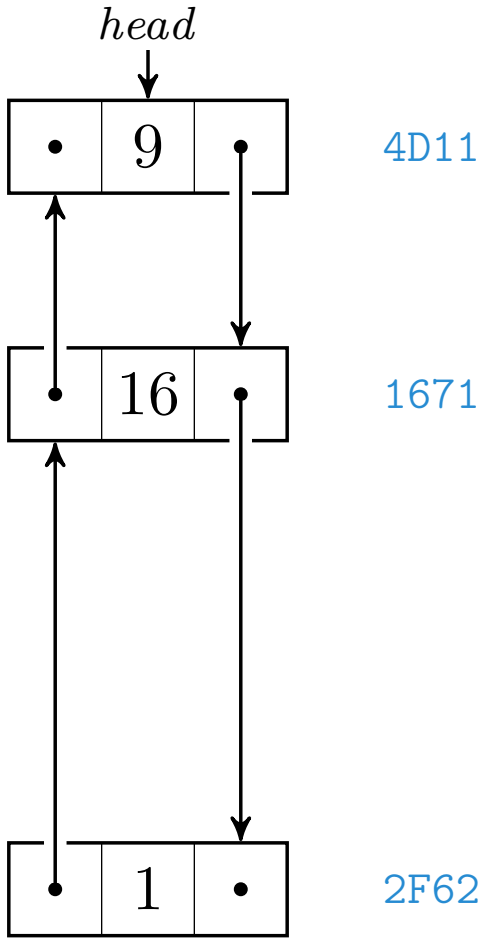
```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

L.*head*, x = 4D11, 2920



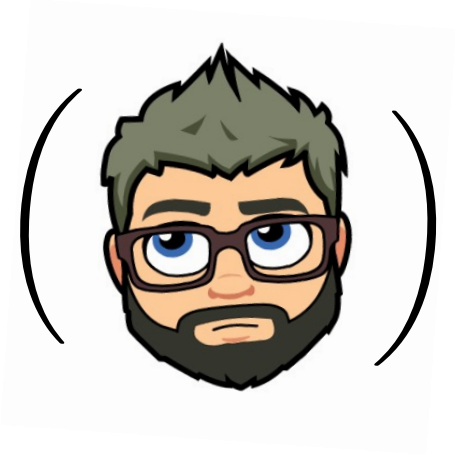
```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

L.*head*, x = 4D11, 2920



4:5

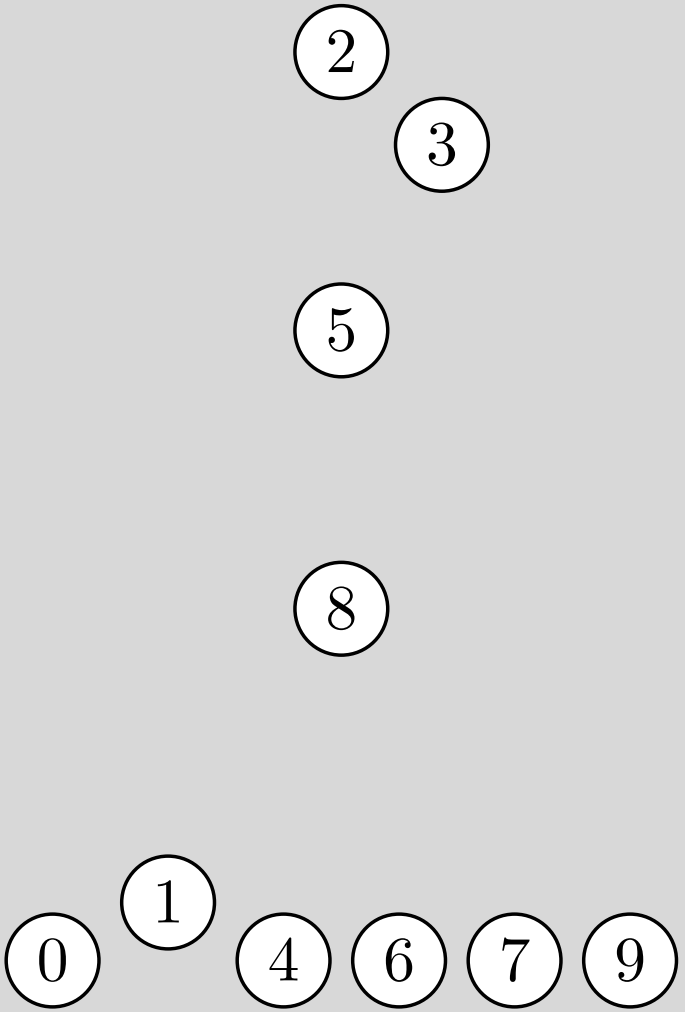
Hashtabeller

$h($  $) = 19312$

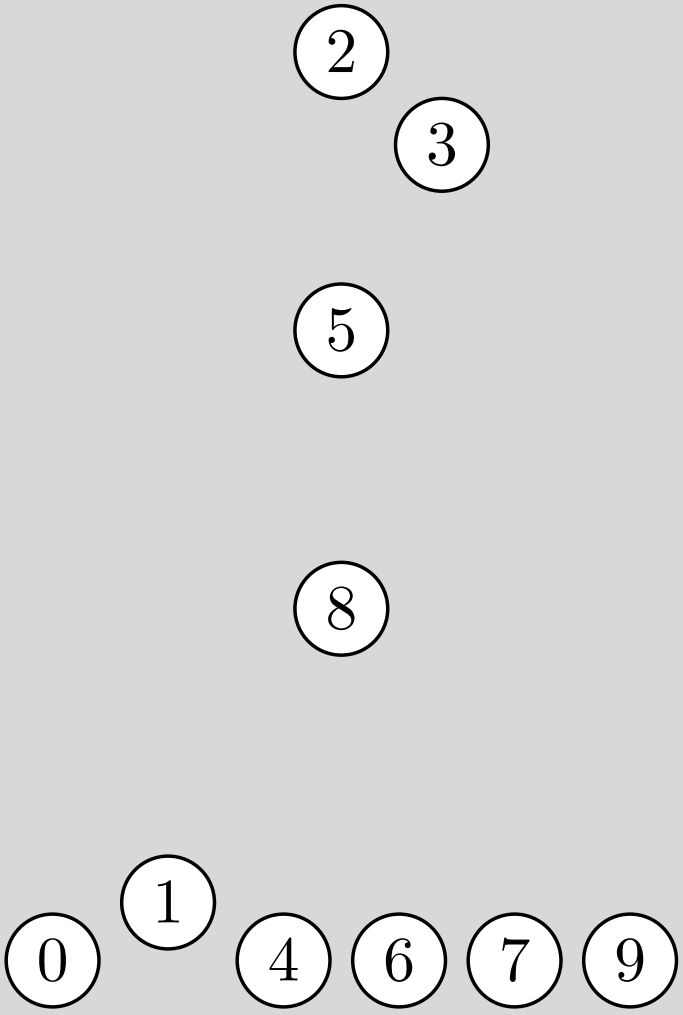
Direkte adressering

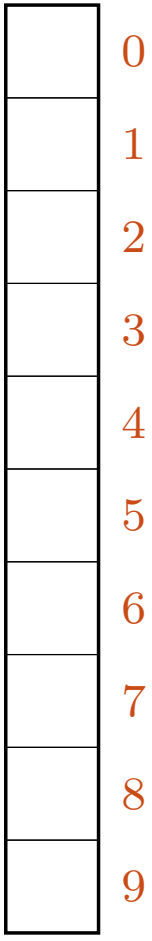
Nøkkel = indeks

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.

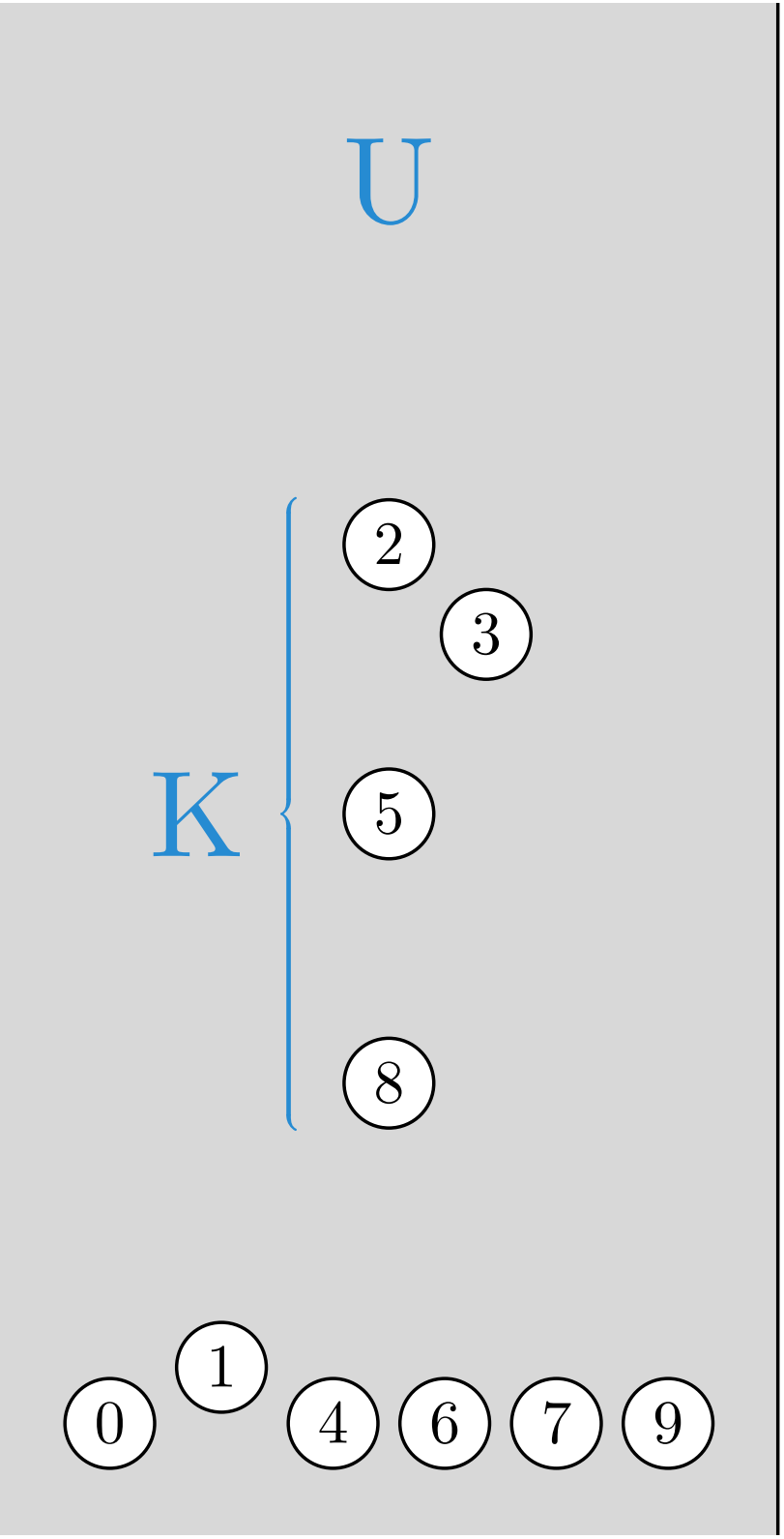


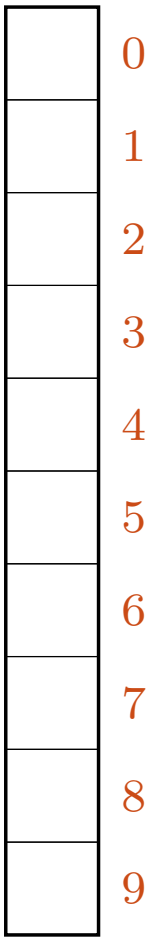
Merk at vi her bruker 0-indeksering!

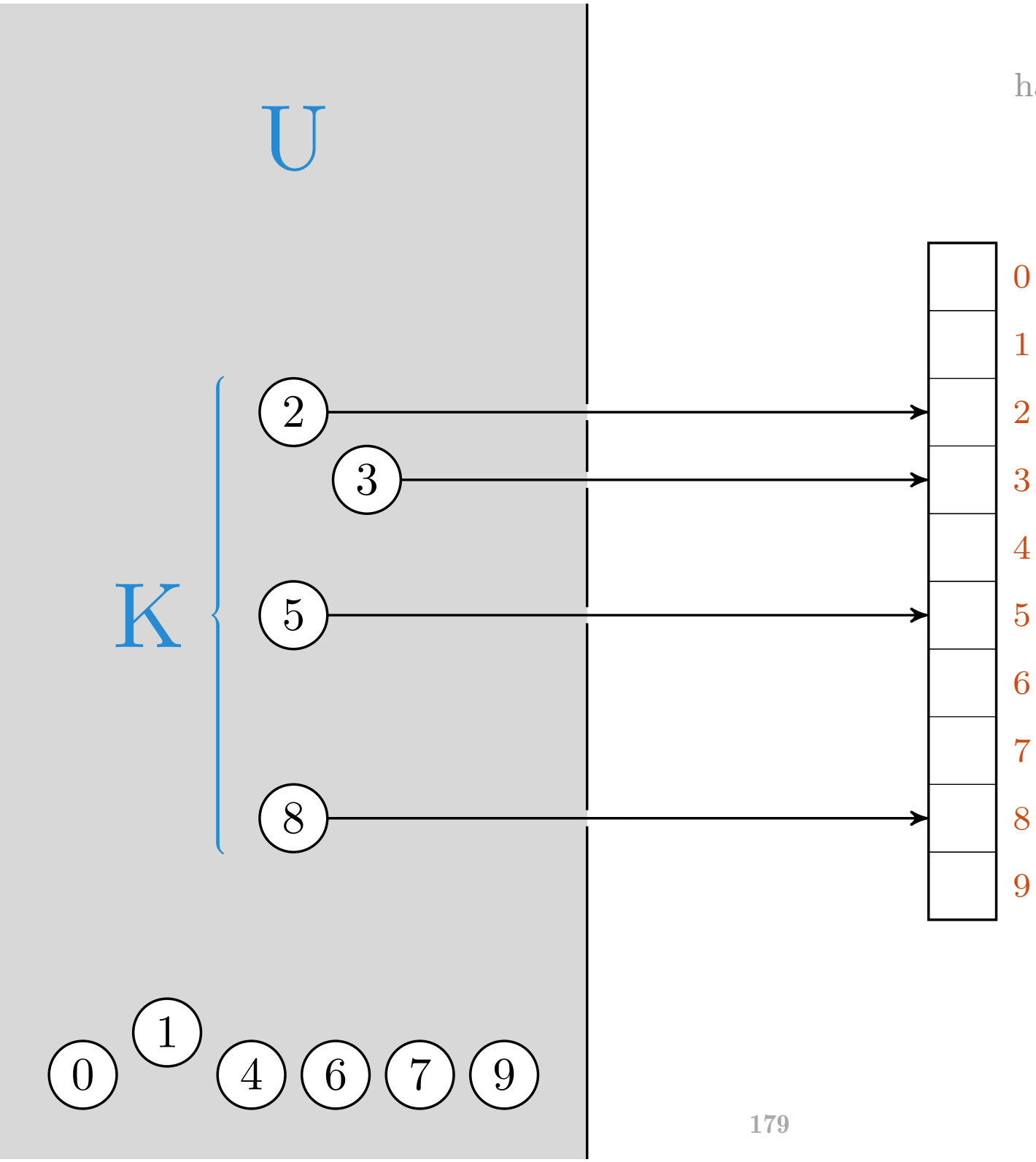


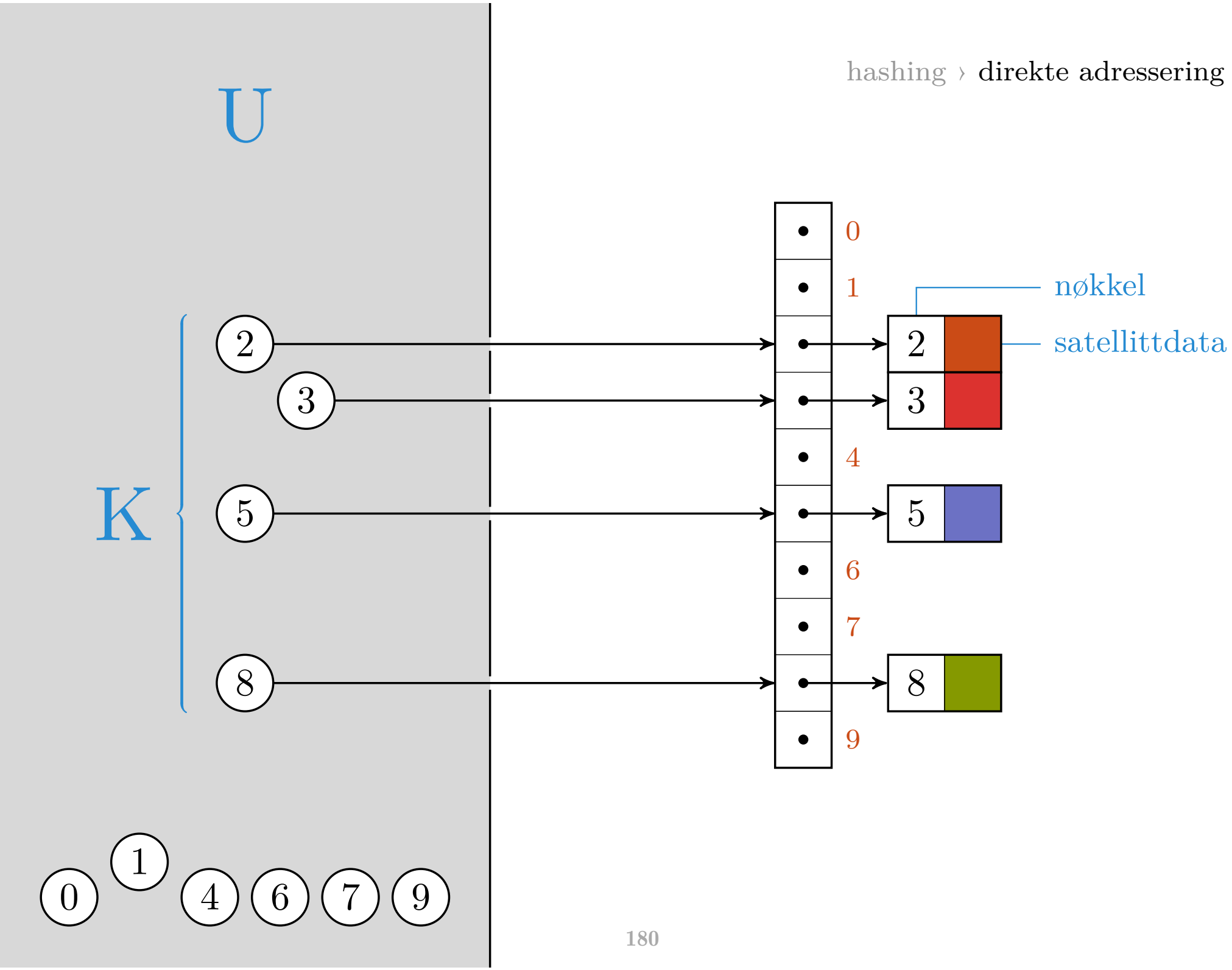


mlh@ntnu.no









DIRECT-ADDRESS-SEARCH(T, k)
1 **return** T[k]

T[k] inneholder x, der $x.key = k$ . . .

DIRECT-ADDRESS-SEARCH(T, k)
1 **return** T[k]

DIRECT-ADDRESS-INSERT(T, x)
1 T[$x.key$] = x

T[k] inneholder x, der $x.key = k$ . . .

mlh@ntnu.no

DIRECT-ADDRESS-SEARCH(T, k)
1 **return** T[k]

DIRECT-ADDRESS-INSERT(T, x)
1 T[$x.key$] = x

DIRECT-ADDRESS-DELETE(T, x)
1 T[$x.key$] = NIL

. . . eller ingenting (NIL)

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

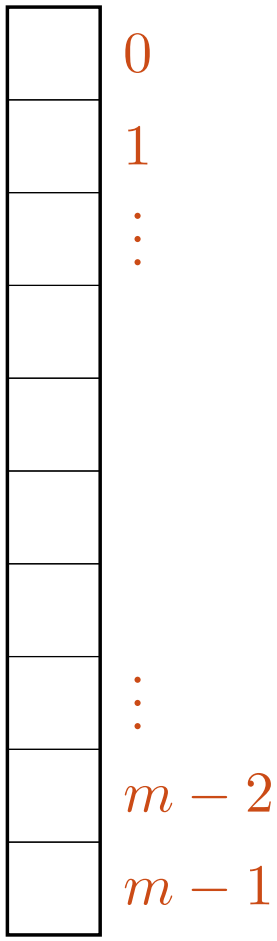
Hashtabeller

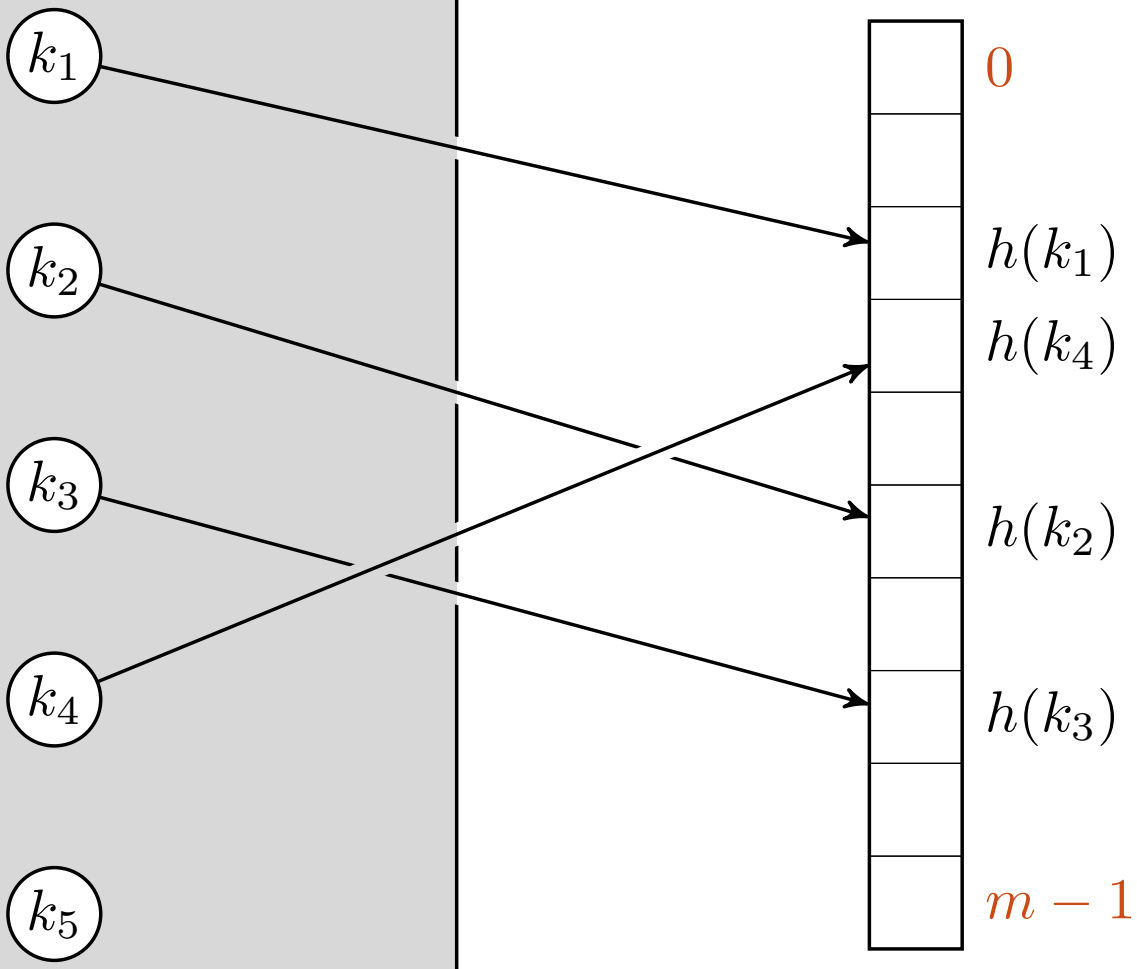
Modifisert nøkkel er indeks

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.

Fortsatt 0-indeksering.





Hashing: Regn ut en indeks fra nøkkelverdien.

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

mlh@ntnu.no

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

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

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

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

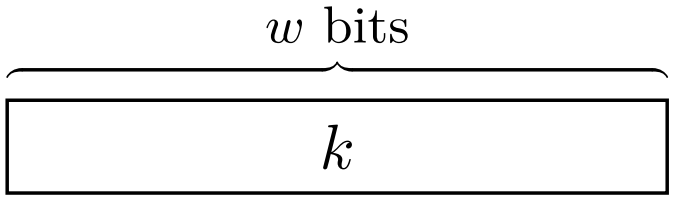
$$h(k) = \lfloor m(k\mathrm{A} \bmod 1) \rfloor$$

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

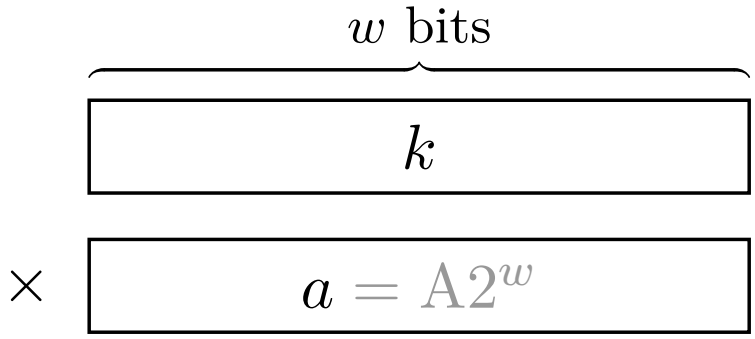
f.eks.

Bedre:
Multipliser-og-skift

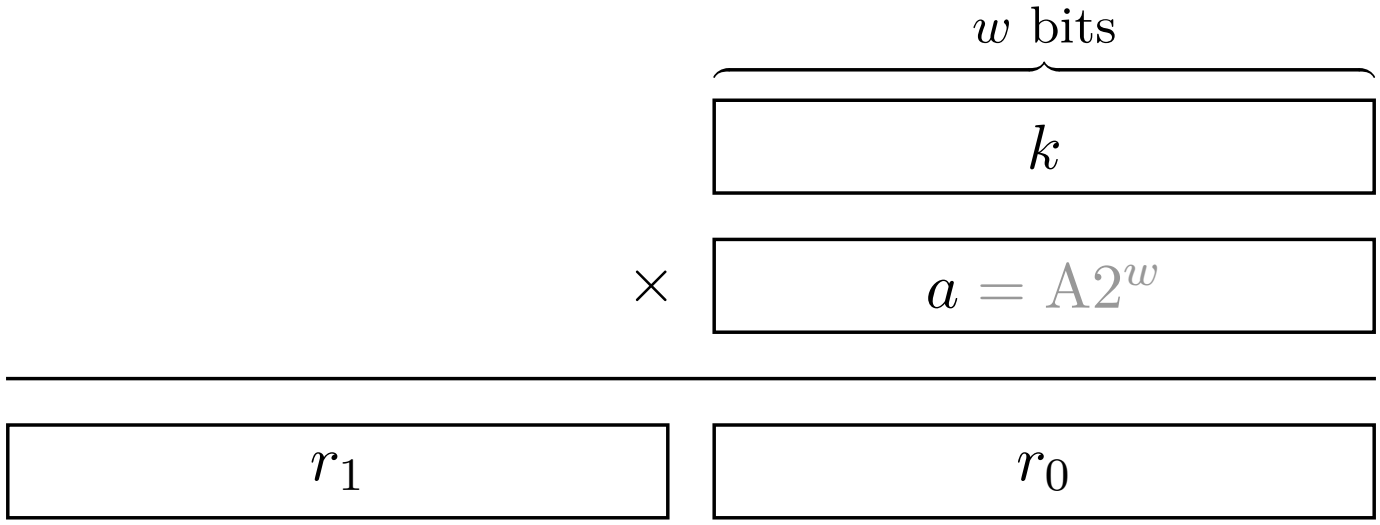
$$h_a(k) = \quad k$$



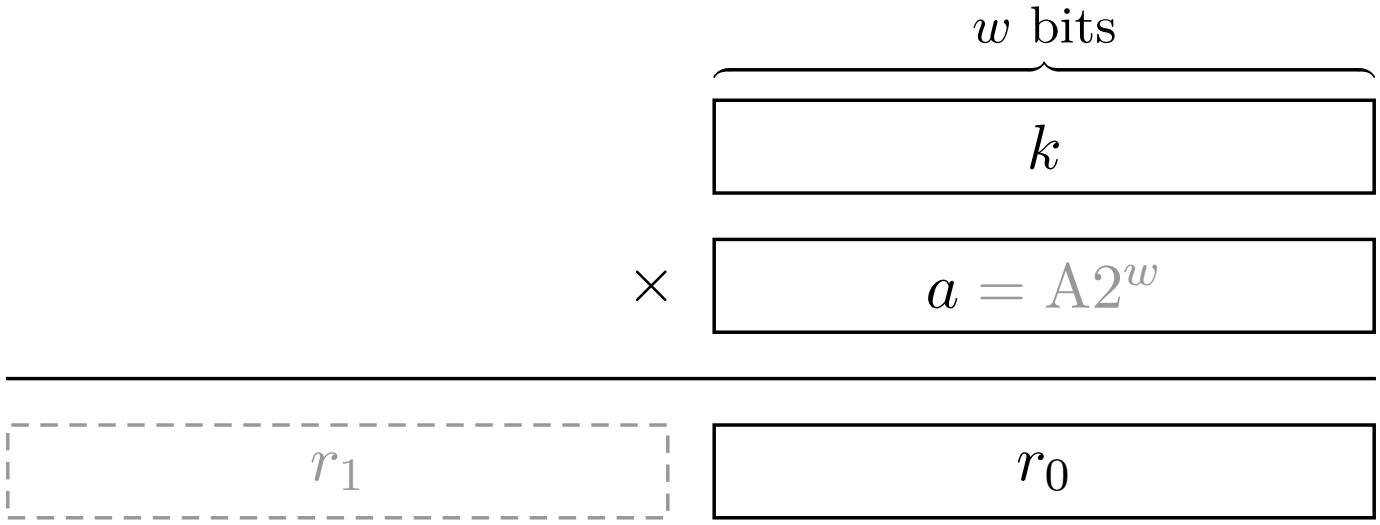
$$h_a(k) = \quad k$$



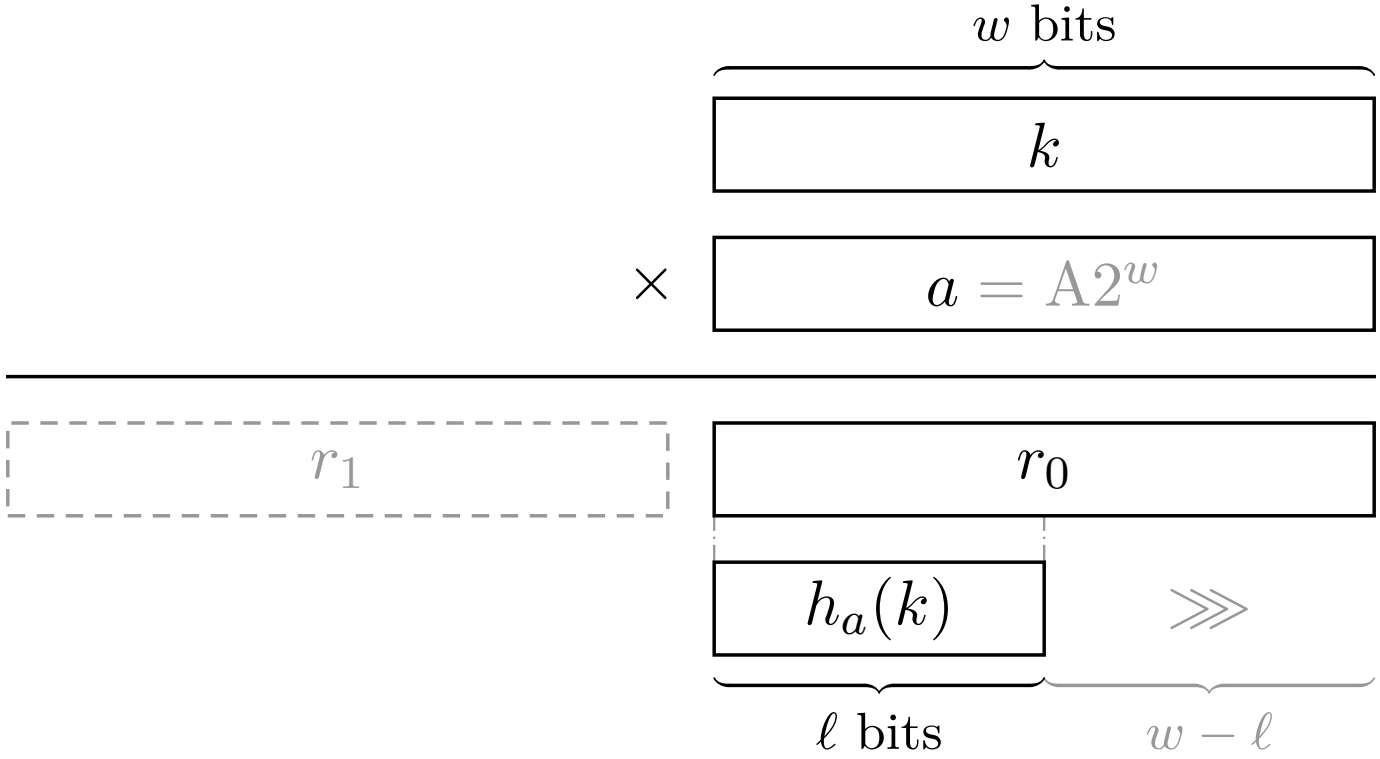
$$h_a(k) = \quad ka$$



$$h_a(k) = \quad ka$$



$$h_a(k) = (ka \bmod 2^w)$$



$$h_a(k) = (ka \bmod 2^w) \ggg (w - \ell)$$

Her kan man også f.eks. bruke kryptografisk hashing som en start. (Faller utenfor pensum.)

Strenger e.l.?

Regn ut et heltall

F.eks. behandle tegn som siffer

Vi kan lage hashfunksjoner for f.eks. strenger av variabel lengde.

F.eks.: Bruk aritmetiske operasjoner til å kombinere tegn eller ord-lengde-segmenter.

$$\sum_{i=1}^{n} \mathrm{A}[i] \cdot p^{i-1} \mod m$$

«Polynomial rolling hash»

mlh@ntnu.no

$$\sum_{i=1}^{n} \mathrm{A}[i] \cdot p^{i-1} \mod m$$

(Ikke i boka, men se oppg. 11.3-6)

mlh@ntnu.no

$$\sum_{i=1}^{n} \mathrm{A}[i] \cdot p^{i-1} \mod m$$

Her finnes det mange alternativer!

mlh@ntnu.no

Det følgende er ikke en pensumalgoritme; bare et eksempel på hash-utregning for en streng.

ROLLING-HASH(A, p, n, m)

```
1  h = 0
2  q = 1
3  for i = 1 to n
4        h = h + A[i] · q  mod m
5        q = q · p  mod m
6  return h
```

$h, q, m = -, -, 1024$

65	1
76	2
71	3
68	4
65	5
84	6

“ALGDAT” i ASCII/UTF-8

mlh@ntnu.no

ROLLING-HASH(A, p, n, m)

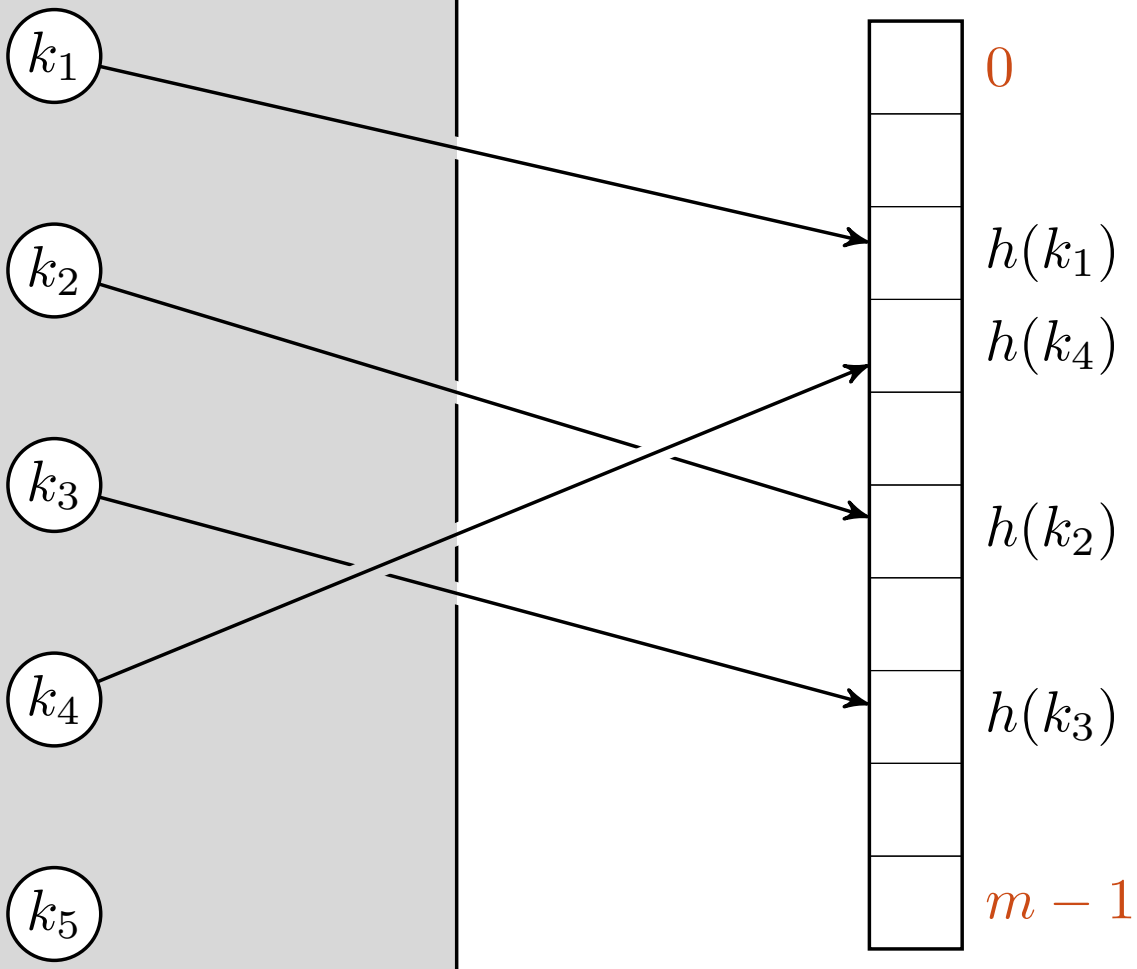
```
1  h = 0
2  q = 1
3  for i = 1 to n
4        h = h + A[i] · q  mod m
5        q = q · p  mod m
6  return h
```

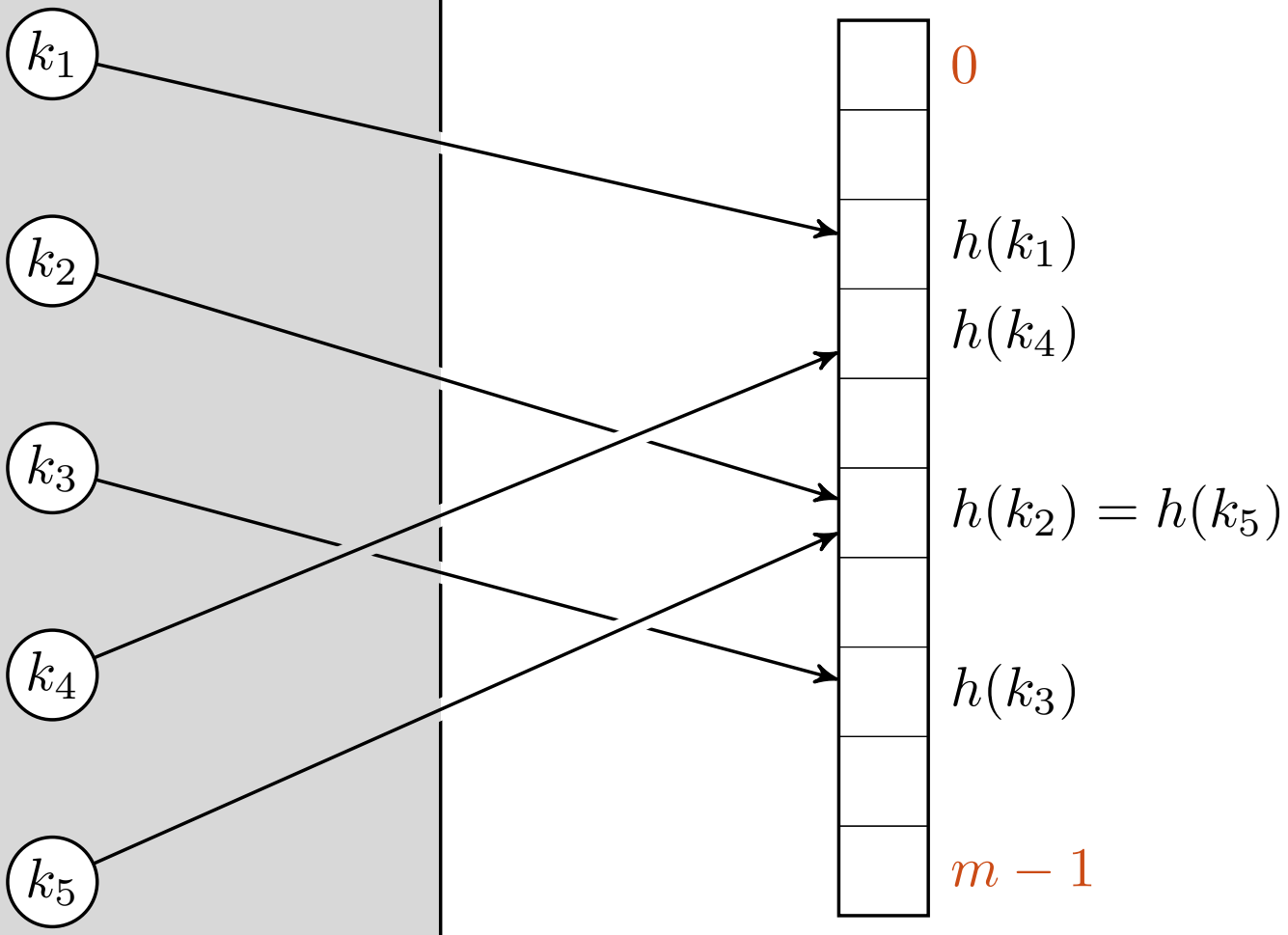
$\rightarrow$ 293

$h, q, m \;=\; 293, 833, 1024$

65	1
76	2
71	3
68	4
65	5
84	6

Men ...

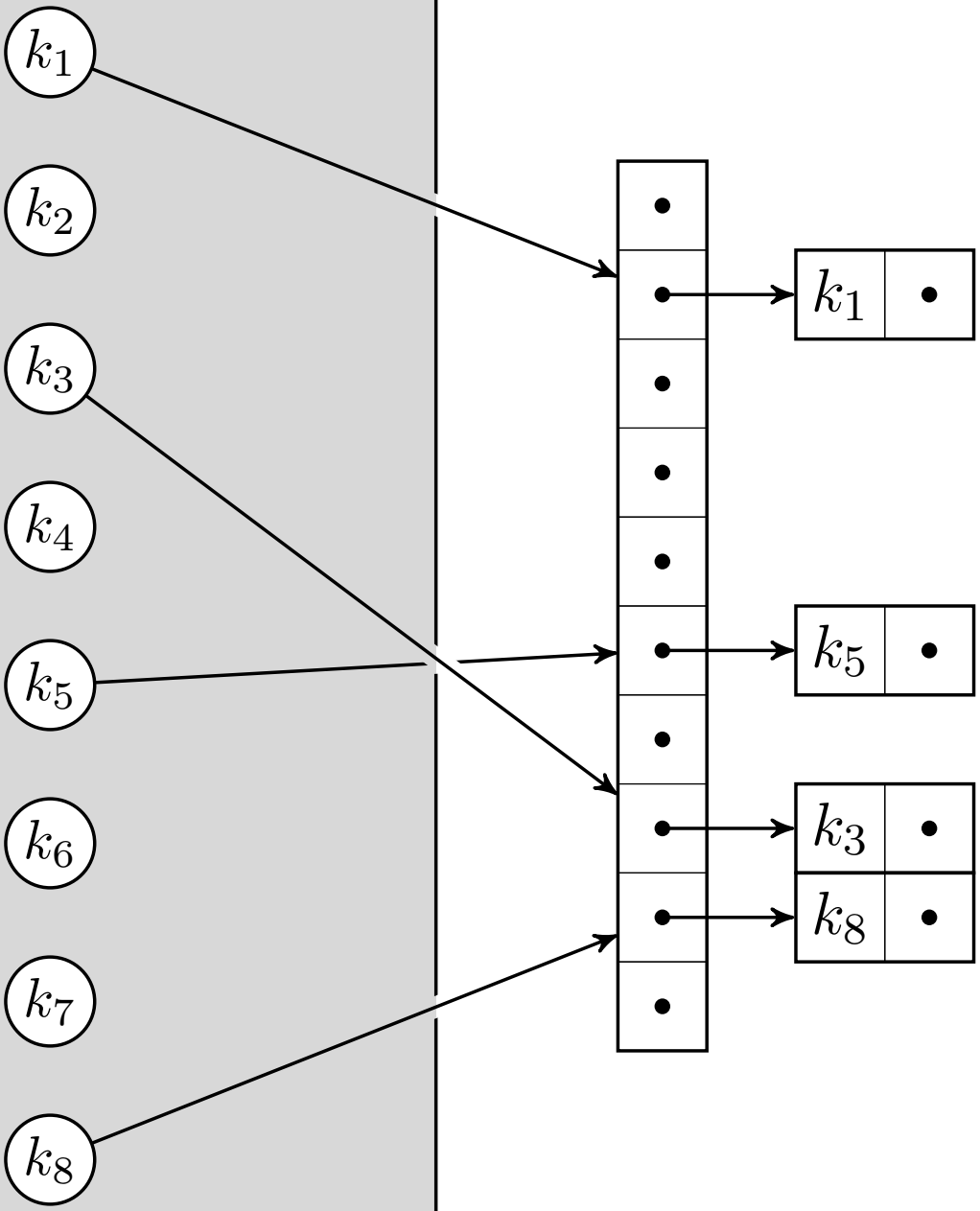


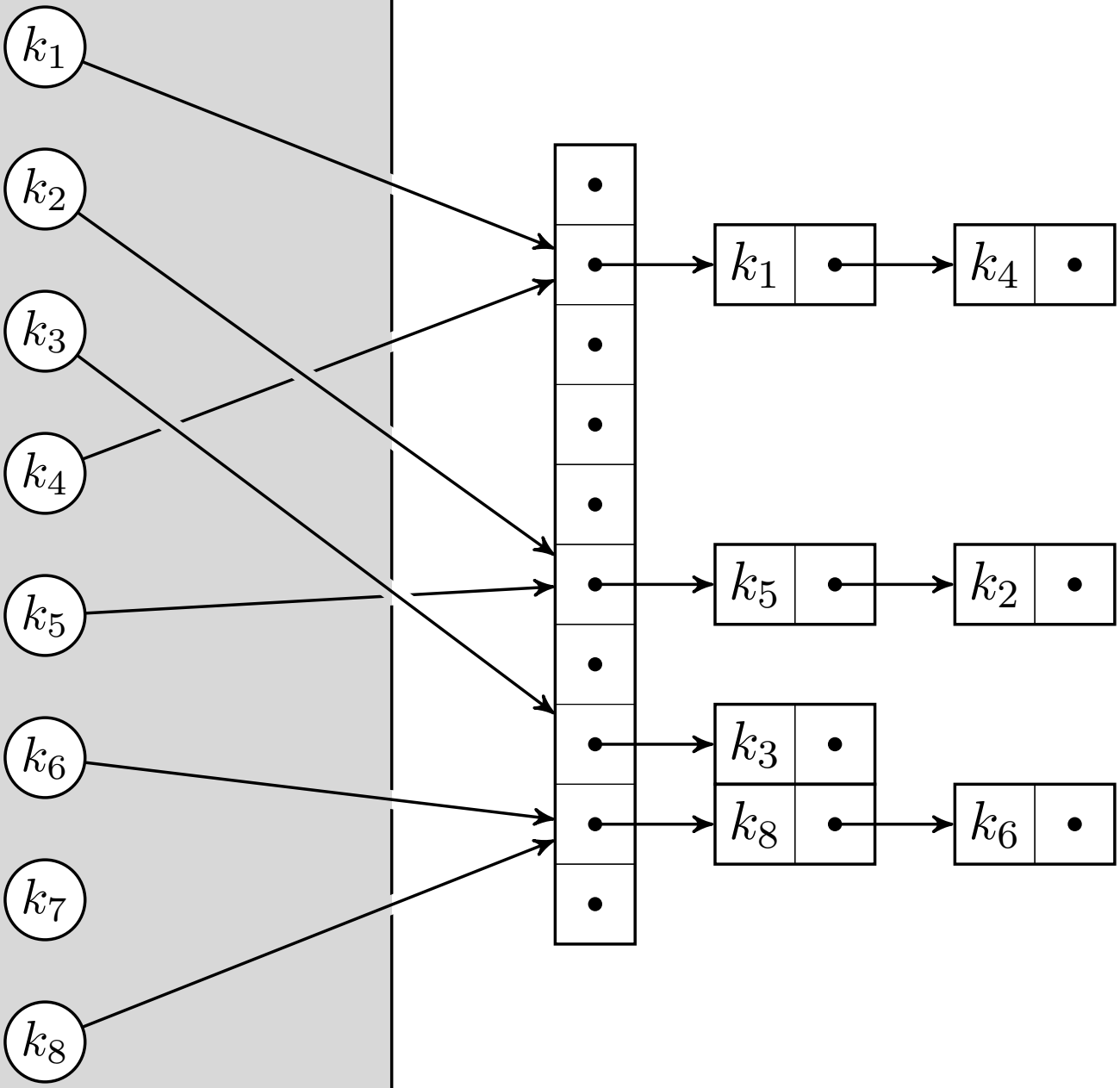


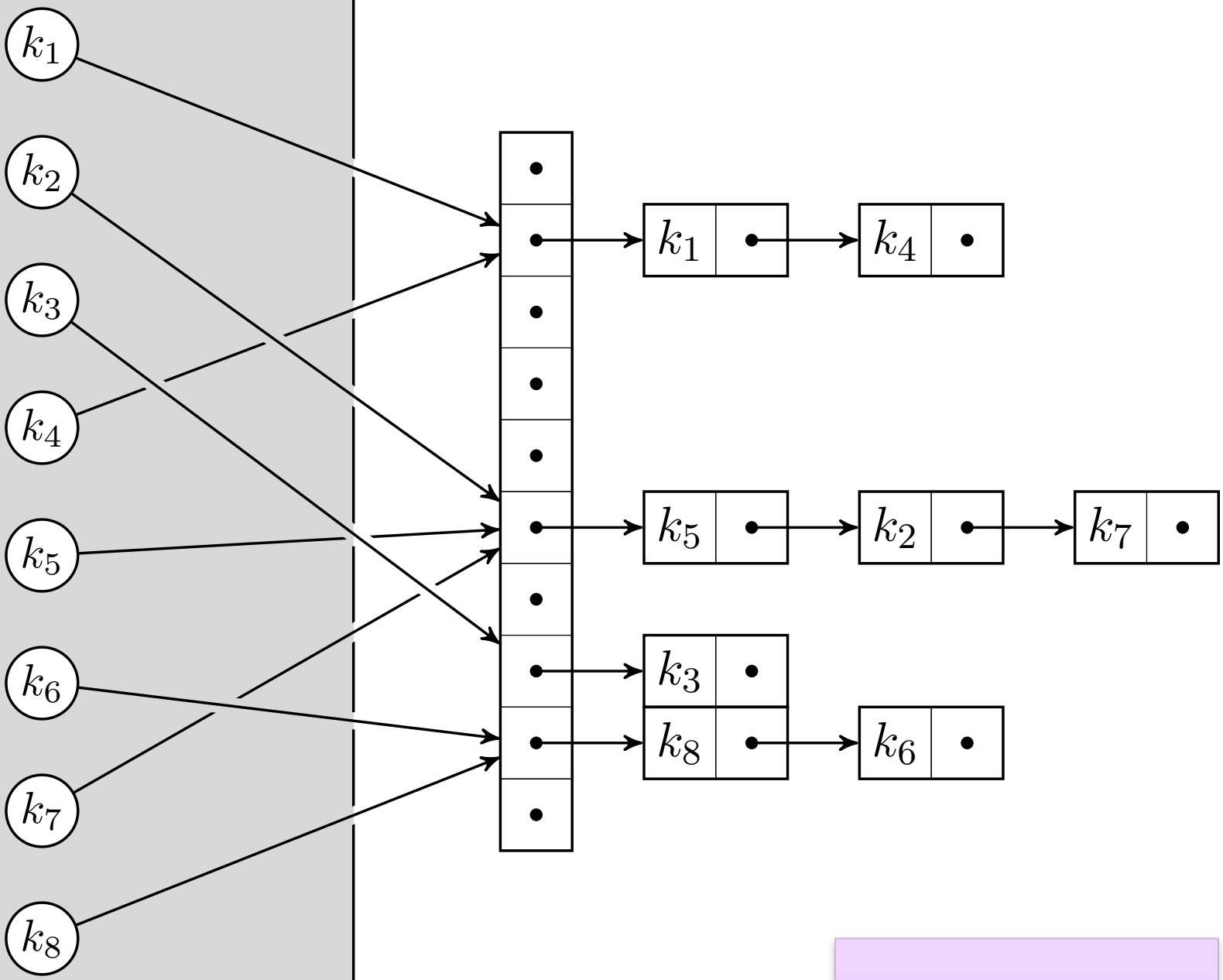
Kjeding (chaining)

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.







Her har vi trolig også satellittdata i tillegg til nøklene.

```
CHAINED-HASH-SEARCH(T, k)
1  return LIST-SEARCH(T[h(k)], k)
```

```
CHAINED-HASH-SEARCH(T, k)
1  return LIST-SEARCH(T[h(k)], k)

CHAINED-HASH-INSERT(T, x)
1  LIST-PREPEND(T[h(x.key)], x)
```

```
CHAINED-HASH-SEARCH(T, k)
1  return LIST-SEARCH(T[h(k)], k)

CHAINED-HASH-INSERT(T, x)
1  LIST-PREPEND(T[h(x.key)], x)

CHAINED-HASH-DELETE(T, x)
1  LIST-DELETE(T[h(x.key)], x)
```


mlh@ntnu.no

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

Statisk datasett?

Skreddersydd hashfunksjon!

F.eks. såkalt «perfekt hashing», der vi unngår alle kollisjoner.

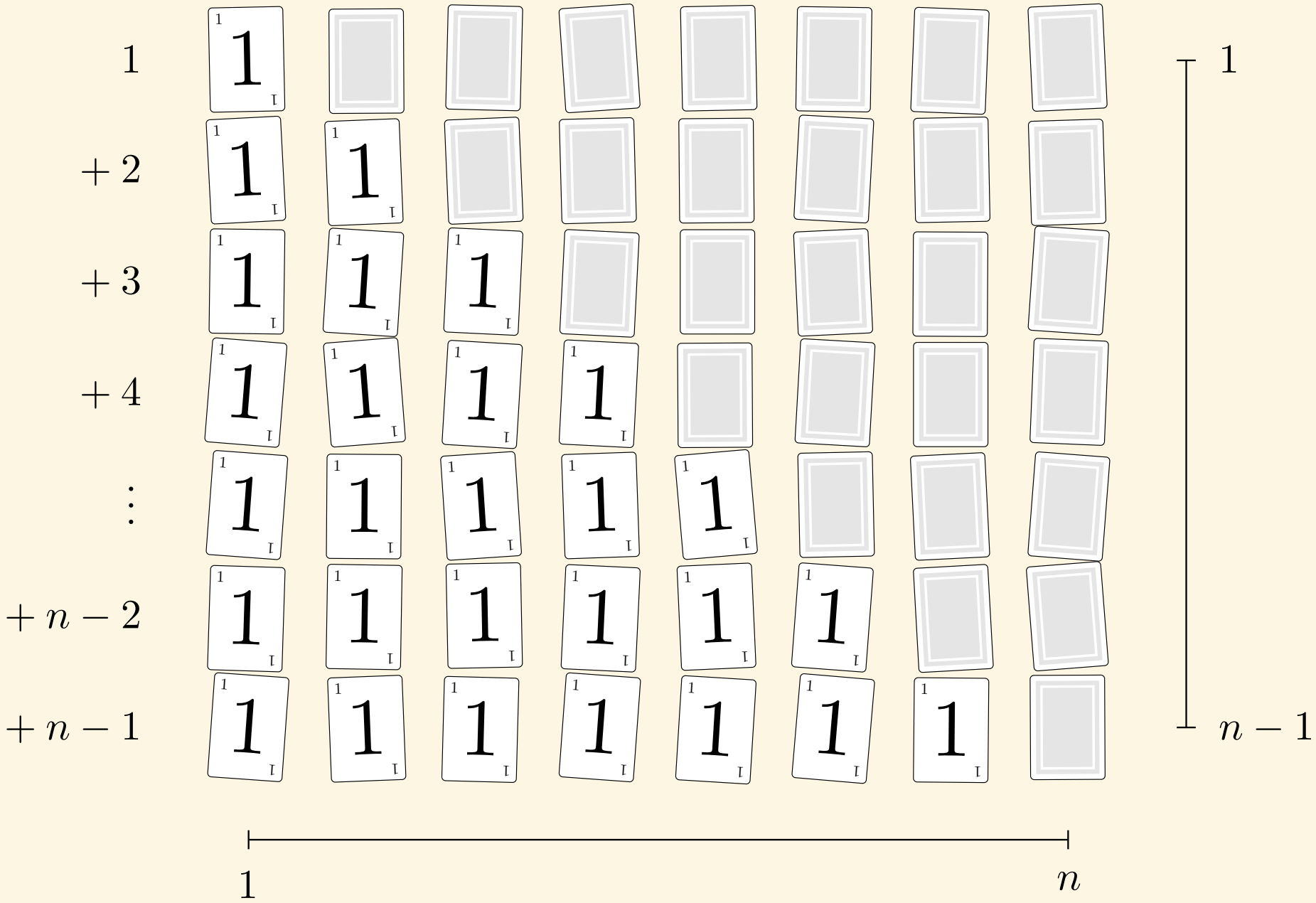
Kan da garantere konstant kjøretid

To kritiske summer

Mellomspill/repetisjon

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

mlh@ntnu.no



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

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

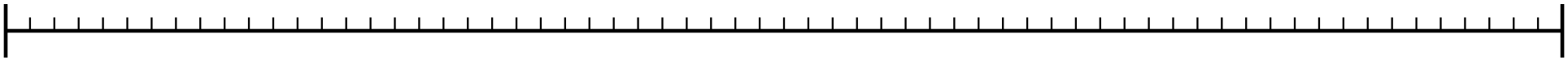
antall $\times$ gjennomsnitt av første og siste

mlh@ntnu.no

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

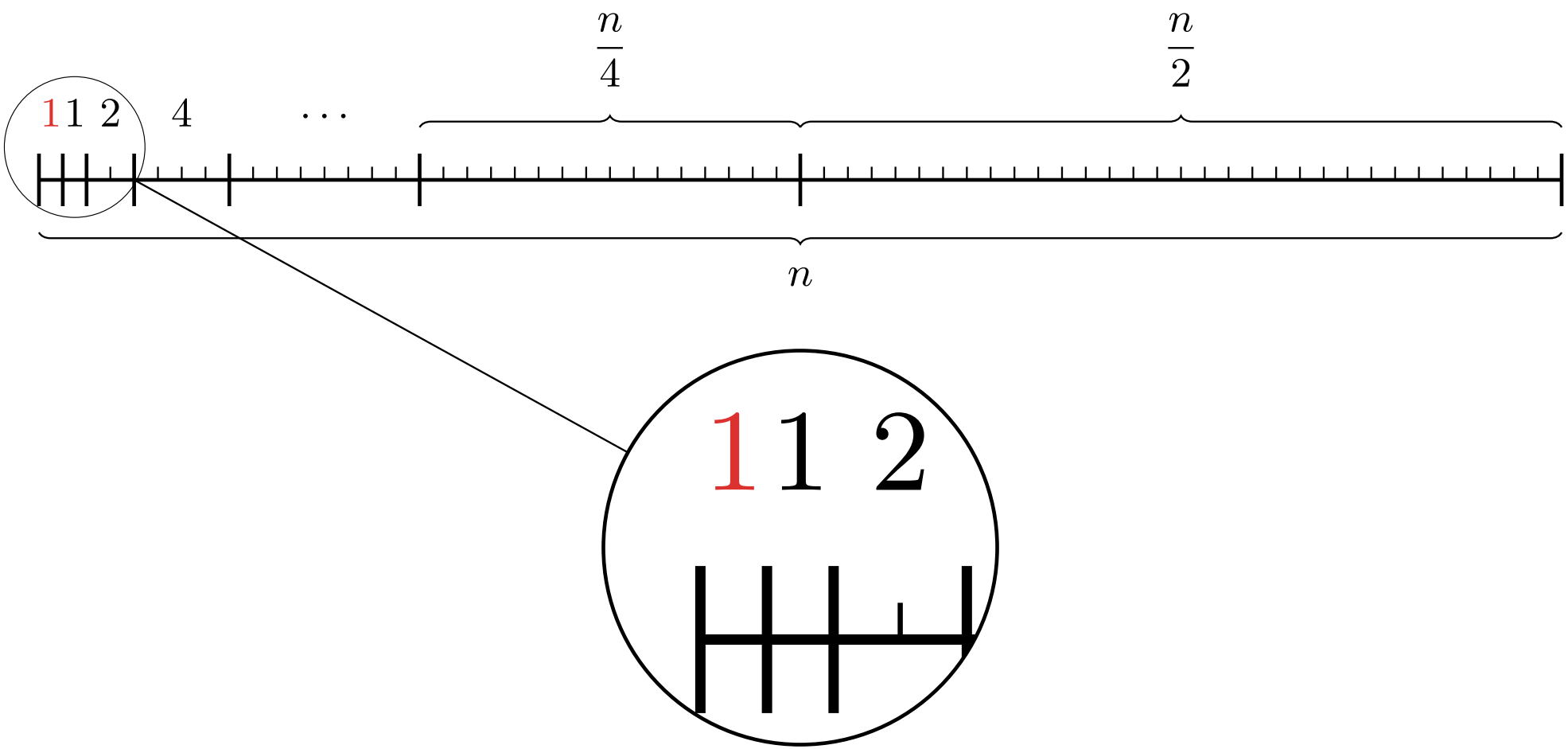
mlh@ntnu.no

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.



Alternativ huskeregel 1: Utslagsturnering med n deltakere (jf. Oppgave i forelesning 1).

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.

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

5:5

Dynamiske tabeller

Amortisert arbeid

- Kjøretid for én enkelt operasjon: Ikke alltid informativt
- Se på gjennomsnitt per operasjon etter at mange har blitt utført!
- Aggregert analyse: Finn totalt arbeid og del på antall operasjoner

Dette får vi bruk for f.eks. i DFS, BFS, Dag-Shortest-Paths, og Prims og Dijkstras algoritmer, der en indre løkke potensielt kan ta lineær tid, men totalen for alle kjøringene er lineær.

Eksempel: Multipop

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2        POP(S)
3        k = k − 1
```

La oss si vi utfører n PUSH og MULTIPOP-operasjoner

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2        POP(S)
3        k = k − 1
```

I verste fall er da kjøretiden for MULTIPOP $\Theta(n)$

mlh@ntnu.no

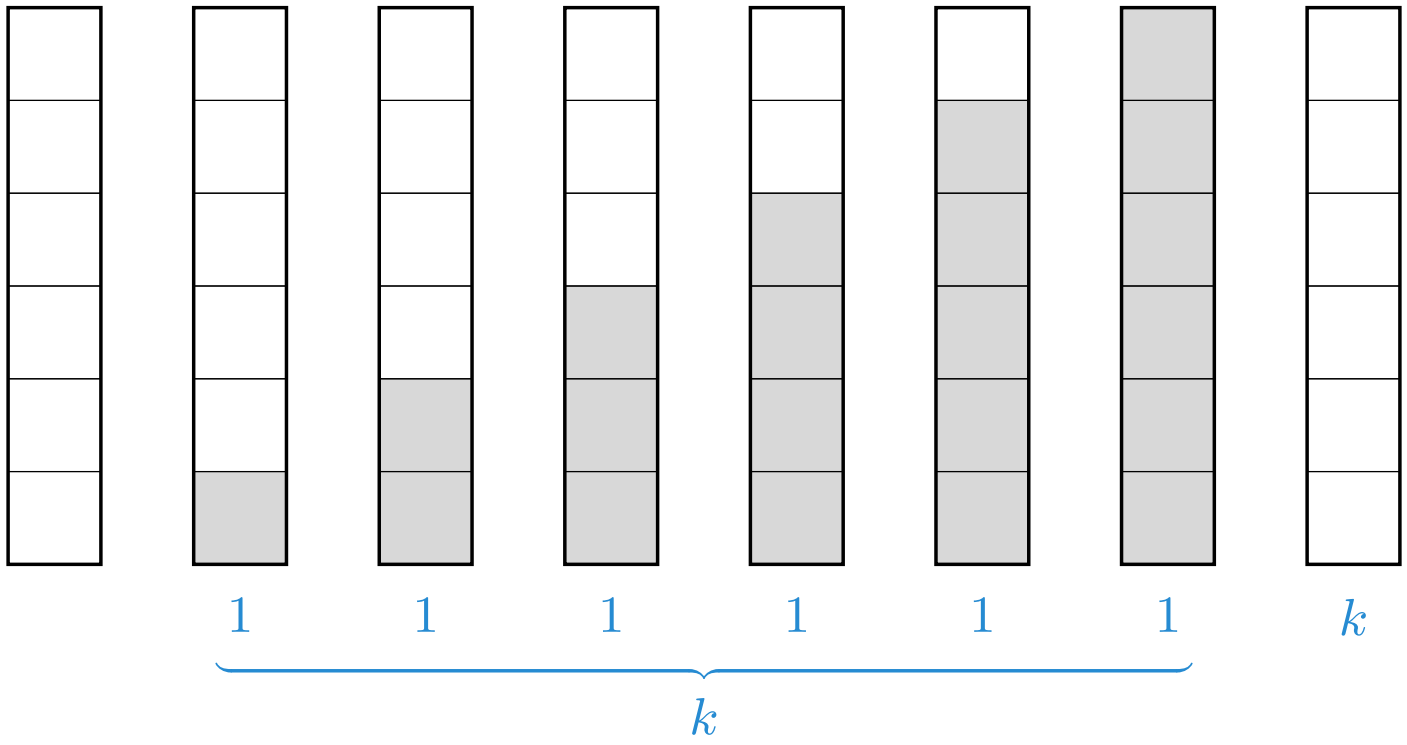
```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2        POP(S)
3        k = k − 1
```

Vi har n operasjoner, hver med kjøretid $O(n)$: Totalt, $O(n^2)$

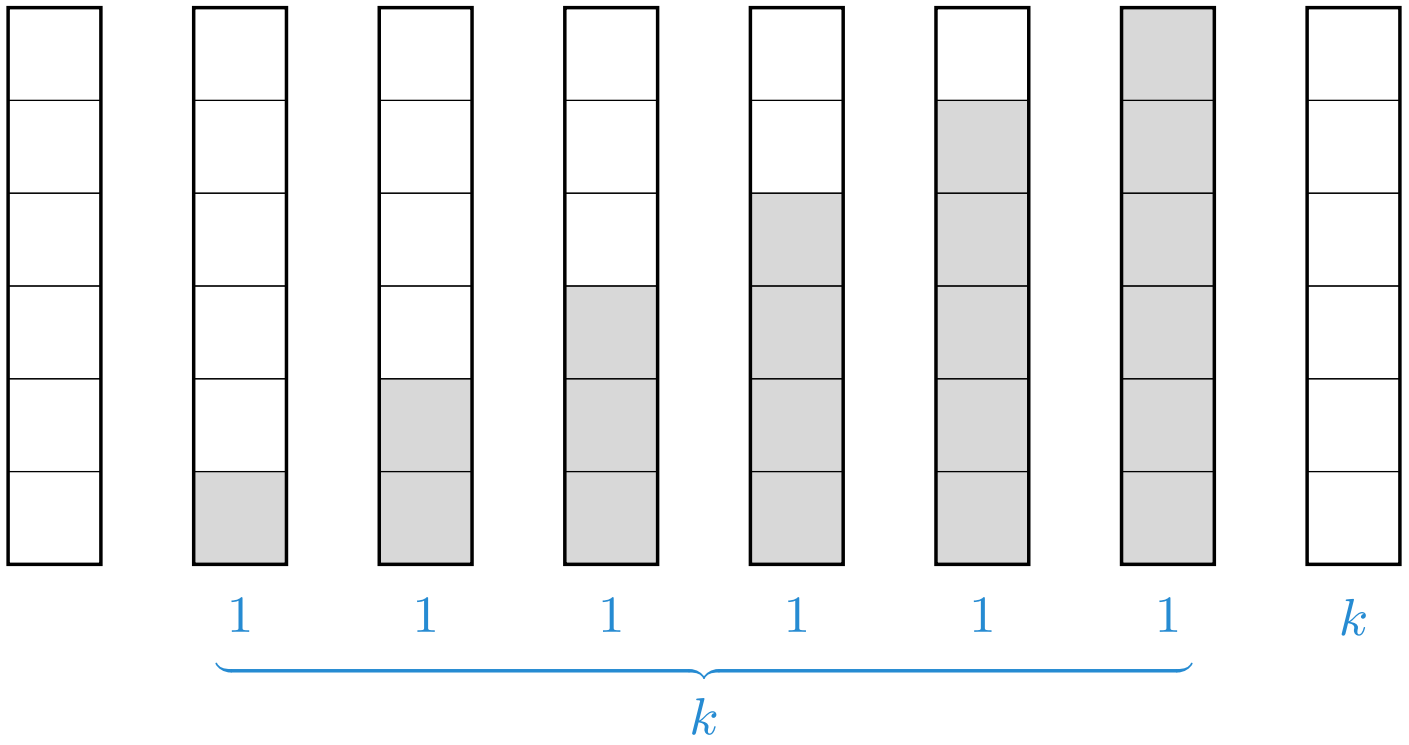
mlh@ntnu.no

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2        POP(S)
3        k = k − 1
```

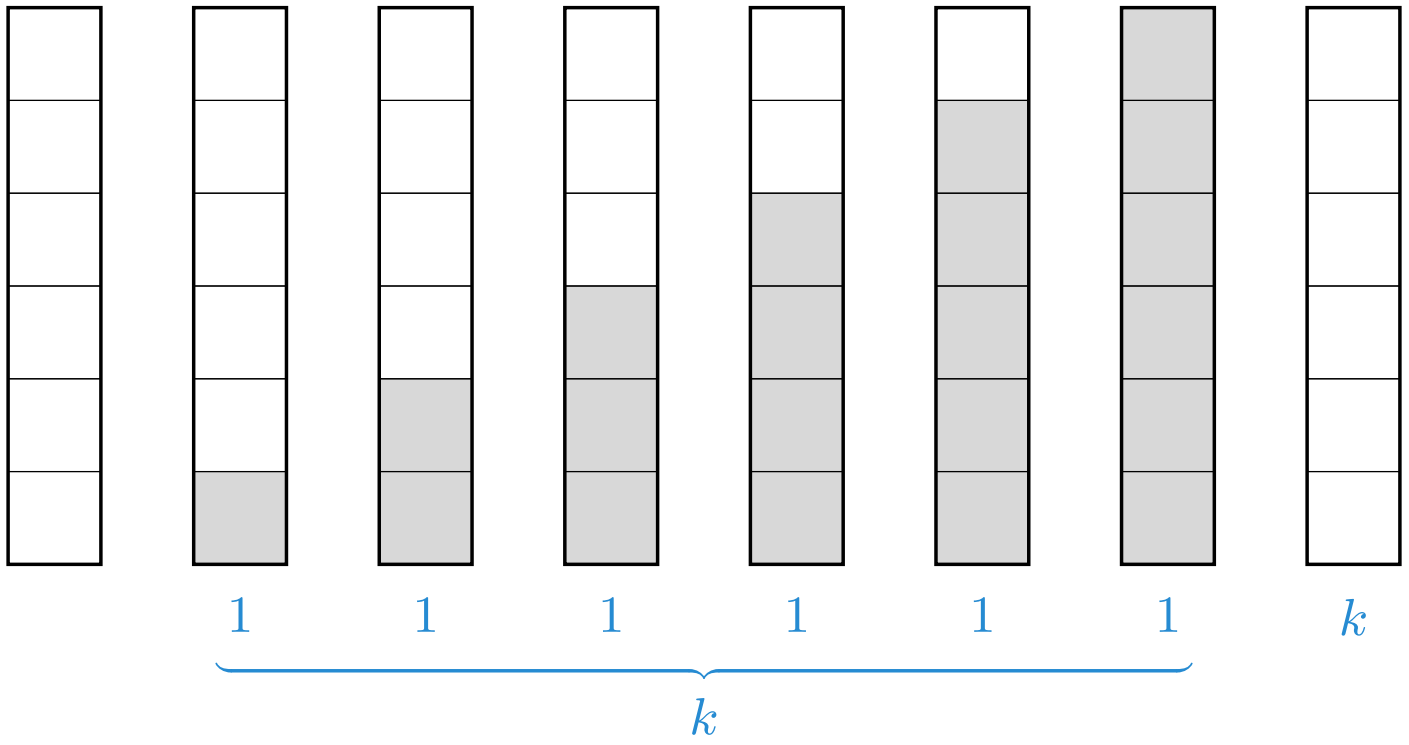
Men dette er upresist. Total kjøretid er *ikke* $\Theta(n^2)$



Før hver MULTIPOP(S, k) må vi minst ha $k \times$ PUSH(S, x)



Total (aggregert) kjøretid for n operasjoner er $\Theta(n)$



$\Theta(n)$ fordelt på n operasjoner er $\Theta(1)$ i snitt (amortisert)

Men tilbake til de dynamiske tabellene våre ...

- **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**

Tabelldobling

TABLE-INSERT(T, x)

T	«tabell»
x	verdi

Tilsvarer PUSH, men utvider T om nødvendig

```
TABLE-INSERT(T, x)
 1  if T.size == 0
```

T	«tabell»
x	verdi
T.*size*	kapasitet

Ingen kapasitet!

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell

Alloker ny intern tabell av lengde 1

mlh@ntnu.no

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1:1]
 3        T.size = 1
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell

Husk at vi har kapasitet 1

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1:1]
 3        T.size = 1
 4  if T.num == T.size
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.

Tabellen er full!

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.
A	midl. tabell

Ny tabell, dobbelt så stor

mlh@ntnu.no

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.
A	midl. tabell

Dette tar lineær tid!

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
 7        free T.table
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.
A	midl. tabell

Trenger ikke denne lenger

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
 7        free T.table
 8        T.table = A
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.
A	midl. tabell

Oppdater til å bruke den nye tabellen

mlh@ntnu.no

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
 7        free T.table
 8        T.table = A
 9        T.size = 2 · T.size
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.
A	midl. tabell

Husk at vi har doblet kapasiteten

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
 7        free T.table
 8        T.table = A
 9        T.size = 2 · T.size
10  insert x into T.table
```

T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.
A	midl. tabell

Altså $\mathrm{T}.table[\mathrm{T}.num + 1] = x$

mlh@ntnu.no

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
 7        free T.table
 8        T.table = A
 9        T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

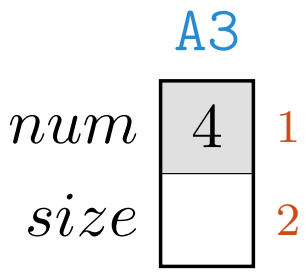
T	«tabell»
x	verdi
T.*size*	kapasitet
T.*table*	intern tabell
T.*num*	ant. elt.
A	midl. tabell

Husk at vi har lagt til et element

TABLE-INSERT(T, x)

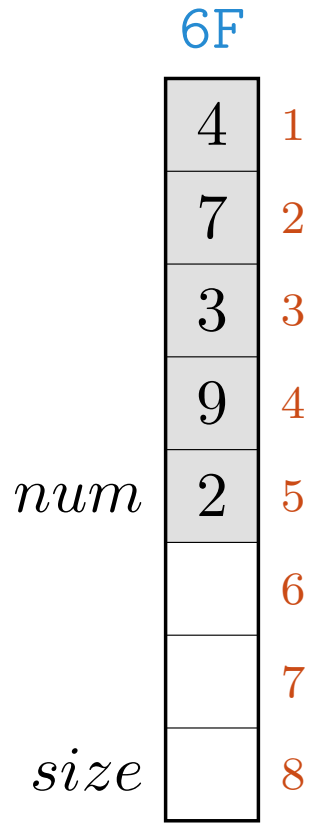
```
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
 7        free T.table
 8        T.table = A
 9        T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

T.*table*, x = A3, 7



```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        alloc T.table[1 : 1]
 3        T.size = 1
 4  if T.num == T.size
 5        alloc A[1 : 2 · T.size]
 6        copy T.table into A
 7        free T.table
 8        T.table = A
 9        T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

T.*table*, x = 6F, 2



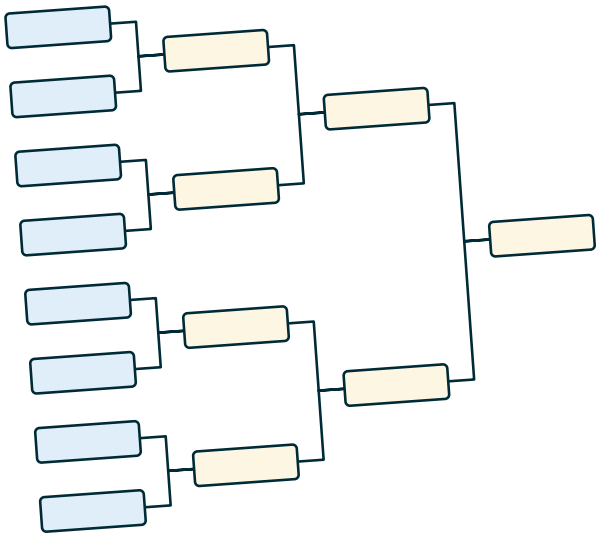
Oppgave

En utslagsturnering med n deltakere har 1 finale, 2 semifinaler, 4 kvartfinaler, etc. Det totale antallet matcher er altså:

$$1 + 2 + 4 + \cdots + n/4 + n/2$$

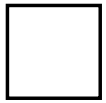
Hva blir summen?

Dette er antall indre noder i et komplett binærtre.

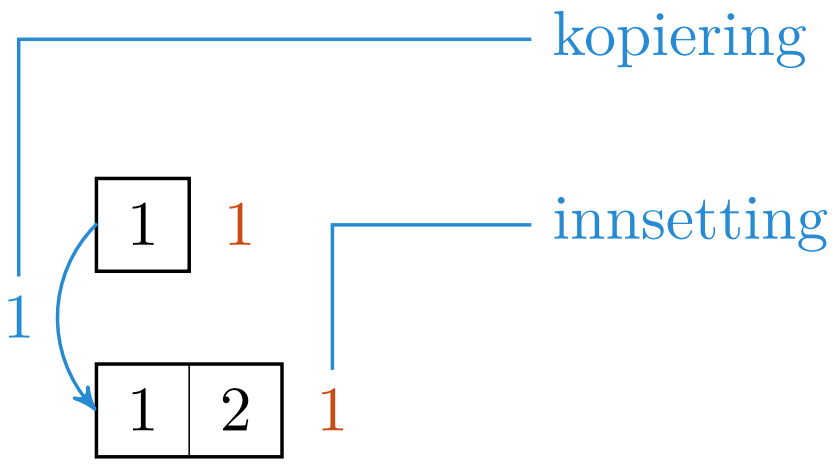


Tenk selv	0:30
Jobb sammen	1:30
Observasjoner	
Løsningsforslag	
Refleksjon	1:00

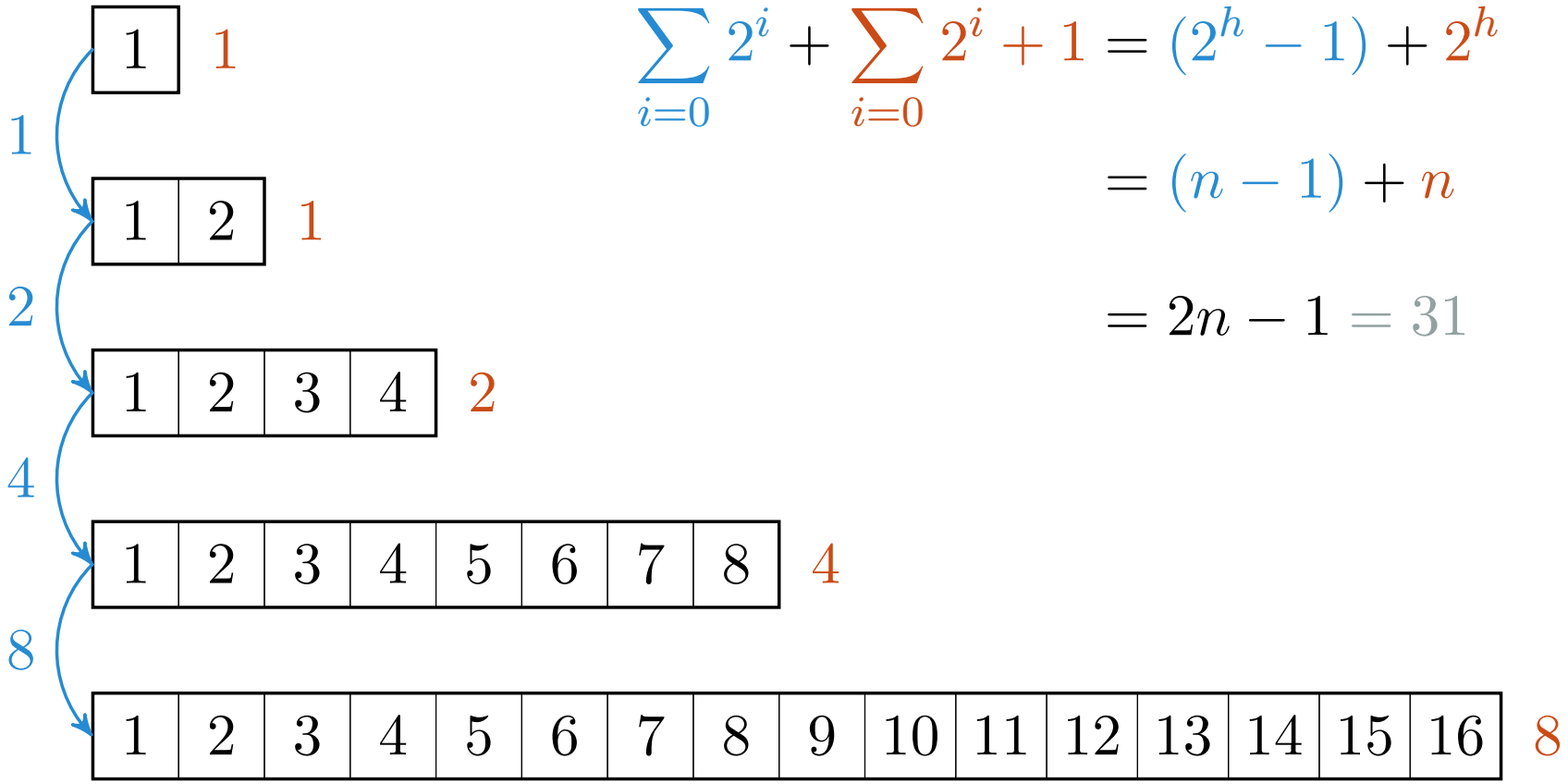
$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$



mlh@ntnu.no



Antar at allokering domineres av innsetting



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

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

$$= 2n - 1 = 31$$

$$h = \log_2 n = 4$$

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

Hashing:

Ny tabell gir annen ny hashfunksjon.

Bare sett inn elementene på nytt!

Snitt-kjøretid 1 & 2

Avg-case og amortisering

Average-case: Forventet kjøretid, hvis vi ser på en sannsynlighetsfordeling (vanligvis uniform) over instanser/inputs. (Vi bruker vanligvis uttrykket «forventet kjøretid» om randomiserte algoritmer.)

Avg-case

Snitt over instanser

Amortisering

Snitt over operasjoner


GO,
YOU!