



INF2220: Forelesning 3

Map og hashing

- Abstrakte datatyper (kapittel 3.1)
- Map (kapittel 4.8)
- Hashing (kapittel 5)



REPETISJON: ALGORITMER OG STOR O



```
long power(long x, long n)
  if (n == 0)
    return 1;
  else
    return x * power(x, n-1);
```

$$T(0) = c_1 \quad \text{for some constant } c_1$$
$$T(n) = c_2 + T(n - 1) \quad \text{for some constant } c_2$$



```
long power(long x, long n)
  if (n==0) return 1;
  if (n==1) return x;
  if ((n % 2) == 0)
    return power(x*x, n/2);
  else
    return power(x*x, n/2) * x;
```

$$T(0) = c_1$$

$$T(1) = c_2$$

$$T(n) = T(n/2) + c_3$$

(Assume n is a power of 2)



```
long power(long x, long n)
  if (n==0) return 1;
  if (n==1) return x;
  if ((n % 2) == 0)
    return power(x,n/2) * power(x,n/2);
  else
    return power(x,n.2) * power(x,n/2) * x;
```

$$T(0) = c_1$$

$$T(1) = c_2$$

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + c_3 \\ &= 2T(n/2) + c_3 \end{aligned}$$

(Again, assume n is a power of 2)



```
int Fibonacci(int n)
{
    if(n == 1 || n == 2)
        return 1;

    return (Fibonacci(n - 1) + Fibonacci(n - 2));
}
```

$$T(n \leq 1) = O(1)$$

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



REPETISJON RØD-SVARTE TRÆR



Rød-svarte trær

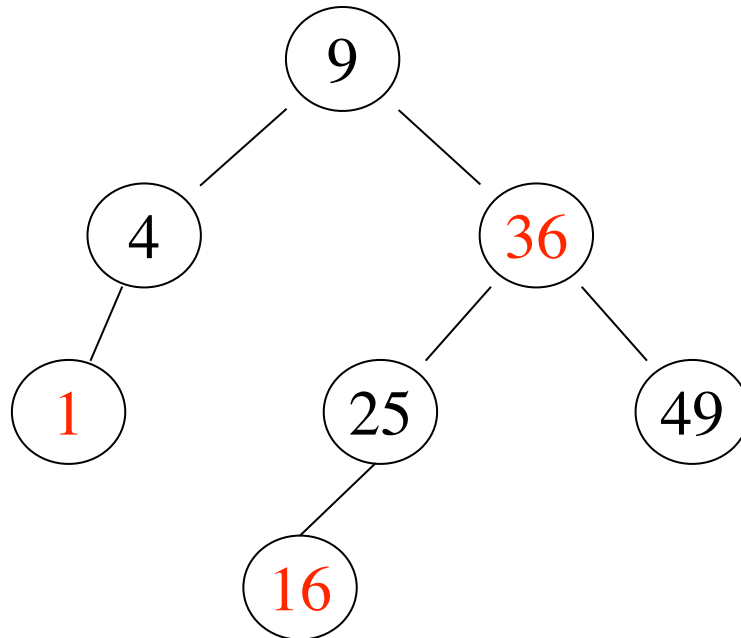
Et rød-svart tre er et binært søketre der hver node er farget enten rød eller svart slik at:

1. Roten er svart.
2. Hvis en node er rød, må barna være svarte.
3. Enhver vei fra en node til en null-peker må inneholde samme antall svarte noder.

Disse reglene sikrer at høyden på et rød-svart tre er maksimalt $2 \cdot \log_2(N+1)$!

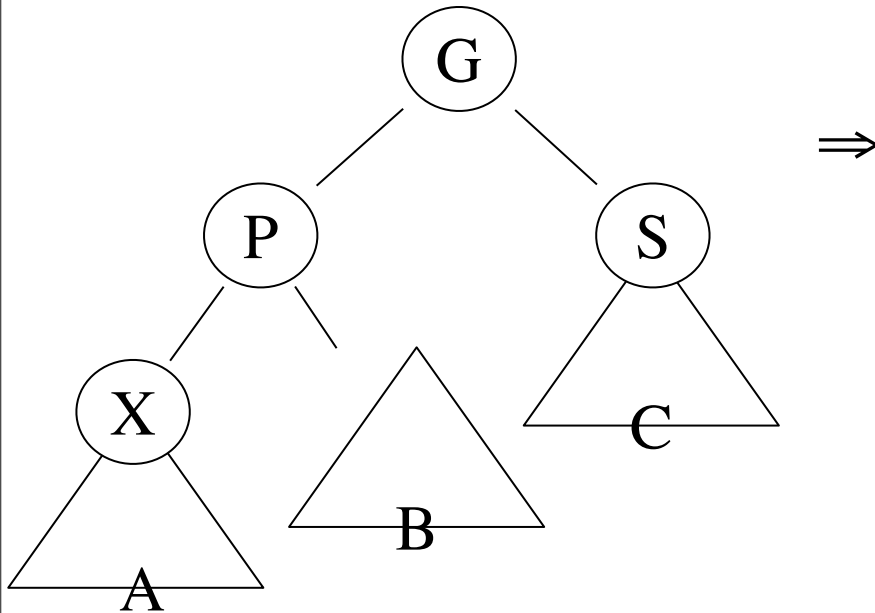
Oppgave

- Farg nodene i følgende tre slik at det blir et rød-svart tre:



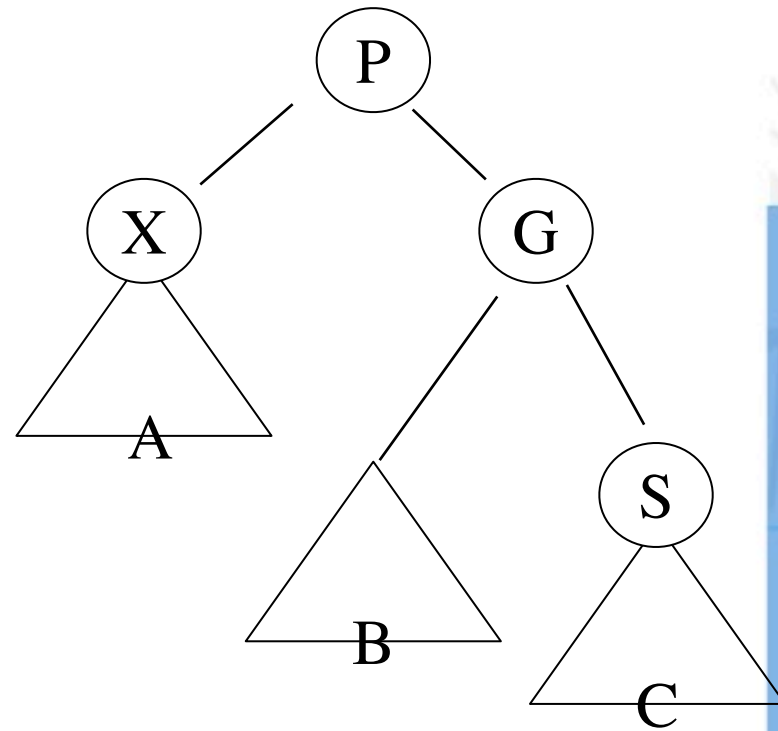
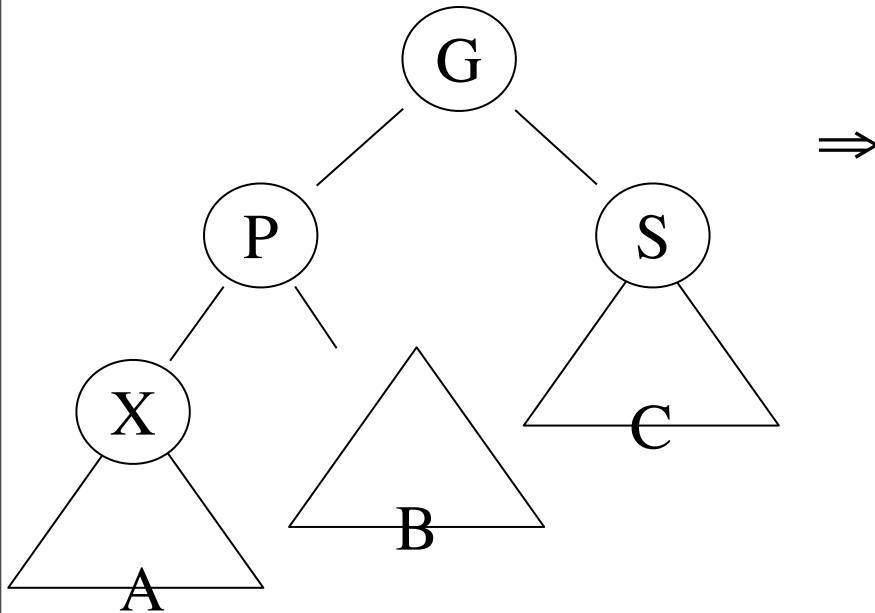
- Sett inn tallet 64 på riktig plass og med riktig farge.
- Forsøk så å sette inn tallet 81.

Zig rotasjon



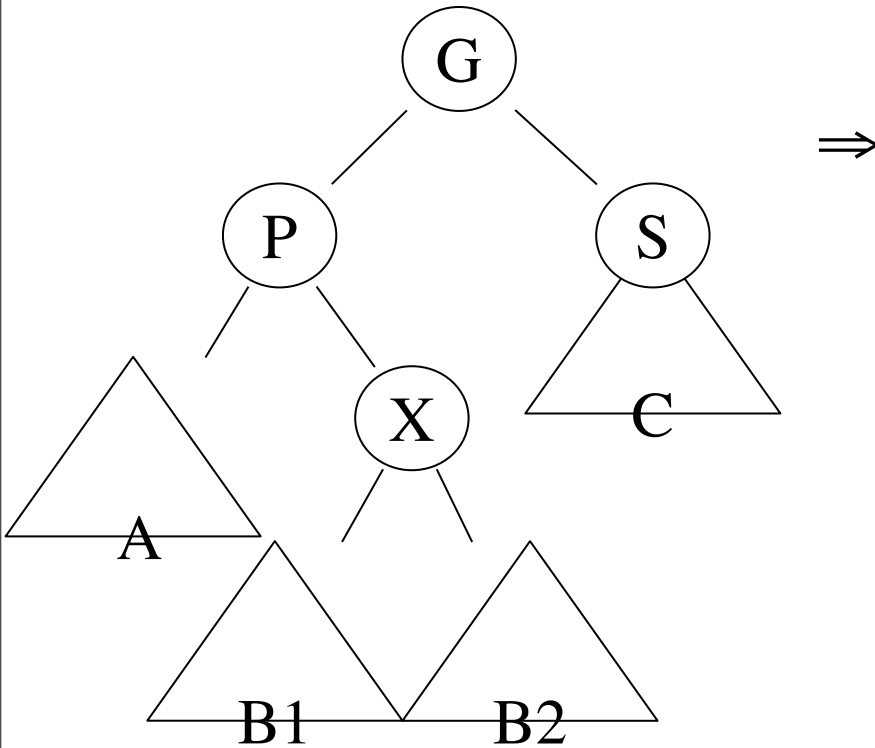
+ symmetrisk tilfelle...

Zig rotasjon



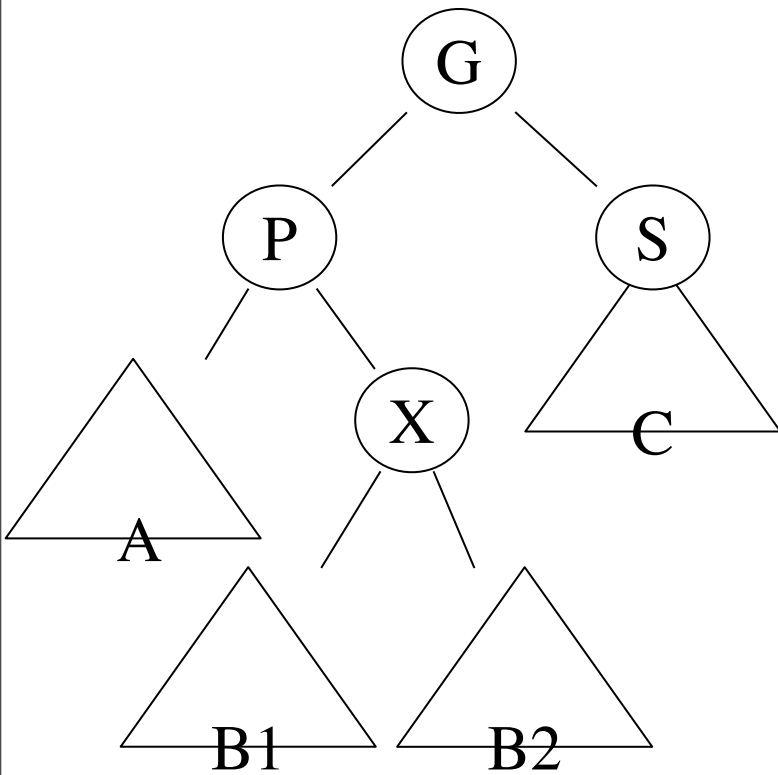
+ symmetrisk tilfelle...

Zig-zag rotasjon

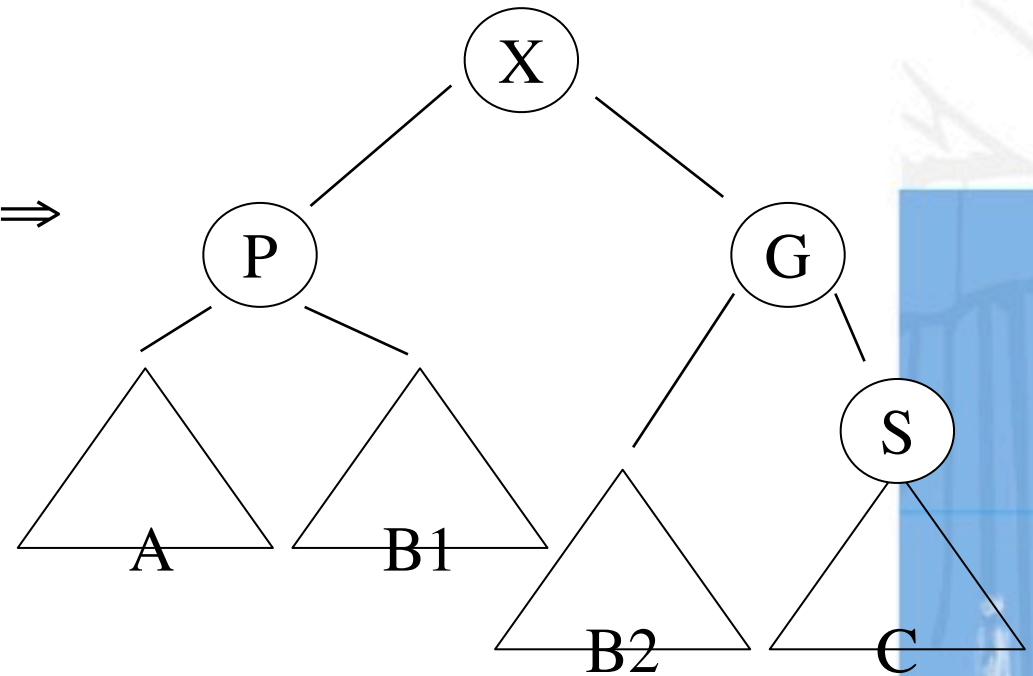


+ symmetrisk tilfelle...

Zig-zag rotasjon



⇒



+ symmetrisk tilfelle...

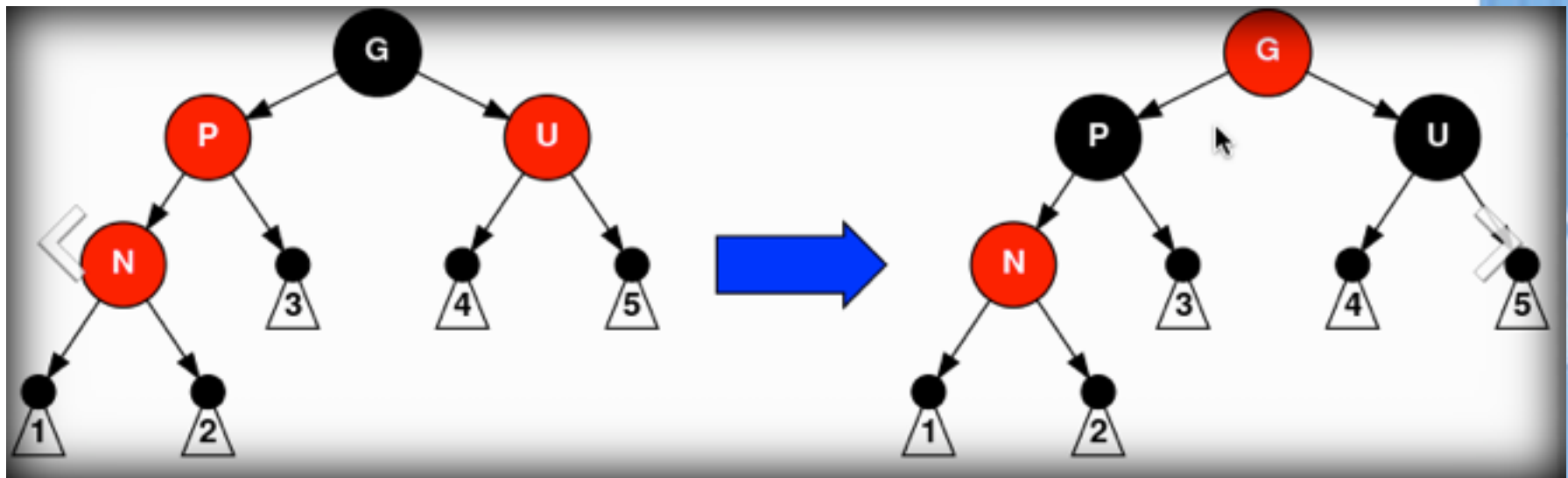


Case 1: The current node N is at
the root of the tree

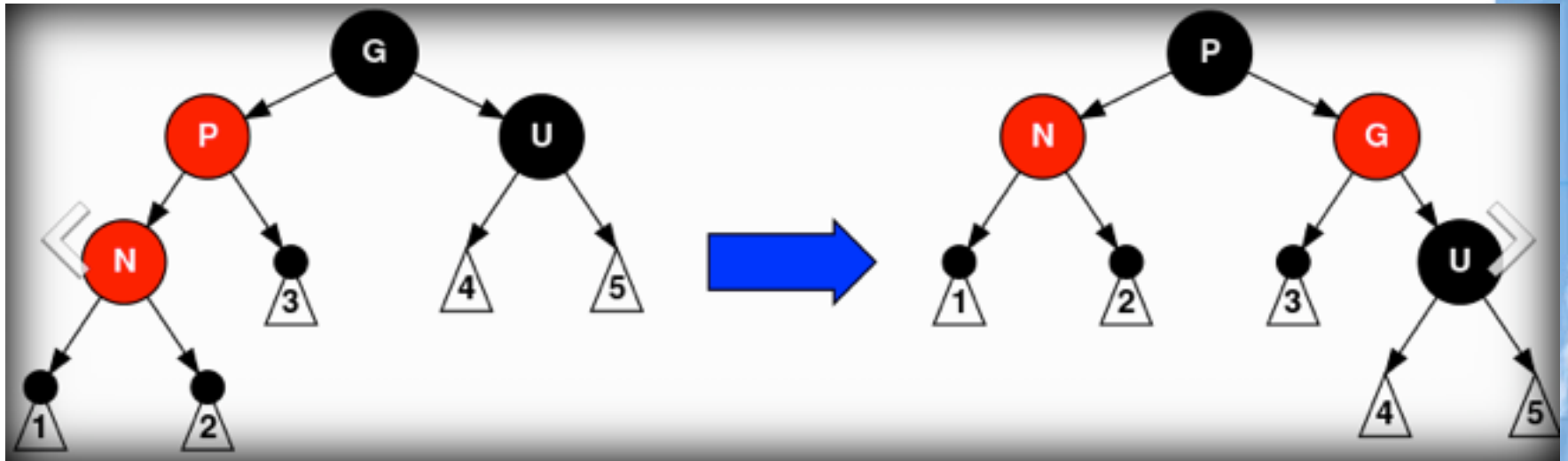


Case 2: The current node's parent P is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid

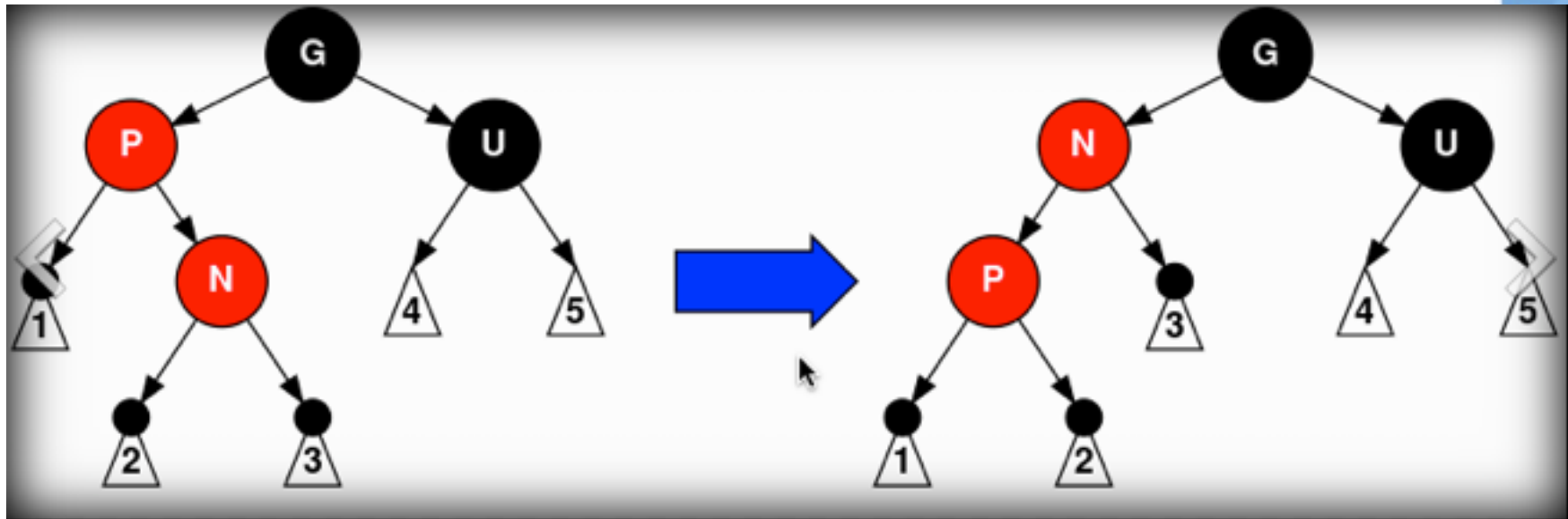
Case 3: Both the parent P and the
uncle U are red



Case 5: The parent P is red but the uncle U is black, the current node N is the left child of P , and P is the left child of its parent G .



Case 4: The parent P is red but the uncle U is black; also, the current node N is the right child of P , and P in turn is the left child of its parent G .





Antall operasjoner

$O(\log n)$



REPETISJON: B-TRÆR

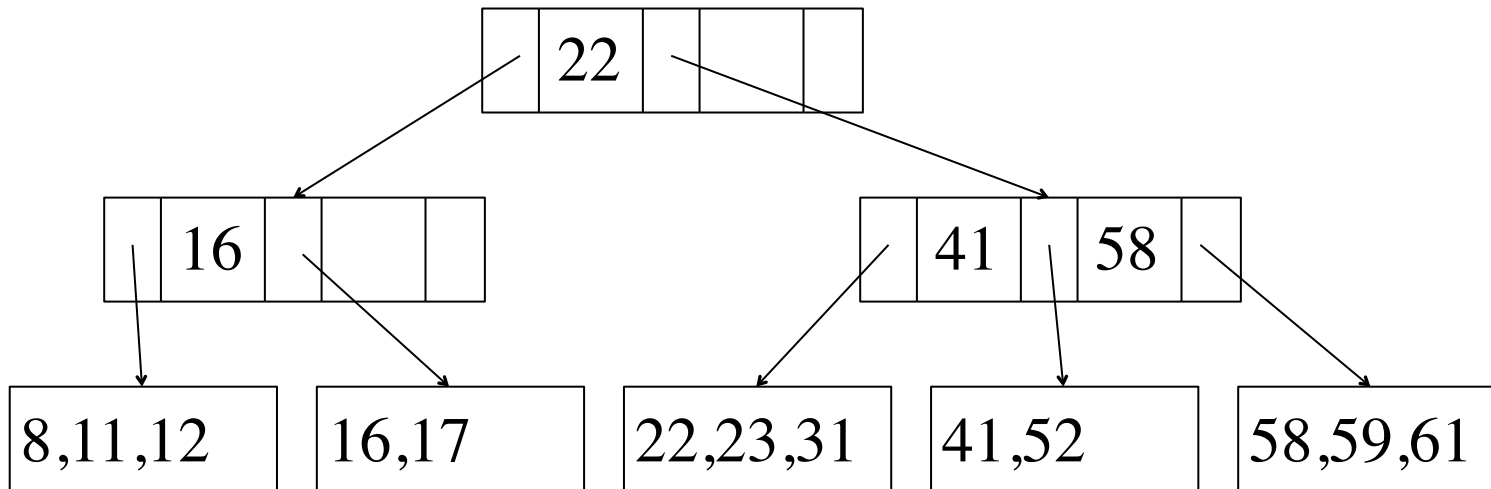


B-trær

- En annen type søketrær.
- Brukes først og fremst når ikke hele treet får plass i internminnet.
- Har stor bredde (hver node har mange barn).
- Er balansert.
- De øverste nivåene lagres i internminnet, resten på disk.
- Brukes særlig i databasesystemer.

Merk: Lærebokens (og våre) B-trær kalles vanligvis B⁺-trær. Tradisjonelle B-trær har data(pekere) i alle noder.

B-trær: eksempel

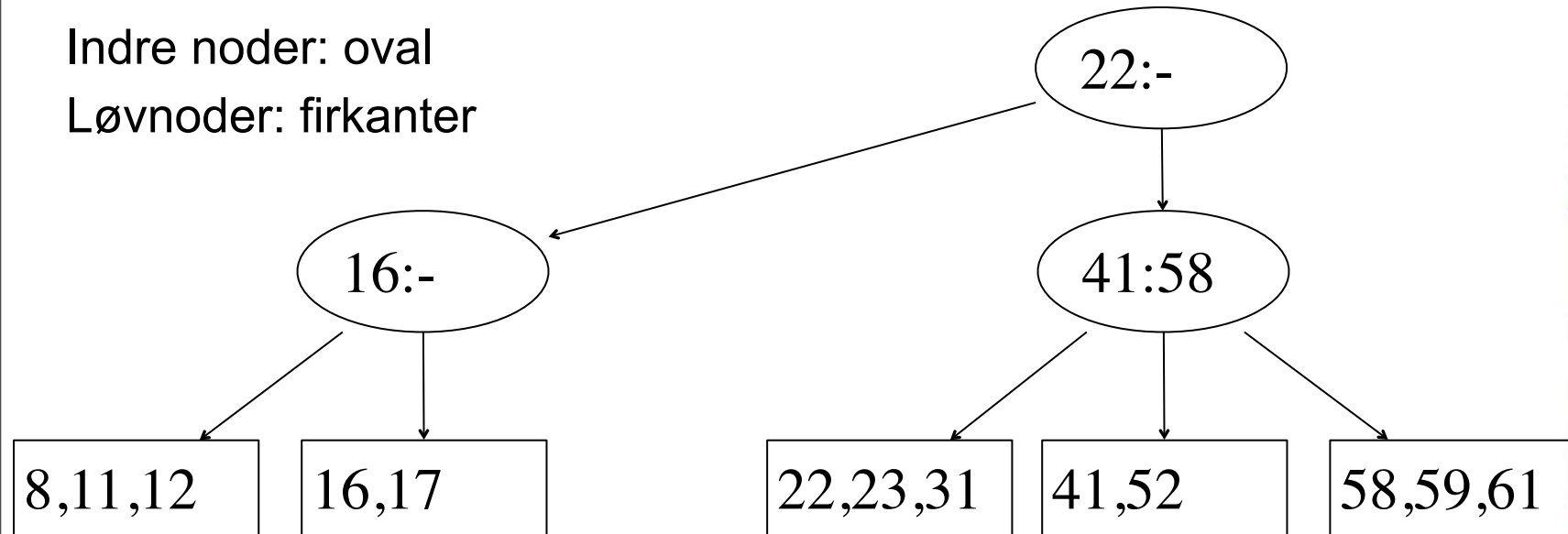


Dette treet har $M=3$ pekere fra hver indre node og $L=3$ dataelementer i hver løvnode.

B-trær: forenklet tegning

Indre noder: oval

Løvnoder: firkanter





Definisjon: B-trær av orden M

1. Alle data (eller pekere til data) er lagret i løvnodene.
2. Interne noder lagrer inntil $M-1$ nøkler for søking: nøkkel i angir den minste verdien i subtre $i+1$.
3. Roten er
 - enten en løvnode
 - eller har mellom 2 og M barn.
4. Alle andre indre noder har mellom $\lceil M/2 \rceil$ og M barn.
5. Alle løvnoder har samme dybde.
6. Alle løvnoder har mellom $\lceil L/2 \rceil$ og L dataelementer (eller datapekere).



Søking etter element x

1. Start i roten.
2. Så lenge vi ikke er i en løvnode:
La nøkkel-verdiene bestemme hvilket barn vi skal gå til.
3. Let etter x i løvnoden.



Innsetting av element x

1. Let etter riktig løvnode for x (som for søking).
2. Dersom det er plass, sett inn x og oppdater nøkkelverdiene langs veien vi gikk.



Innsetting av element x

1. Let etter riktig løvnode for x (som for søking).
2. Dersom det er plass, sett inn x og oppdater nøkkelverdiene langs veien vi gikk.
3. Dersom løvnoden er full, deler vi den i to og fordeler de $L+1$ nøklene jevnt på de to nye løvnodene.



Innsetting av element x

1. Let etter riktig løvnode for x (som for søking).
2. Dersom det er plass, sett inn x og oppdater nøkkelverdiene langs veien vi gikk.
3. Dersom løvnoden er full, deler vi den i to og fordeler de $L+1$ nøklene jevnt på de to nye løvnodene.
4. Dersom splittingen medfører at foreldernoden får for mange, må den også splittes (osv oppover i treet).



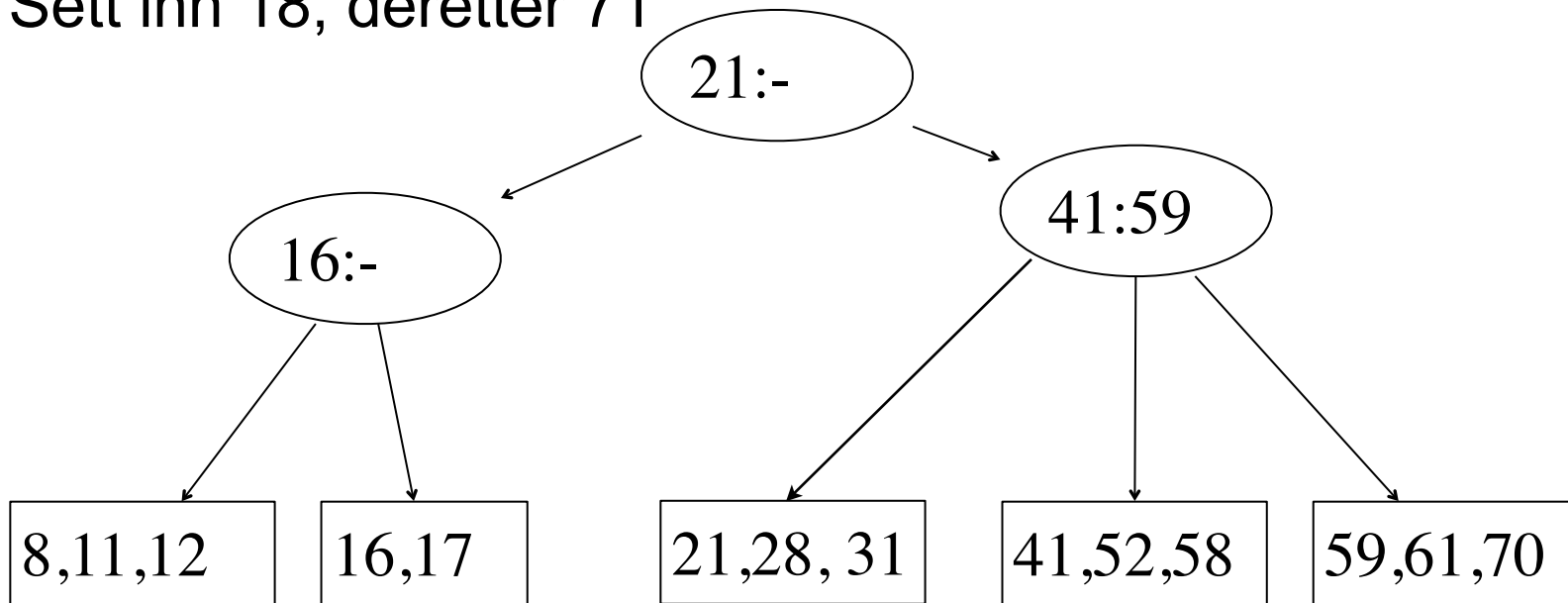
Innsetting av element x

1. Let etter riktig løvnode for x (som for søking).
2. Dersom det er plass, sett inn x og oppdater nøkkelverdiene langs veien vi gikk.
3. Dersom løvnoden er full, deler vi den i to og fordeler de $L+1$ nøklene jevnt på de to nye løvnodene.
4. Dersom splittingen medfører at foreldernoden får for mange, må den også splittes (osv oppover i treet).
5. Dette kan medføre at vi til slutt må splitte roten i to (dersom roten får $M+1$ barn).
Merk: Dette er det eneste som gjør at et B-tre vokser i høyden!

Insetting: eksempel

$M = L = 3$

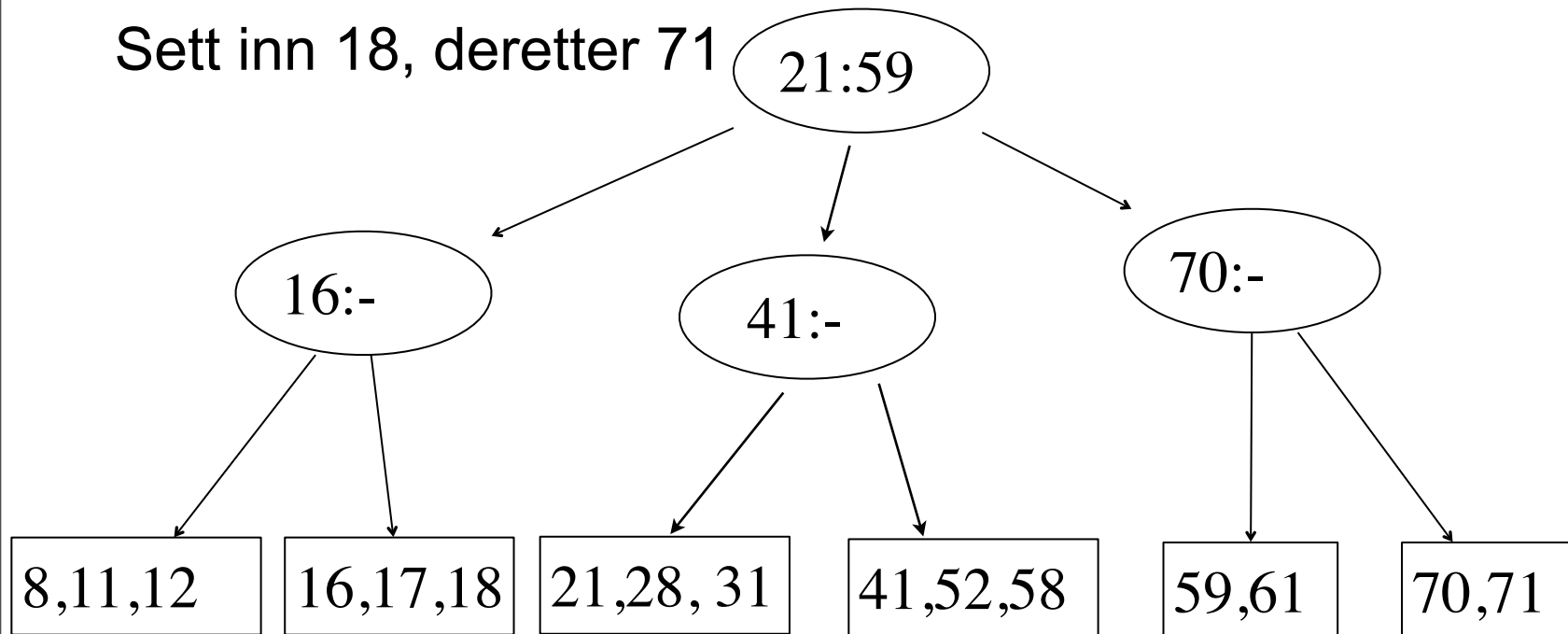
Sett inn 18, deretter 71



Insetting: eksempel

$M = L = 3$

Sett inn 18, deretter 71





ABSTRAKTE DATATYPER



Abstrakte datatyper

En ADT består av:

- Et sett med objekter.
- Spesifikasjon av operasjoner på disse.

Eksempler:

- ADT: binært søketre
Operasjoner: innsetting, søking, fjerning, ...
- ADT: mengde
Operasjoner: union, snitt, finn, ...
- ADT: map
Operasjoner: containsKey, get, put, ...



Hvorfor bruke ADTer?

ADTer skiller det som er viktig (funksjonaliteten) fra detaljene (den konkrete implementasjonen). Dermed kan vi:

- **Gjenbruke** ADTen i andre programmer.
- Enklere overbevise oss om at programmet er **riktig**.
- **Forandre** innmaten (kodingen) av ADTen uten å forandre resten av programmet fordi grensesnittet er det samme.
- Lage **modulære** programmer.

I Java er det naturlig å spesifisere en ADT som et interface.



MAP





Map – ADT

- Samling (nøkkel,verdi)-par.
- Nøklene må være unike.
- Viktigste operasjoner:
 - containsKey(key): returnerer true hvis key finnes som nøkkel
 - get(key): returnerer verdien assosiert med key
 - put(key,value): legger til et nytt nøkkel/verdi-par
 - keySet(): returnerer alle nøklene (som et sett)
 - values(): returnerer alle verdiene
- SortedMap: Nøklene er organisert i sortert orden.



HASHING



Hashing – innledende eksempel

Anta at en bilforhandler har 50 ulike modeller han ønsker å lagre data om.

Hvis hver modell har et entydig nummer mellom 0 og 49 kan vi enkelt lagre dataene i en array som er 50 lang.

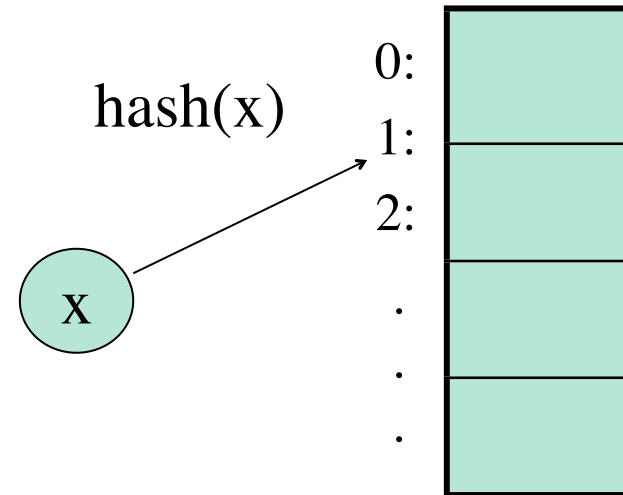
Hva hvis numrene ligger mellom 0 og 49 999?

- Array som er 50 000 lang:
 - Sløsing med plass!
- Array som er 50 lang:
 - Søking tar lineær tid...

Hashtabeller

Ideen i hashing er å

- lagre elementene i en array (hashtabell).
- la verdien til elementet x (eller en del av x , da kalt nøkkelen til x), bestemme plasseringen (indeksen) til x i hashtabellen.



Egenskaper til en god hash-funksjon:

- **Rask** å beregne.
- Kan **gi alle mulige verdier** fra 0 til $\text{tableSize} - 1$.
- Gir en **god fordeling** utover tabellindeksene.



Idealsituasjonen

Perfekt situasjon:

- n elementer
- tabell med n plasser
- hash-funksjon slik at
 - den er lett (rask) å beregne
 - forskjellige nøkkelverdier gir forskjellige indekser

Eksempel:

Hvis modellene er nummerert

0, 1 000, 2 000, ..., 48 000, 49 000

kan data om modell i lagres på indeks $i/1\ 000$ i en tabell som er 50 stor.

Problem: Hva hvis modell 4 000 får nytt nummer 3 999?

Eksempel: ideell situasjon

Input: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Hash-funksjon: $\text{hash}(x, \text{tableSize}) = \sqrt{x}$

| | |
|----|----|
| 0: | 0 |
| 1: | 1 |
| 2: | 4 |
| 3: | 9 |
| 4: | 16 |
| 5: | 25 |
| 6: | 36 |
| 7: | 49 |
| 8: | 64 |
| 9: | 81 |

Hva hvis hash-funksjonen
hadde vært

$\text{hash}(x, \text{tableSize}) =$
 $x \bmod \text{tableSize}$
istedenfor?

Eksempel: ideell situasjon

Input: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Hash-funksjon: $\text{hash}(x, \text{tableSize}) = \sqrt{x}$

| | |
|----|----|
| 0: | 0 |
| 1: | 1 |
| 2: | 4 |
| 3: | 9 |
| 4: | 16 |
| 5: | 25 |
| 6: | 36 |
| 7: | 49 |
| 8: | 64 |
| 9: | 81 |

Hva hvis hash-funksjonen
hadde vært

$\text{hash}(x, \text{tableSize}) =$
 $x \bmod \text{tableSize}$
istedenfor?

| | |
|----|--|
| 0: | |
| 1: | |
| 2: | |
| 3: | |
| 4: | |
| 5: | |
| 6: | |
| 7: | |
| 8: | |
| 9: | |

Eksempel: ideell situasjon

Input: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Hash-funksjon: $\text{hash}(x, \text{tableSize}) = \sqrt{x}$

| | |
|----|----|
| 0: | 0 |
| 1: | 1 |
| 2: | 4 |
| 3: | 9 |
| 4: | 16 |
| 5: | 25 |
| 6: | 36 |
| 7: | 49 |
| 8: | 64 |
| 9: | 81 |

Hva hvis hash-funksjonen
hadde vært

$\text{hash}(x, \text{tableSize}) =$
 $x \bmod \text{tableSize}$
istedenfor?

| | |
|----|--|
| 0: | |
| 1: | |
| 2: | |
| 3: | |
| 4: | |
| 5: | |
| 6: | |
| 7: | |
| 8: | |
| 9: | |



Hashtabell – ADT

En hashtabell tilbyr

- innsetting
- sletting
- søking

med **konstant** gjennomsnittstid.

Men: operasjoner som `finnMinste` og `skrivSortert` har **ingen** garantier.



Når bruker vi hashtabeller?

Brukes gjerne når vi først og fremst ønsker et raskt svar på om et gitt element finnes i datastrukturen eller ikke.

Eksempler:

- **Kompilatorer**: Er variabel y deklarerert?
- **Stavekontroller**: Finnes ord x i ordlisten?
- **Spill**: Har jeg allerede vurdert denne stillingen via en annen trekkrekkefølge?
- **XML parsing**: Nøkler blir attributtnavn, verdi blir innhold.

Nesten alle scriptspråk has hash som del av språket. (Perl, Python, Ruby, PHP, ...)



Hashing: problemstillinger

- Hvordan velge **hash-funksjon**?
 - Ofte er nøklene strenger.
- Hvordan håndtere **kollisjoner**?
- Hvor **stor** bør hash-tabellen være?



Hash-funksjoner

Eksempel:

- Heltall som nøkler
- Begrenset antall tabellindekser
- La hash-funksjonen være
$$\text{hash}(x, \text{tableSize}) = x \bmod \text{tableSize}$$
- Dette gir jevn fordeling for tilfeldige tall.

Pass på at ikke nøklene har spesielle egenskaper: Hvis $\text{tableSize} = 10$ og alle nøklene slutter på 0 vil alle elementene havne på samme indeks!

Huskeregel: La alltid tabellstørrelsen være et **primtall**.

Strenger som nøkler: funksjon 1

Vanlig strategi: ta utgangspunkt i ascii/unicode-verdiene til hver bokstav og "gjør noe lurt".

Funksjon 1: **Summer verdiene** til hver bokstav.

```
int hash1(String key, int tableSize) {
    int hashVal = 0;

    for (int i = 0; i < key.length(); i++) {
        hashVal += key.charAt(i);
    }

    return (hashVal % tableSize);
}
```

Fordel: Enkel å implementere og beregne.

Ulempe: Dårlig fordeling hvis tabellstørrelsen er stor.



Strenger som nøkler: funksjon 2

Funksjon 2: Bruk bare de **tre første bokstavene** og vekt disse.

```
int hash2(String key, int tableSize) {
    int hashVal = key.charAt(0) +
        27 * key.charAt(1) +
        729 * key.charAt(2);

    return (hashVal % tableSize);
}
```

Fordel: Grei fordeling for tilfeldige strenger.

Ulempe: Vanlig språk er ikke tilfeldig!

Strenger som nøkler: funksjon 3

Funksjon 3: $\sum_{i=0}^{\text{keySize}-1} \text{key}[\text{keySize}-i-1] * 37^i$

```
int hash3(String key, int tableSize) {
    int hashVal = 0;

    for (int i = 0; i < key.length(); i++) {
        hashVal = 37*hashVal + key.charAt(i);
    }

    return Math.abs(hashVal % tableSize);
}
```

Fordel: Enkel og relativt rask å beregne. Stort sett bra nok fordeling.

Ulempe: Beregningen tar lang tid for lange nøkler.



Hash-funksjoner: Oppsummering

- Må (i hvert fall teoretisk) kunne gi **alle mulige verdier** fra 0 til `tableSize - 1`.
- Må gi en **god fordeling** utover tabellindeksene.
- Tenk på hva slags data som skal brukes til nøkler.
 - Fødselsår kan gi god fordeling i persondatabaser, men ikke for en skoleklasse!

Generelt: Bør være mange ganger `tableSize` før man gjør mod-operasjonen.



hashCode i Java

Alle Java-klasser er subklasser av `java.lang.Object`, som inneholder metoden

```
int hashCode()
```

som typisk returnerer minneadresse konvertert til int.

Men: Objekter vi regner som like (via `equals`) må ha samme hashverdi.

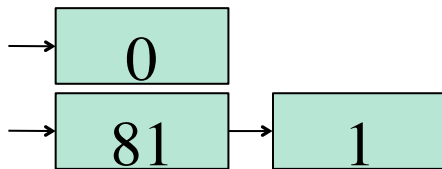


Kollisjonshåndtering

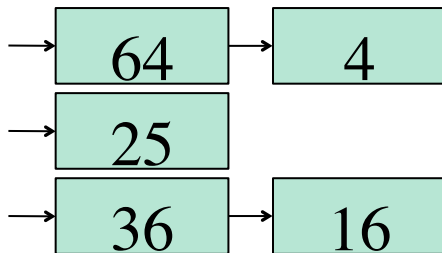
Hva gjør vi når to elementer hashes til den samme indeksen?

- **Åpen hashing:** Elementer med samme hashverdi samles i en liste (eller annen passende struktur).
- **Lukket hashing:** Dersom en indeks er opptatt, prøver vi en annen indeks inntil vi finner en som er ledig.

Åpen hashing (Separate chaining)

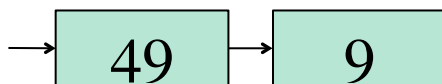


Forventer at hash-funksjonen er god, slik at alle listene blir korte.

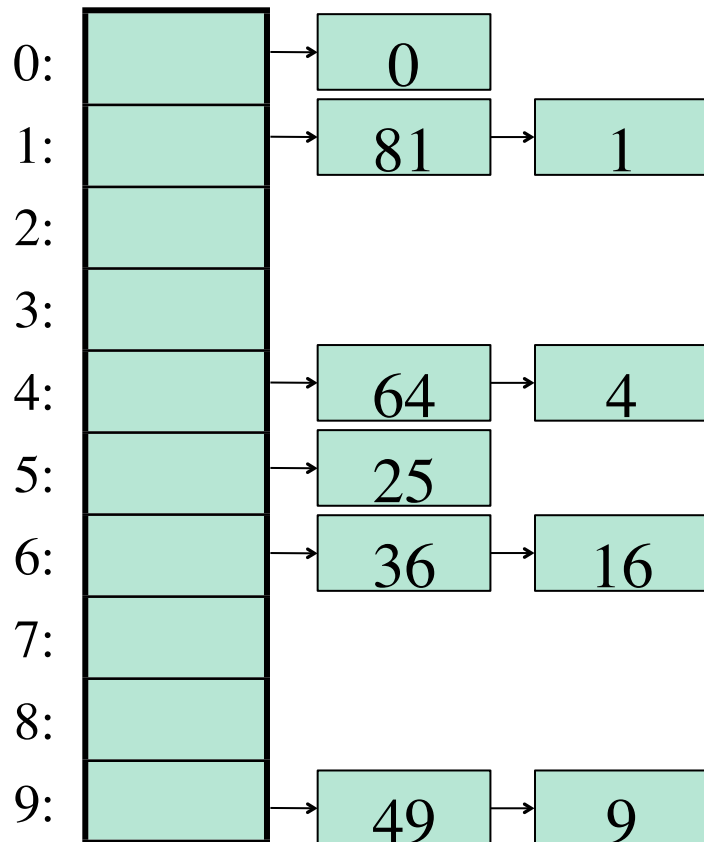


Load-faktoren λ er antall elementer i hash-tabellen i forhold til tabellstørrelsen.

For åpen hashing ønsker vi $\lambda \approx 1.0$



Åpen hashing (Separate chaining)



Forventer at hash-funksjonen er god, slik at alle listene blir korte.

Load-faktoren λ er antall elementer i hash-tabellen i forhold til tabellstørrelsen.
For åpen hashing ønsker vi $\lambda \approx 1.0$



Lukket hashing – åpen adressering

Prøver alternative indekser $h_0(x)$, $h_1(x)$, $h_2(x)$, ... inntil vi finner en som er ledig.

h_i er gitt ved

$$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{tableSize}$$

slik at $f(0) = 0$.

Merk at vi trenger en større tabell enn for åpen hashing – generelt ønsker vi her $\lambda < 0.5$

Skal se på tre mulige strategier (valg av f):

- Lineær prøving
- Kvadratisk prøving
- Dobbel hashing



Lineær prøving

Velger f til å være en **lineær** funksjon av i , typisk $f(i) = i$

Eksempel:

Input: 89, 18, 49, 58, 69

Hash-funksjon:

$$\text{hash}(x, \text{tableSize}) = x \bmod \text{tableSize}$$



Lineær prøving

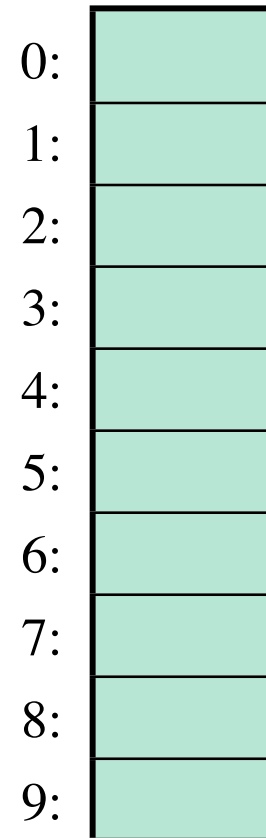
Velger f til å være en **lineær** funksjon av i , typisk $f(i) = i$

Eksempel:

Input: 89, 18, 49, 58, 69

Hash-funksjon:

$$\text{hash}(x, \text{tableSize}) = x \bmod \text{tableSize}$$





Kvadratisk prøving

Velger f til å være en **kvadratisk** funksjon av i ,
typisk $f(i) = i^2$

Eksempel:

Input: 89, 18, 49, 58, 69

Hash-funksjon:

$$\text{hash}(x, \text{tableSize}) = x \bmod \text{tableSize}$$



Kvadratisk prøving

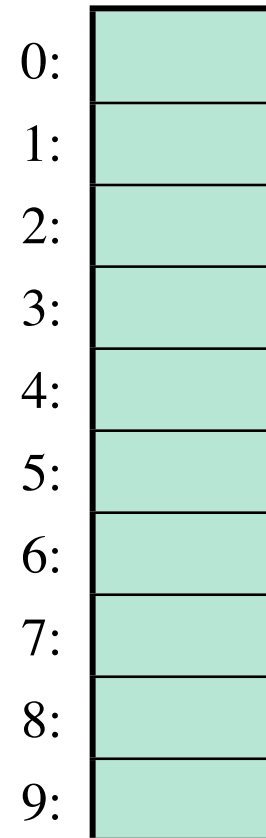
Velger f til å være en **kvadratisk** funksjon av i,
typisk $f(i) = i^2$

Eksempel:

Input: 89, 18, 49, 58, 69

Hash-funksjon:

$$\text{hash}(x, \text{tableSize}) = x \bmod \text{tableSize}$$





Dobbel hashing

Bruker en **ny hash-funksjon** for å løse kollisjonene, typisk $f(i) = i * \text{hash}_2(x)$, med $\text{hash}_2(x) = R - (x \bmod R)$ hvor R er et primtall mindre enn `tableSize`.

Eksempel:

Input: 89, 18, 49, 58, 69

Andre hash-funksjon:

$$\text{hash}_2(x) = 7 - (x \bmod 7)$$

Dobbel hashing

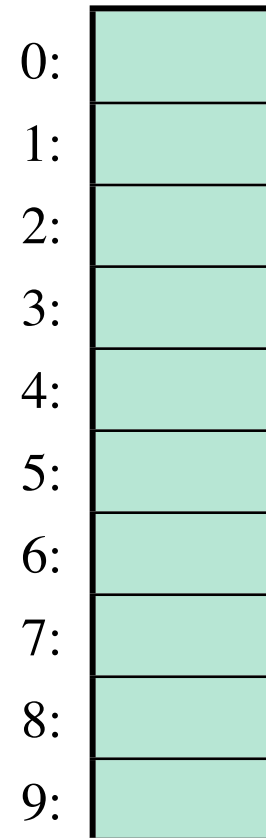
Bruker en **ny hash-funksjon** for å løse kollisjonene, typisk $f(i) = i * \text{hash}_2(x)$, med $\text{hash}_2(x) = R - (x \bmod R)$ hvor R er et primtall mindre enn tableSize .

Eksempel:

Input: 89, 18, 49, 58, 69

Andre hash-funksjon:

$$\text{hash}_2(x) = 7 - (x \bmod 7)$$





Rehashing

Hvis tabellen blir for full, begynner operasjonene å ta veldig lang tid.

Mulig løsning:

- Lag en ny hashtabell som er omtrent dobbelt så stor (men fortsatt primtall!).
- Gå gjennom hver element i den opprinnelige tabelln, beregn den nye hash-verdien og sett inn på rett plass i den nye hashtabellen.

Dette er en dyr operasjon, $O(n)$, men opptrer relativt sjelden (må ha hatt $n/2$ innsetninger siden forrige rehashing).

- **Amortised cost** $O(1)$



Utvidbar hashing

Brukes spesielt når internminnet blir for lite, og det vesentligste blir antall diskoperasjoner.

Anta at vi

- Skal lagre N elementer, der N varierer over tid.
- Kan lagre maksimalt M elementer i en diskblokk.

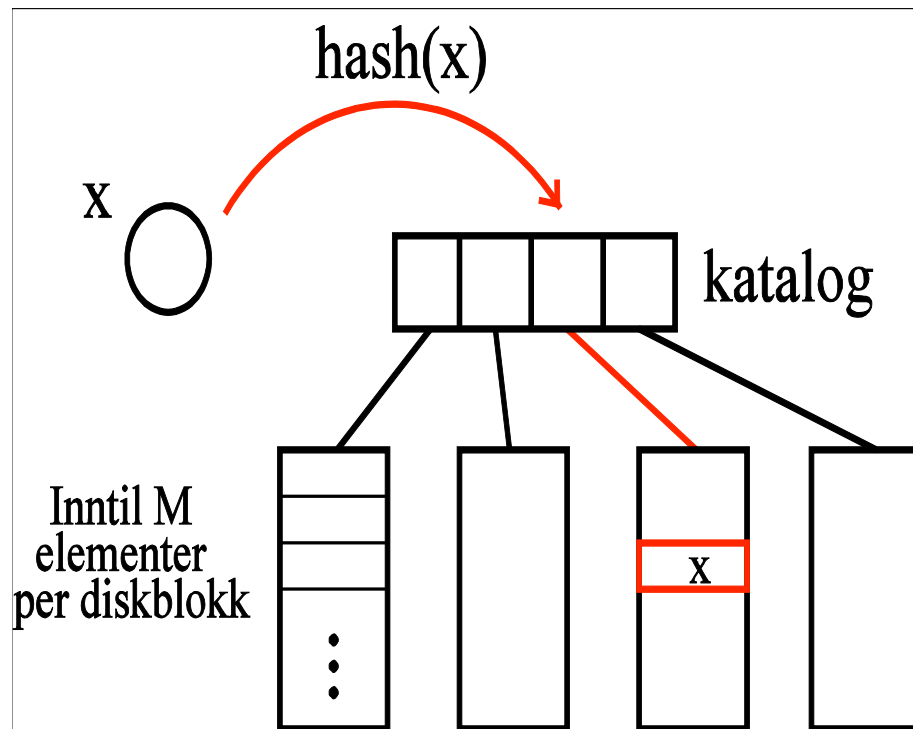
Problemet med vanlig hashing er at

- Kollisjoner kan føre til at $\text{find}(x)$ må undersøke mange diskblokker selv om hashfunksjonen distribuerer elementene godt.
- Rehashing blir veldig kostbart.

Løsning

Lar hash-funksjonen – via en katalog – angi hvilken diskblokk et element x befinner seg i (hvis det finnes).

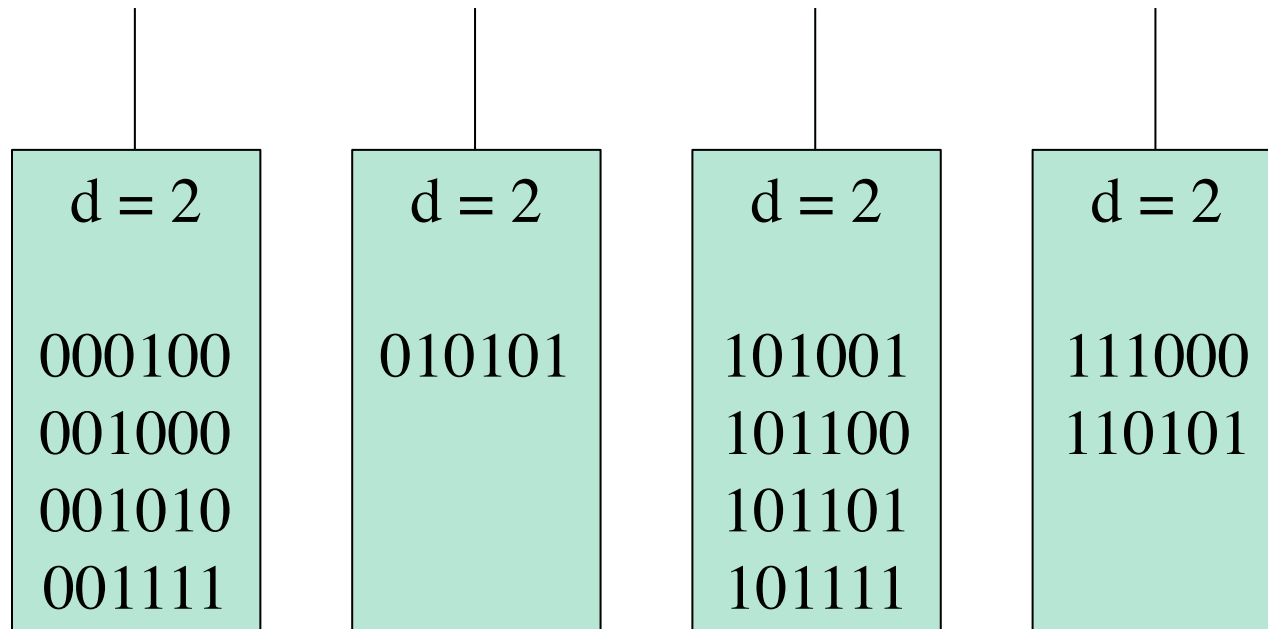
Dermed trenger $\text{find}(x)$ bare to diskaksesser (og bare en aksess dersom katalogen kan lagres i internminnet).



Utvidbar hashing: eksempel

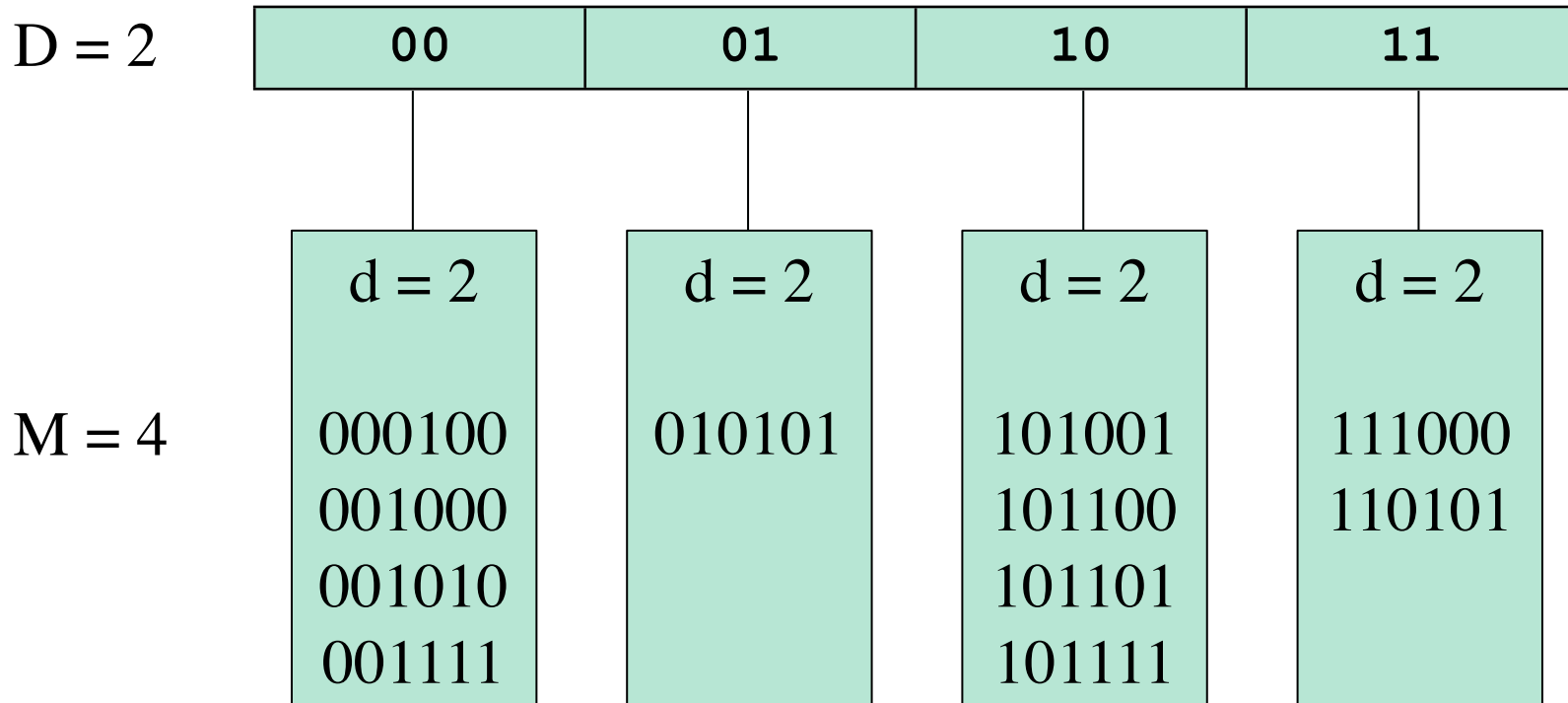
$D = 2$

$M = 4$



Forenklet tegning, hvor vi for hvert element bare lagrer hashverdien til elementet (som 6-sifret binærtall).

Utvidbar hashing: eksempel



Forenklet tegning, hvor vi for hvert element bare lagrer hashverdien til elementet (som 6-sifret binærtall).



Utvidbar hashing

- Katalogen har 2^D indekser.
- Hver diskblokk har plass til M elementer.
- For hver diskblokk L lagrer vi et tall $d_L \leq D$.

Invariant: Det garanteres at alle elementene i L har minst de d første bitene felles.

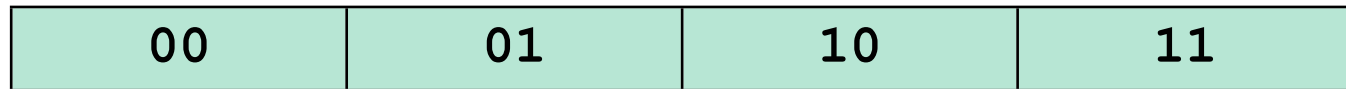
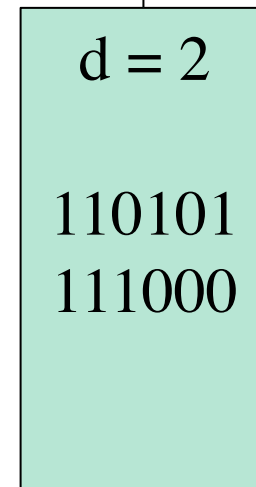
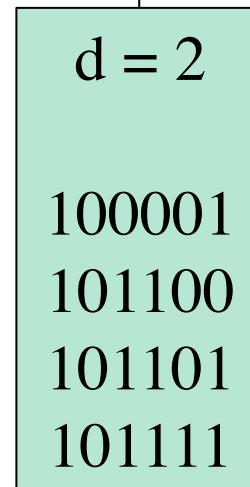
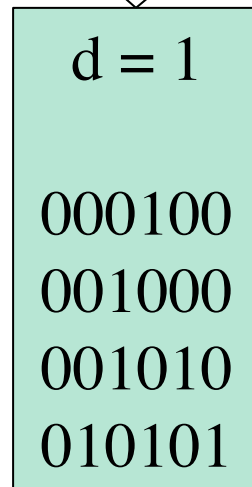
- Dersom $d_L = D$ vil nøyaktig en indeks i katalogen peke på diskblokk L .
 - Dersom $d_L < D$ vil to eller flere indekser i katalogen peke på L .
- Over tid forventer vi at ca 69% av hver diskblokk er fylt opp.



Innsettingsalgoritme

1. Beregn $\text{hash}(x)$ og finn riktig diskblokk L ved å slå opp i katalogen på de D første sifrene i hashverdien.
2. Hvis det er færre enn M elementer i L , sett x inn i L .
3. Hvis L derimot er full, sammenlign d_L med D :
 - a) Dersom $d_L < D$ splitter vi L i to blokker L_1 og L_2 :
 - i. Sett $d_{L_1} = d_{L_2} = d_L + 1$.
 - ii. Gå gjennom elementene i L og plasser dem i L_1 eller L_2 avhengig av verdien på de $d_L + 1$ første sifrene.
 - iii. Prøv igjen å sette inn x (gå til punkt 2).
 - b) Dersom $d_L = D$:
 - i. Doble katalogstørrelsen ved å øke D med 1.
 - ii. Fortsett som ovenfor (splitt L i to blokker osv.)

Innsetting: eksempel

 $D = 2$  $M = 4$ 



Java's HashMap

Klassen `java.util.HashMap`:

- Implementerer en hashtabell som mapper nøkler til verdier.
- Implementerer grensesnittet `Map` (men ikke `SortedMap`).
- Bruker åpen hashing.
- Default tabellstørrelse 16.
- Default load faktor $\lambda < 0.75$
 - rehashing hvis denne overskrides.



Data structure animation





Neste forelesning: 14. september

PRIORITETSKØ OG HEAP