Uke 6,
Forelesning 2

- Forutsetninger for og essensen i faget
- Metodekall, rekursjon, permutasjoner
- Analyse av algoritmer
- Introduksjon til ADT'er
- De første ADT'er: Lister, stabler og køer

Uke 1, Uke 2 og Uke 3

- Flere ADT'er: Generelle trær, binære trær og binære søketrær
- Oblig. 1 og "dronning konkurransen"

Uke 4 og Uke 5

- Flere ADT'er: Hashing, hash-tabeller

Uke 6

OVERSIKT – *Uke 6, Forelesning 2 (W6.L2)*

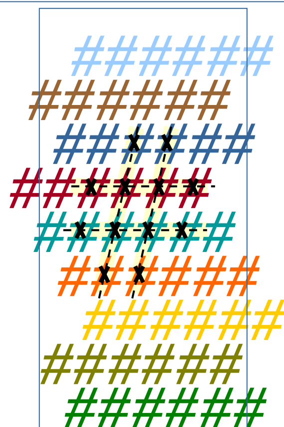
Vi avslutter hashing (Kap. 5):

- Kollisjoner og kollisjonshåndtering
- Åpen hashing
- Lukket hashing
  - Lineær prøving
  - Kvadratisk prøving
  - Dobbel hashing
- Rehashing
- (Vi ikke helt ferdig med hashing enda)

Så tar vi opp datastrukturer for lagring/gjenfinning på disk:

- Utvidbar hashing
- B-trær (eller B^+-trær)

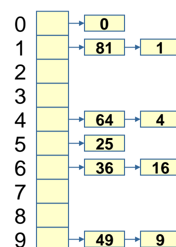
TEMA: HASHING



Hashing fortsetter...

- **Kollisjon:**
  Hva gjør vi hvis 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.
- Det finnes flere strategier for å velge hvilken annen indeks vi skal prøve.

- **Åpen hashing**, "Separate Chaining":
  Definerer lister for hver hash-tabell plass, og legger en gjentagelse i listen på sin plass



- Vi forventer at hash-funksjonen er god, slik at alle listene blir korte.
- Vi definerer load-faktoren, λ, til en hash-tabell til å være antall elementer i 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:
  $h_i(x) = (hash(x) + f(i)) \bmod tableSize$
  og slik at $f(0) = 0$.
- Merk at vi trenger en større tabell enn for åpen hashing — generelt ønsker vi her $\lambda < 0.5$.
- Vi skal se på tre mulige strategier for 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:
  $hash(x, tableSize) = x \bmod tableSize$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

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:
  hash(x, tableSize) = x mod tableSize

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Dobbel hashing

- Bruker en ny hash-funksjon for å løse kollisjonene, typisk $f(i) = i \cdot hash_2(x)$, med $hash_2(x) = R - (x \bmod R)$ hvor R er et primtall mindre enn tableSize.

Eksempel

- Input:
  89, 18, 49, 58, 69
- Andre hash-funksjon:
  $hash_2(x) = 7 - (x \bmod 7)$

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Rehashing

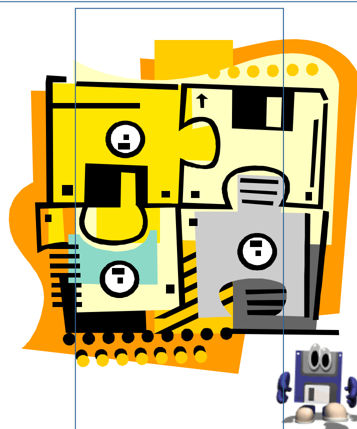
- Hvis tabellen blir for full, begynner operasjonene å ta veldig lang tid.
- Mulig løsning:
  - Lag en ny hash-tabell som er omtrent dobbelt så stor – Men husk: Fortsatt primtall!
  - Gå gjennom hvert element i den opprinnelige tabellen, beregn den nye hash-verdien og sett inn på rett plass i den nye hash-tabellen.
- Dette er en dyr operasjon, **O(n)**, men opptrer relativt sjelden (må ha hatt **n/2** innsettinger siden forrige rehashing).

Hash-funksjoner:

- Må (i hvert fall teoretisk) kunne gi alle mulige verdier fra **0** til **tableSize - 1**.
- Må gi en god fordeling utover tabellindeksene.
- Tenk nøye på hva slags data som skal brukes til nøkler!

  Eksempel: Fødselsår kan gi god fordeling i persondatabaser, men ikke for en skoleklasse der studentene er sannsynligvis født i det samme året alle sammen!

- **Bursdagsparadokset:** Hvor mange tilfeldig valgte personer skal til før det er sannsynlig at to av dem har bursdag på samme dag? (Svar: 24)
  Dvs.: Det hender nesten aldri at alle elementer hasher til forskjellige indekser!
- Dermed **kollisjonshåndtering:**
  - **Åpen hashing:** Elementer med samme hash-verdi 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.
- Ved lukket hashing kan vi ikke ha for full tabell (antall elementer < ½ × tabell størrelsen).
- Dersom antall elementer blir for stort i forhold til hash-tabellen, kan det lønne seg å flytte elementene til en større tabell (rehashing).
- Og husk at rehashing er kostbart, dvs. $O(n)$



Datastrukturer for data på disk...

Når internminnet blir for lite...

- En lese-/skriveoperasjon på en harddisk (aksesstid 7-12 millisekunder) tar nesten 1 000 000 ganger lenger tid enn tilsvarende operasjon i internminnet (aksesstid ca. 10 nanosekunder).
- O–modellen er ikke lenger gyldig ved hyppige diskaksesser siden den forutsetter at alle operasjonene tar like lang tid
- Dersom ikke hele datastrukturen får plass i internminnet (hurtiglageret), lønner det seg nesten alltid å bruke ADT'er som minimaliserer antall diskoperasjoner.

- To datastrukturer, "utvidelser" av ADT'er vi har sett, er nettopp for data som ligger på disk:
  Utvidbar hashing og **B-trær**...

Utvidbar hashing: Anta at vi...

- skal lagre **N** dataelementer og at **N** varierer med tiden.
- kan lagre maksimalt **M** elementer i en **diskblokk** (= den delen av harddisken som kan hentes inn i **én** enkel leseoperasjon).

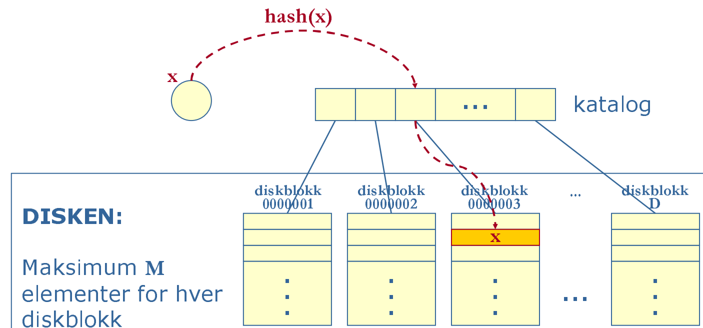
Problemet med å bruke vanlig hashing er at...

- kollisjoner kan føre til at **find(x)** må undersøke mange disk-blokker selv om hash-funksjonen distribuerer elementene godt (bursdagsparadokset).
- rehashing blir fryktelig kostbart.

UTVIDBAR HASHING – *En løsning #1*

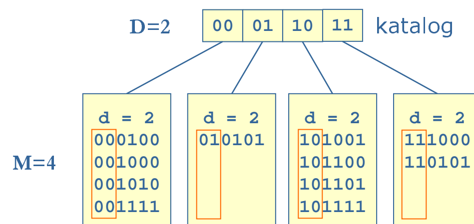
En mulig løsning:

- Vi lar hash-funksjonen — via en katalog — angi hvilken diskblokk et element x befinner seg i (hvis det finnes).
- Dermed trenger **find(x)** bare to diskaksesser (og bare en aksess dersom katalogen kan lagres i internminnet).



UTVIDBAR HASHING – *En løsning #2*

- I diskblokken lagrer vi for hvert element
  - den binære strengen som koder hash-verdien til elementet.
  - "innmaten" i elementet eller en peker til innmaten dersom elementene er store (elementene kan foruten dataverdier være objekter eller klassiske dataposter).
  - Vi benytter de D første bitene av hash-verdien til oppslag i katalogen.



- I eksemplet over forenkler vi litt og lagrer for hvert element bare hash-verdien til elementet (som 6-sifret binært tall).

- Katalogen har $2D$ 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 (vi kan ikke skille to elementer i L fra hverandre ved å se på bare de d første bitene).
  - Dersom $d_L = D$ vil akkurat én indeks i katalogen peke på diskblokk L.
  - Dersom $d_L < D$ vil to eller flere indekser i katalogen peke på L.

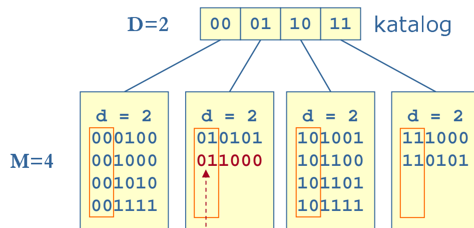
Innsettingsalgoritme

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

Eksempel

$M = 4$ (dvs. maks. 4 elementer i en diskblokk)

Antall bits i hash-verdiene $= 6$

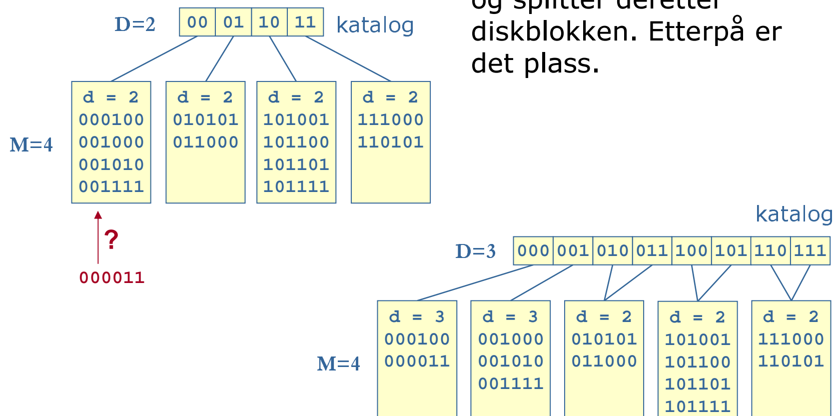


Sett inn element med hash-verdi '011000'

Det er ledig plass i diskblokken, så vi setter den rett inn.

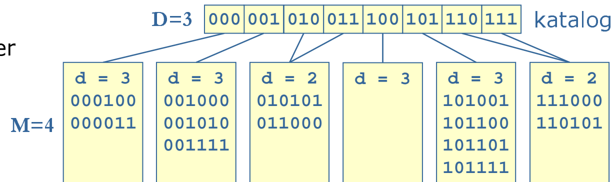
Videre: Sett inn '000011'

- Diskblokk '00' er full og $d_{00} = D$, … så vi utvider katalogen og splitter deretter diskblokken. Etterpå er det plass.

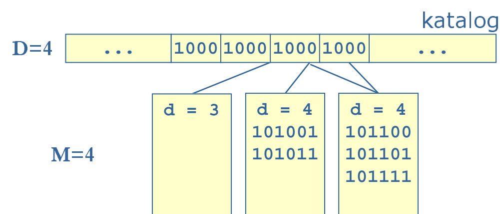


UTVIDBAR HASHING – *Eksempel #3*

Sett nå inn '101011'
Diskblokk '101' er full og $d_{101} < D$, så vi splitter diskblokken:



Diskblokk '101' er fortsatt full, så vi må øke D til 4 og splitte en gang til.



Merk! Algoritmen virker ikke hvis det er mer enn M elementer som hasher til samme verdi.

UTVIDBAR HASHING – *Fyllingsgrad*

Fyllingsgrad til diskblokkene

- Katalogen blir veldig stor dersom mange elementer har mange ledende sifre felles.
- Får å få (forhåpentligvis) god spredning bruker vi hash-funksjon i stedet for å bruke nøkkelen til elementet direkte.
- Hvis vi antar uniform fordeling av bitmønstrene, er antall diskblokker

$$B = \frac{N}{M}\log_2 e = \frac{N}{M}\times\frac{1}{\ln 2} \approx 1{,}443\frac{N}{M}$$

… Siden ikke alle blokker har alle maks. M elementer, er det "noe mer" enn N/M

- Forventet antall elementer pr. diskblokk

$$E = \frac{N}{B} = \frac{N}{\frac{N}{M}\times\frac{1}{\ln 2}} = N\times\frac{M}{N}\times\frac{\ln 2}{1} = M\times\ln 2 \approx 0{,}693\times M$$

dvs. at i det "lange løp" vil ca. 69% av hver diskblokk være fylt opp.

- Forventet størrelse på katalogen er

$$K = \frac{N^{1+\frac{1}{M}}}{M}$$

dvs. at liten *M* gir stor katalog.

B-TRÆR – *Introduksjon*

- B-trær er søketrær, men de er ikke binære!
- B-trær brukes først og fremst når ikke hele treet får plass i internminnet.
- Et B-tre har stor bredde (hver node har mange barn) og er balansert.
- Dermed blir ikke treet så dypt.
- De øverste nivåene i treet (iallfall rot-noden) lagres i internminnet, resten av treet på disk…
- Og fordi treet ikke er så dypt, blir det få diskaksesser.
- Hvis alle elementene får plass i internminnet, er B-trær fortsatt en rimelig rask datastruktur, med garantert logaritmisk dybde (i motsetning til ubalanserte binære søketrær) for data på disk!
- Dermed brukes B-trær særlig i **databasesystemer**.

B-TRÆR – *Definisjon av B-trær av orden M*

Definisjon: B-trær av orden M

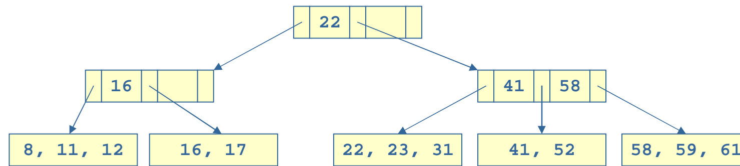
1. Alle data (eller pekere til data) er lagret i bladnodene.
2. Ikke–bladnoder lagrer inntil $M-1$ nøkler for bruk i søking; nøkkel i angir den minste nøkkel i subtre $i+1$.
3. Roten er enten en bladnode eller den har mellom 2 og M barn.
4. Alle ikke–bladnoder (unntatt roten) har mellom $\lceil M/2 \rceil$ og M barn.
5. Alle bladnoder har samme dybde og har mellom $\lceil L/2 \rceil$ og L dataelementer eller datapekere, der L er en konstant felles for alle bladnoder.

- Merk at B-trær er søketrær.
- Bladnodene inneholder de virkelige dataene, eventuelt pekere til objekter eller andre poster som inneholder dataene.

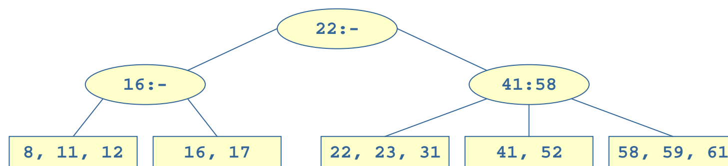
- **Merk:** Lærebokens (og våre) B-trær er ellers i litteraturen kjent som $\mathbf{B^+}$-trær. Tradisjonelle B-trær har Data (pekere) i alle noder.

B-TRÆR – *Eksempel*

Et eksempel på et B-tre av orden 3, også kalt et **2-3 tre**:



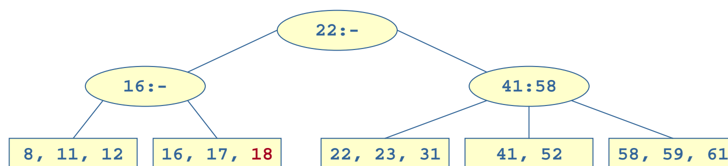
Her er M = 3 og L = 3. Vi skal forenkle tegningen litt:



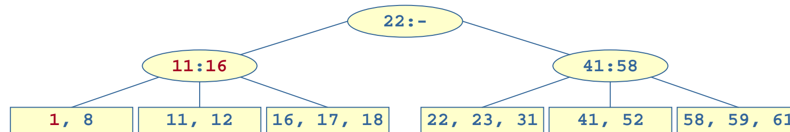
B-TRÆR – *Find & Insert #1*

- **Find(x)** er lett: Vi starter i roten og lar nøkkel–verdiene bestemme hvilken gren vi skal følge videre, helt til vi kommer til en bladnode.
- Så leter vi etter x i bladnoden.

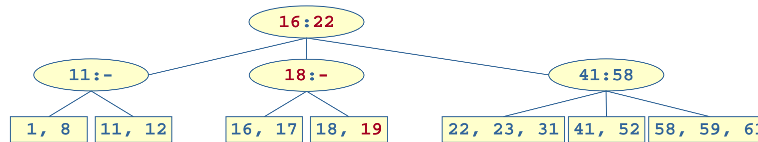
- **Insert(x)** er mer komplisert: Først finner vi riktig bladnode å sette x i, akkurat som i **find(x)**.
- Dersom det er plass til x i bladnoden, setter vi inn x og oppdaterer, om nødvendig, nøkkel–verdiene langs stien vi gikk.
- **Eksempel:** Sett inn '18' i treet på forrige foil.



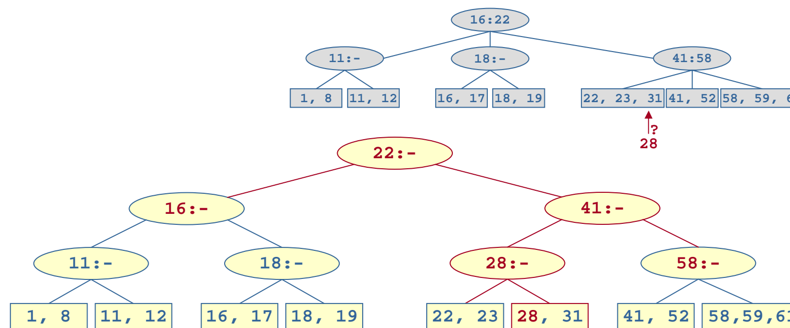
- Dersom bladnoden er full, deler vi den i to og fordeler de $L + 1$ nøklene jevnt på de to nye bladnodene.
- **Eksempel:** Sett inn '1' i treet på forrige foil.



- Dersom splittingen medfører at foreldernoden får for mange barn, splitter vi den også, gir den nye noden til besteforelderen, osv.
- **Eksempel:** Sett inn '19' i treet ovenfor.



- Dette kan medføre at vi til slutt må splitte roten i to: Det skjer hvis roten får $M + 1$ barn. Da lager vi en ny rot med den gamle roten og den nye noden som barn.
- **Dette er den eneste måten et B-tre kan vokse i høyden på!**
- **Eksempel:** Sett inn '28' i treet på forrige foil.

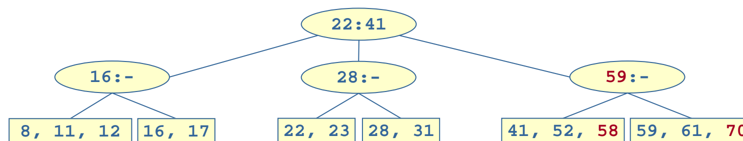


Sletting av et element x

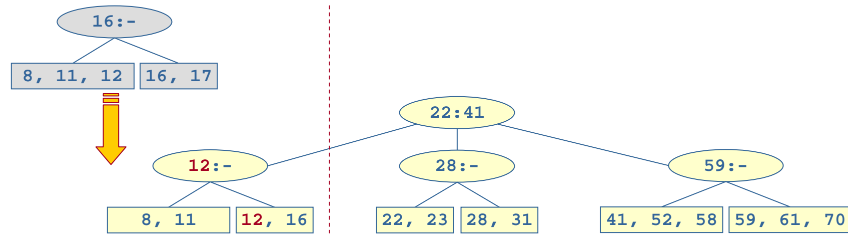
- Vi finner først elementet som skal slettes ved å traversere treet til riktig bladnode **B**.
- Dersom **B** har minst $\lceil L/2 + 1 \rceil$ elementer, kan vi uten komplikasjoner slette x fra bladnoden.
- Dersom **B** har akkurat $\lceil L/2 \rceil$ elementer, vil noden ha for få elementer etter at vi sletter x.
- Vi løser problemet ved å kombinere noden med en av nabosøsknene sine:
  - Dersom venstre (høyre) søsken har $\lceil L/2 + 1 \rceil$ elementer eller mer, flytter vi det største (minste) elementet til **B**.
  - Dersom nabosøskenen har akkurat $\lceil L/2 \rceil$ elementer, slår vi de to nodene sammen til én node (som får L eller $L - 1$ elementer, avhengig av om L er oddetall eller partall).

- Dette kan medføre at foreldernoden får et barn for lite.
  Vi kombinerer da foreldernoden med en av nabosøsknene sine, osv.
- Vi kan til slutt ende opp med en rot som har bare ett barn.
  I så fall sletter vi roten og lar barnet bli ny rot.
  Treet krymper dermed med ett nivå.
- Vi må, om nødvendig, huske på å oppdatere nøkkelverdiene på stien fra roten til B.

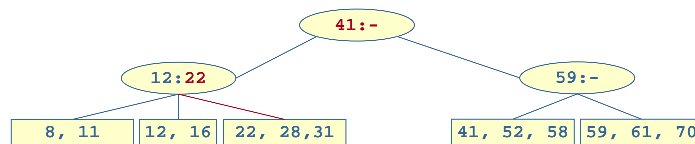
Eksempel: Ta utgangspunkt i følgende tre:



Slett elementet '17' :



Slett elementet '23':



Tidsforbruk

- Siden hver interne node, untatt rotnoden, har minst $\lceil M/2 \rceil$ barn, er dybden til et B-tre maksimalt $\lceil \log_{\lceil M/2 \rceil} N \rceil$.
- På hver node fra roten til bladnoden må vi utføre $O(\log M)$ arbeid (ved binærsøk i sortert array) for å avgjøre hvilken gren vi skal gå.
- Dermed tar **find** $O(\log M \cdot \log_{\lceil M/2 \rceil} N) = O(\log N)$ tid (antar M og L omtrent like).
- Ved **insert** og **delete** kan det hende at vi må utføre $O(M)$ arbeid på hver node for å rydde opp (f.eks. flytte alle nøkkelverdiene i tabellen en plass til venstre).
- Så **insert** og **delete** kan ta $O(\log M \cdot \log_{\lceil M/2 \rceil} N) = O((M/\log M) \cdot \log N)$ tid (antar M og L omtrent like).

Hvor stor skal M være?

Hvor mange barn skal en node få lov å ha?

- Hvis hele B-treet får plass i internminnet, har empiriske målinger vist at $M = 3$ og $M = 4$ er de beste valgene (innsetting og sletting tar for lang tid hvis M blir for stor).
- Men B-trær har sin store styrke når ikke hele treet får plass i internminnet.
- Siden en diskoperasjon tar nesten 1 000 000 ganger mer tid enn en operasjon i internminnet, gjelder det å minimalisere antall diskaksesser.
- Hvis det er plass, er det en god idé å lagre alle internnoder i internminnet og alle bladnoder på harddisk.
- Da kan man velge $M = 4$ og L så stor at hver bladnode fyller en diskblokk (eventuelt et disk cluster).

Hvor stor skal M være når hele treet ligger på disk?

- Treet blir bredere og får mindre dybde desto større M er.
- Mindre dybde betyr færre diskaksesser, mens vi kan se bort fra det ekstra oppryddingsarbeidet i nodene som en stor M medfører fordi det foregår i internminnet.
- I praksis velger man M så stor at en internnode fortsatt får plass på én diskblokk (eller cluster), typisk i området $32 \leq M \leq 256$.
- Man velger L slik at det samme gjelder for bladnodene.
- Analyser viser at B-trær blir 69% fulle (dvs. $\ln 2$, samme fyllingsgrad som utvidbar hashing).

- Vi skal avslutte B-trær (kap. 4.7)
- Vi skal introdusere prioritetskøer eller hauger (”heaps” på engelsk, kap. 6)