UNIVERSITETET I OSLO

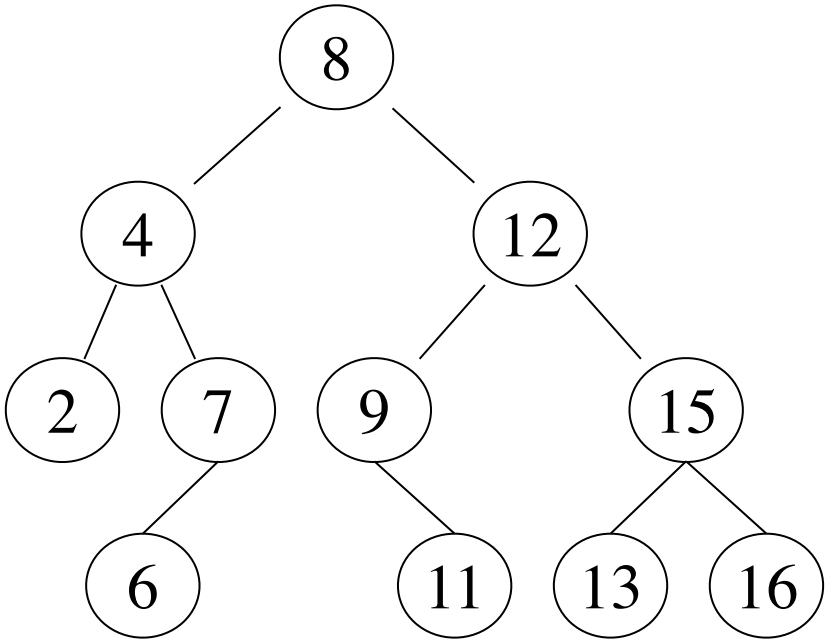
INF2220: Forelesning 2

Balanserte søketrær

- Rød-svarte trær (kapittel12.2)
- B-trær (kapittel 4.7)

REPETISJON:
BINÆRE SØKETRÆR

Binære søketrær



For enhver node i et binært søketre gjelder:

- Alle verdiene i venstre subtre er mindre enn verdien i noden selv.
- Alle verdiene i høyre subtre er større enn verdien i noden selv.

Binære søketrær

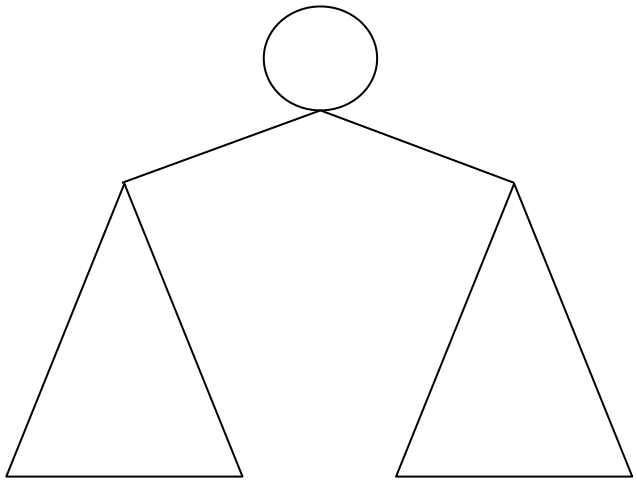
Metode for å sjekke om et binærtre er et binært søketre (antar unike int-elementer):

```
// Kall: bst(rot, -UENDELIG, +UENDELIG)

boolean bst(BinNode n, int min, int max) {
    if (n == null)
        return true;

    if (n.element < min || n.element > max)
        return false;

    return ( bst(n.venstre, min, n.element - 1) &&
             bst(n.hoyre, n.element + 1, max) );
}
```

Gjennomsnitts-analyse

Intuitivt forventer vi at alle operasjonene som utføres på et binært søketre vil ta O(log n) tid siden vi hele tiden grovt sett halverer størrelsen på treet vi jobber med.

Det kan bevises at den gjennomsnittlige dybden til nodene i treet er O(log n) når alle innsettingsrekkefølger er like sannsynlige (se kap. 4.3.5).

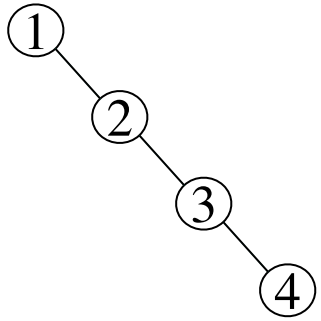


Oppgave 4.5

Vis at et binærtre med høyde h har maksimalt $2^{h+1} – 1$ noder.

Worst-case analyse

I verste fall brukes bare venstre- (eller høyre-) pekerne i treet, og det binære søketreet blir i praksis lik en enkel liste.



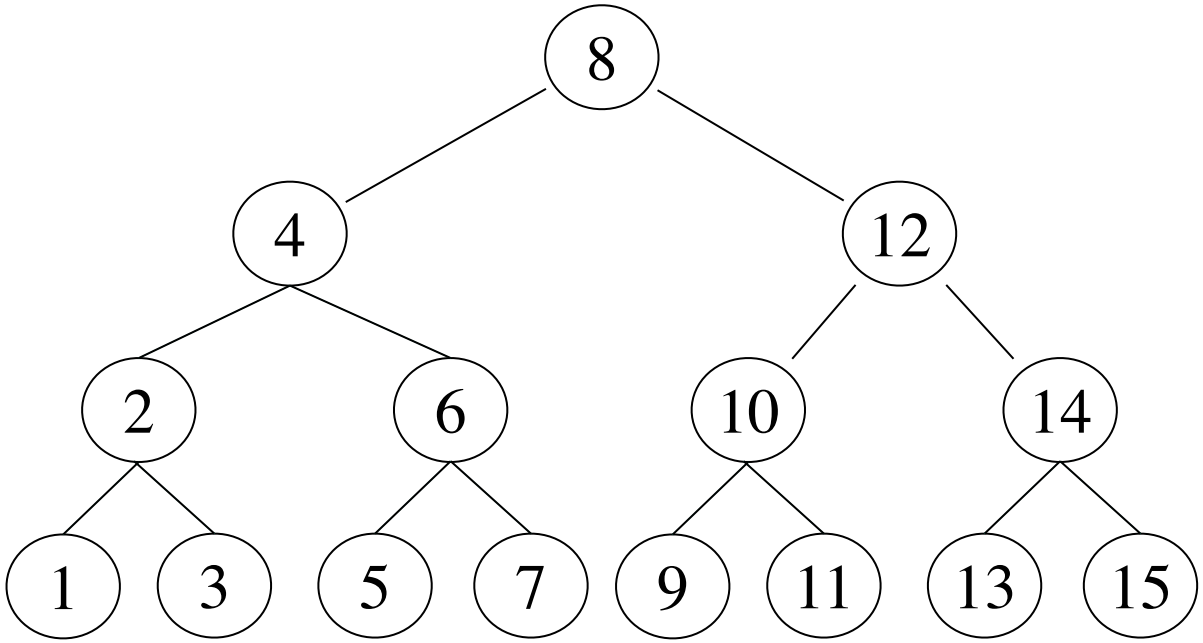
Vi får da worst-case O(n) tid for innsetting, søking, sletting osv.

Ubalanserte trær kan for eksempel skyldes:

- spesiell innsettingsrekkefølge (som i Vildanden)
- ujevn sletting

Balanserte trær

Eksempel på perfekt balansert tre:



Balanserte trær

- Det finnes ulike mekanismer for å sikre mest mulige balanserte trær. Felles er at de er basert på en invariant.
  - AVL-trær: For alle indre noder er høydeforskjellen mellom barna max 1.
- Utfordringen er å opprettholde invarianten under innsetting og sletting. Generell metode:
  - Gjør innsetting/sletting som vanlig.
  - Gjenopprett balansen ved rotasjoner som restrukturerer treet.
- Vi skal se på rød-svarte trær.

RØD-SVARTE TRÆR

Rød-svarte trær

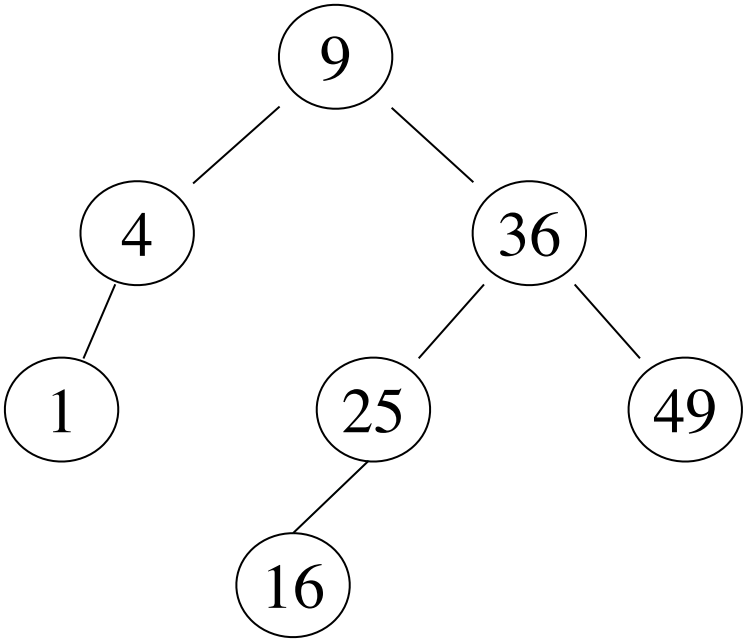
Et rød-svart tre er et binært søketre der hver node er farget enten rød eller svart slik at:

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 $2*\log_2(N+1)$!

Oppgave

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

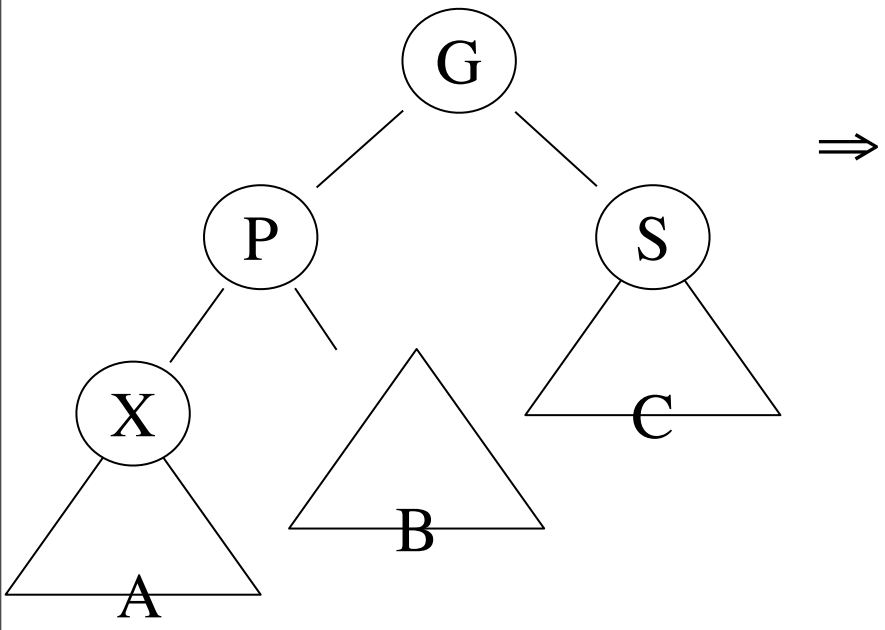


- Sett inn tallet 64 på riktig plass og med riktig farge.
- Forsøk så å sette inn tallet 81.

Innsetting i rød-svart tre

- Innsetting er ikke noe problem hvis forelder-noden er svart:
  - Den nye noden kan settes inn som rød, og antall svarte noder på veien til null-pekerne blir som før.
- Hvis forelder-noden er rød:
  - Den nye noden kan ikke være rød (krav 2).
  - Den nye noden kan ikke være svart (krav 3).
  - Treet må endres ved hjelp av rotasjoner og omfarging.

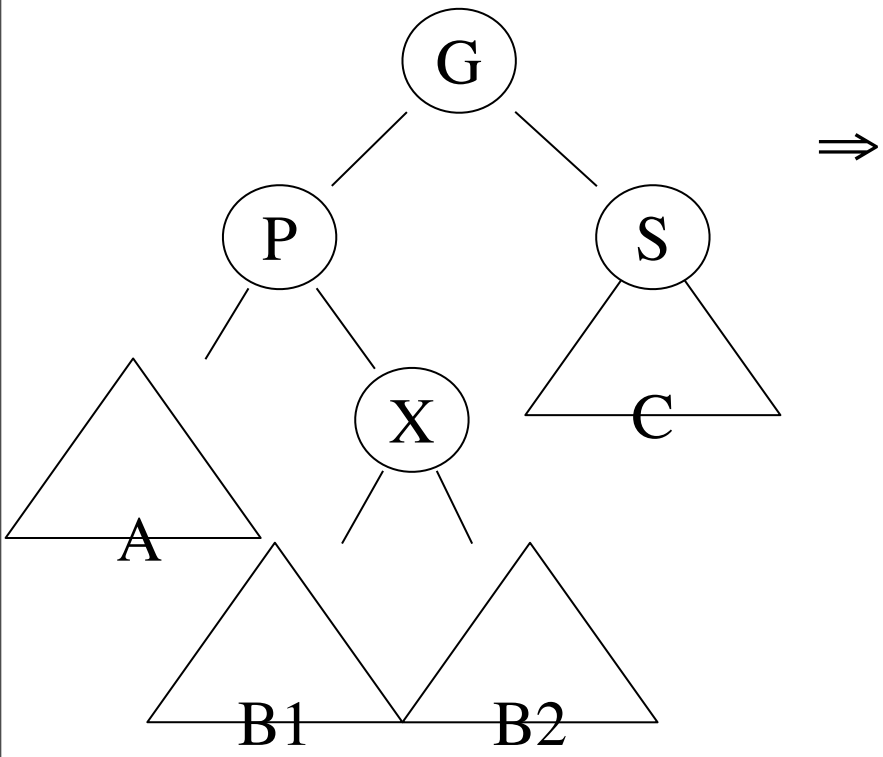
Zig rotasjon



$\Rightarrow$

+ symmetrisk tilfelle…

Zig-zag rotasjon



$\Rightarrow$

+ symmetrisk tilfelle…

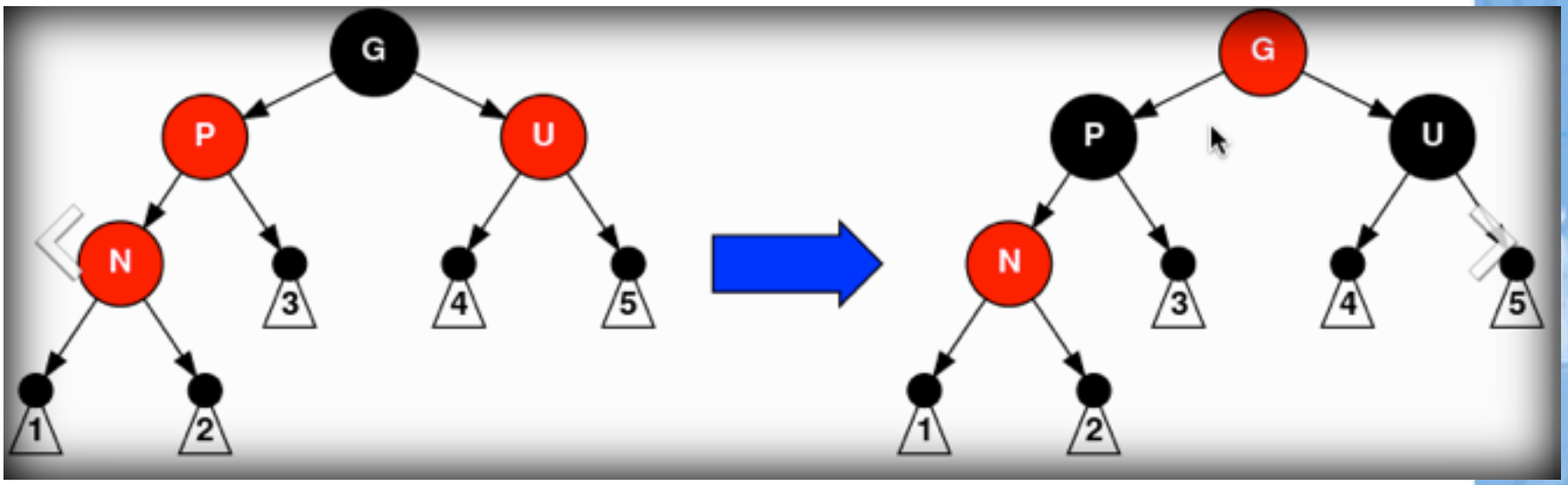
Innsetting i rød-svart tre

1. Gjør innsetting som i vanlig binært søketre, der den nye noden N farges rød.
2. La P og G være forelder og besteforelder til N, og U søsken til P.
3. Hvis P er svart: Alt ok, innsetting ferdig.
4. Hvis P er rød:
   a. Hvis N og P begge er venstre (høyre) barn:
      Gjør zig rotasjon med nødvendige fargeendringer.
   b. Hvis N er venstre og P høyre barn eller motsatt:
      Gjør zig-zag rotasjon med nødvendige fargeendringer.
   c. Sett N til å være den nye roten i det roterte subtreet.
   d. Hvis N nå er roten i selve treet: Farg denne svart.
      Ellers: Gjenta fra steg 2.

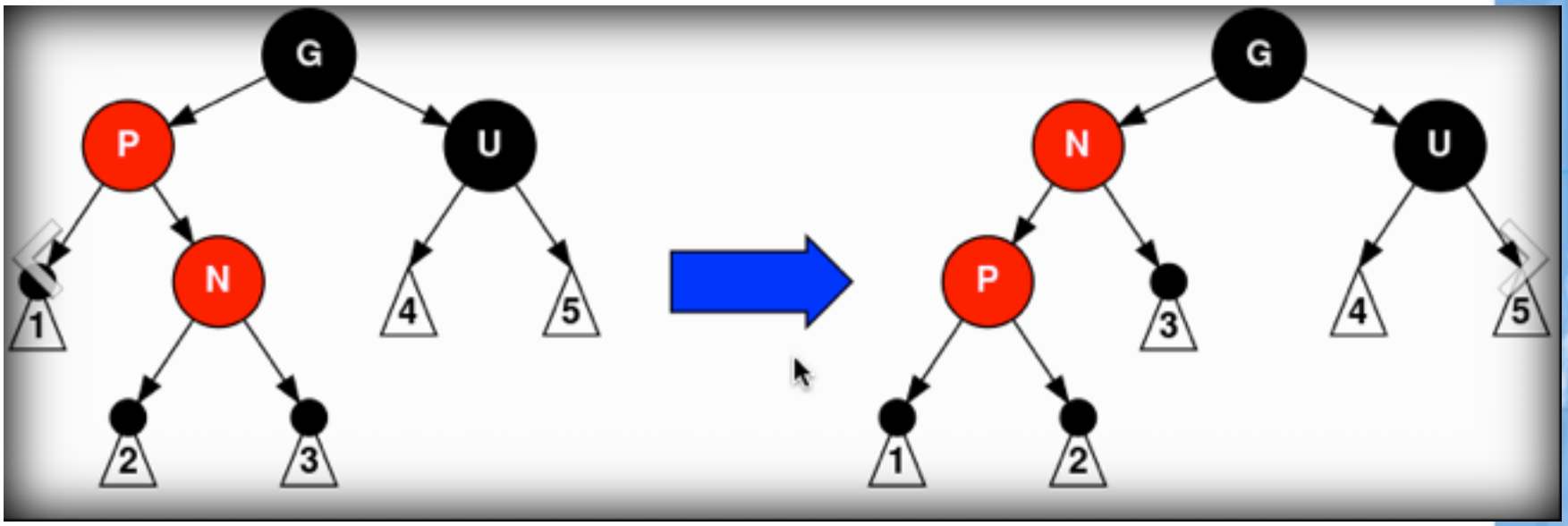
Case 1: The current node N is at the root of the tree

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

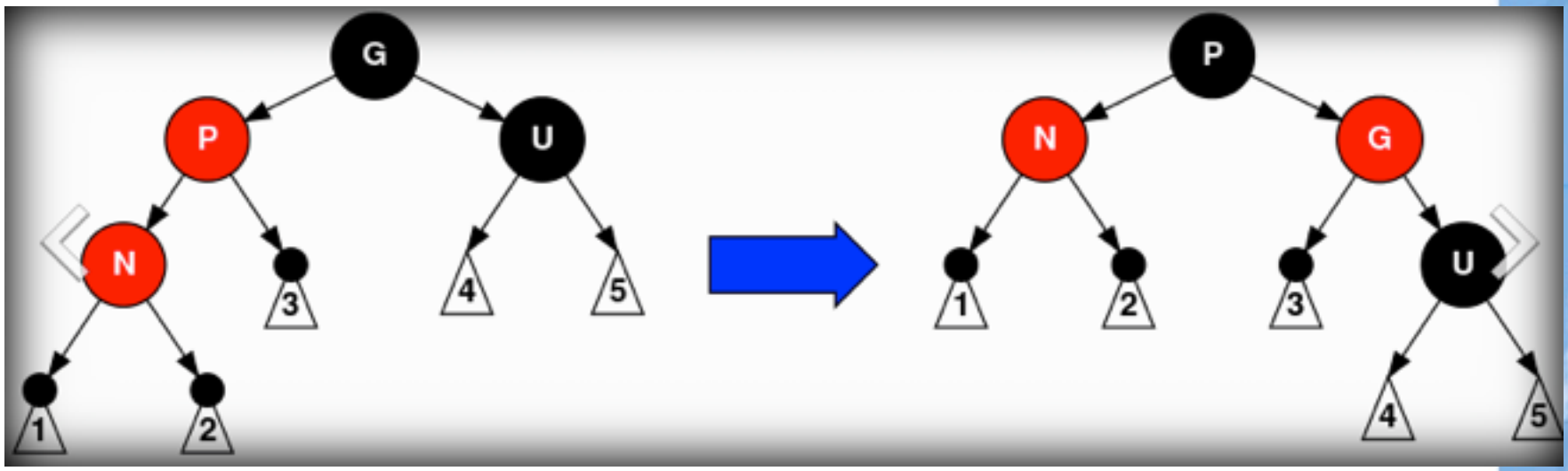
Case 3: Both the parent P and the uncle U are red



Case 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.



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.



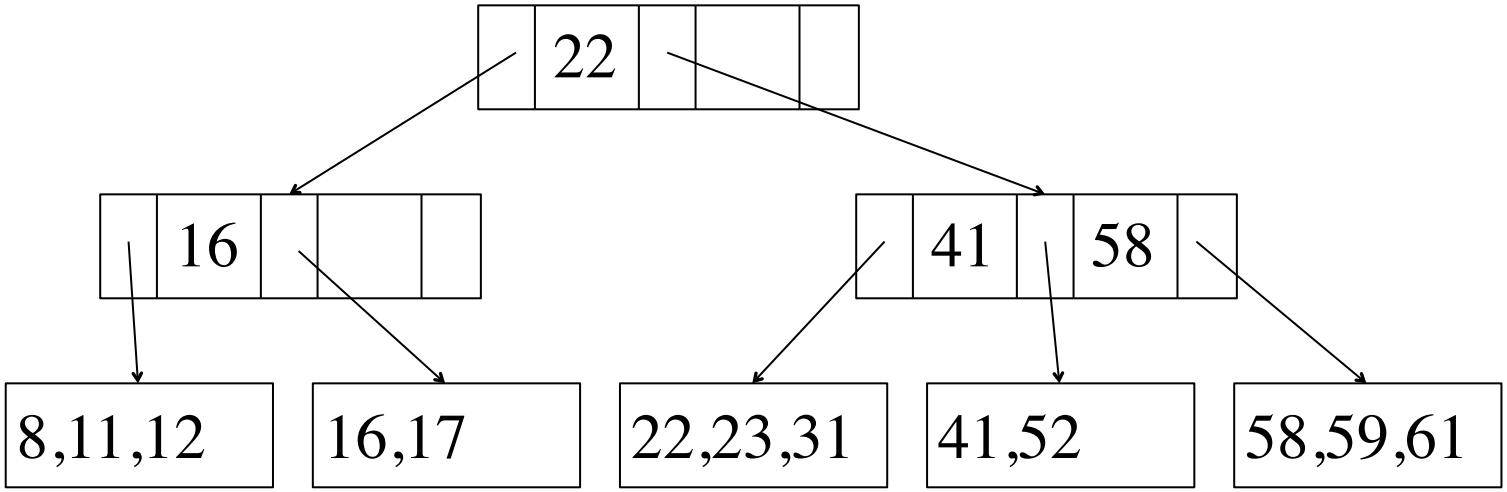
B-TRÆR

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

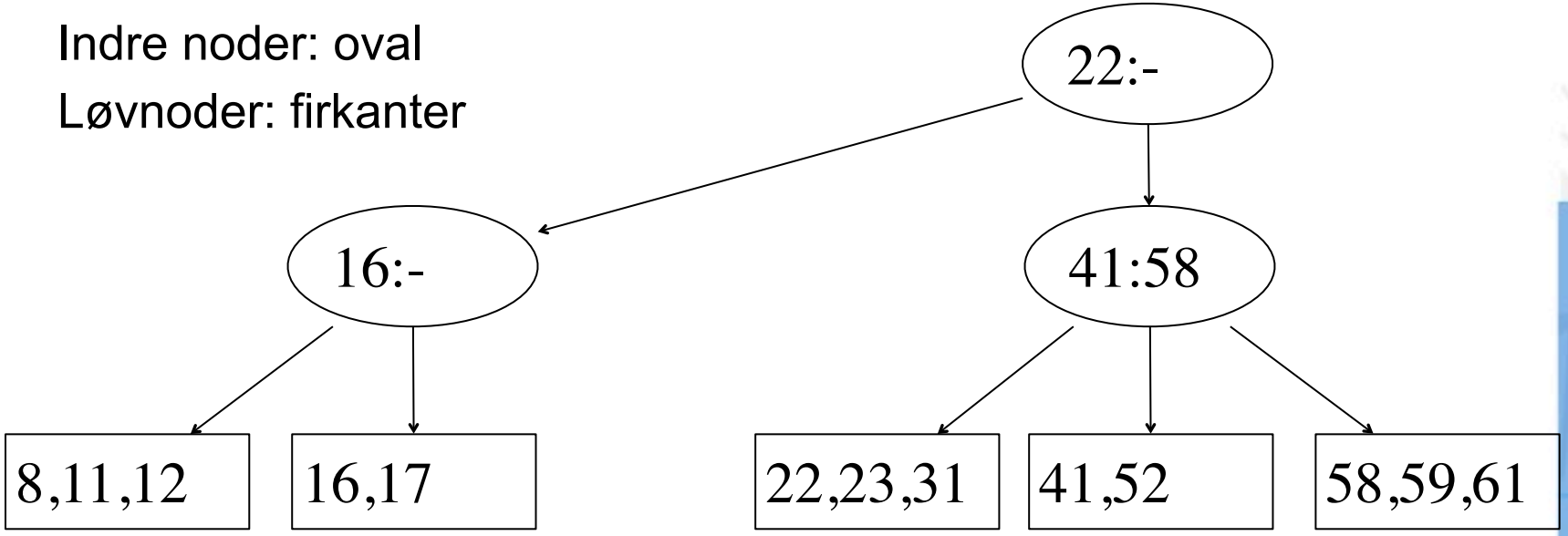


Dette treet har M=3 pekere fra hver indre node og L=3 dataelementer i hver løvnode.

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

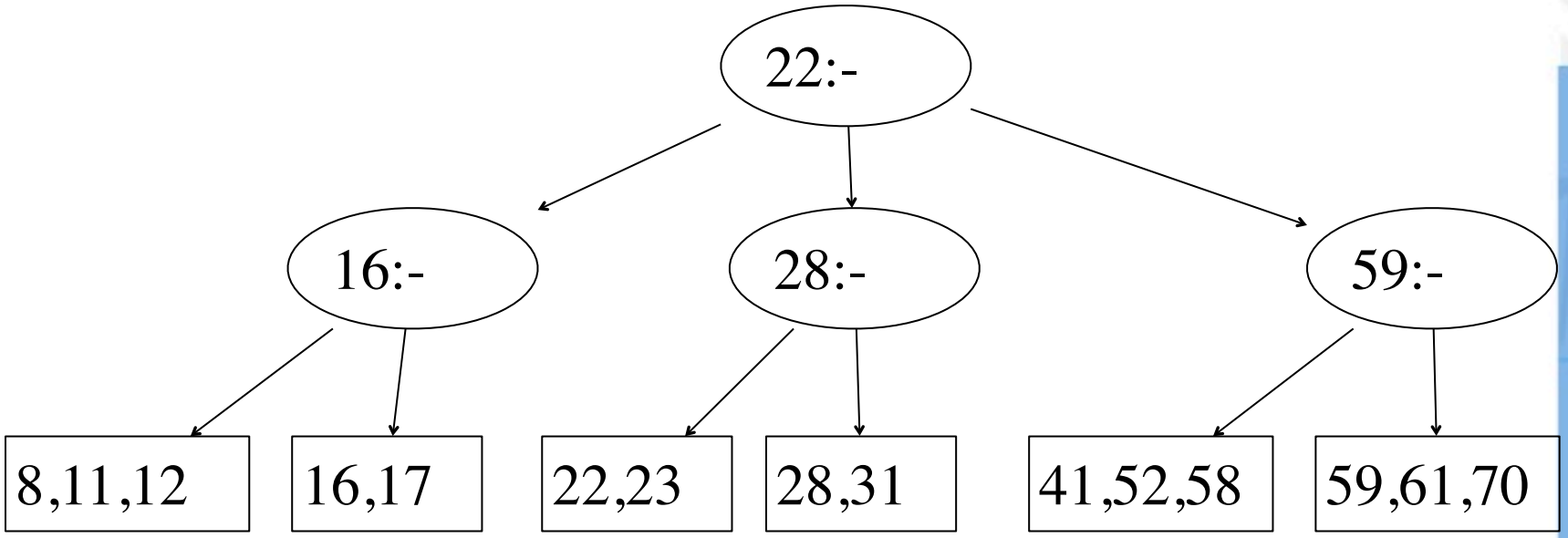
1. Start i roten.
2. Så lenge vi ikke er i en løvnode:
   La nøkkel-verdiene bestemme hvilket barn vi skal gå til.
3. Let etter x i løvnoden.

Innsetting av element x

1. Let etter riktig løvnode for x (som for søking).
2. Dersom det er plass, sett inn x og oppdater nøkkel-verdiene 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!

Sletting: eksempel

Fjern først 17, deretter 23 fra dette treet:



Sletting av element x

1. Finn riktig løvnode N for x ved søking.
2. Hvis N har minst $\lceil L/2+1 \rceil$ elementer, kan vi enkelt slette x.
3. Hvis ikke, må N kombineres med en av nabosøsknene:
   a. Hvis venstre (høyre) søsken har minst $\lceil L/2+1 \rceil$ elementer, flytter vi det største (minste) elementet over til N.
   b. Hvis søsken har akkurat $\lceil L/2 \rceil$ elementer, slår vi de to nodene sammen til en node (med L eller L-1 elementer).
4. Gjenta punkt 3 dersom foreldernoden nå har ett barn.
5. Til slutt: Hvis roten bare har ett barn, slettes denne og barnet blir ny rot.
   (Treet krymper nå ett nivå.)
6. Husk å oppdatere nøkkelverdiene underveis!

Tidsforbruk

- Vi antar at M og L er omtrent like.
- Siden hver indre node unntatt roten har minst $\lceil M/2 \rceil$ barn, er dybden til B-treet maksimalt $\lceil \log_{\lceil M/2 \rceil} N \rceil$.
- For hver node må vi utføre O(log M) arbeid (binærsøk) for å avgjøre hvilken gren vi skal gå til.
- Dermed tar søking $O(\log M * \log_{M/2} N) = O(\log N)$ tid.
- Ved innsetting og sletting kan det hende at vi må gjøre O(M) arbeid på hver node for å rydde opp (for eksempel flytte alle nøkkelverdiene i tabellen en plass til venstre).
- Innsetting og sletting kan dermed ta $O(M \log_{M/2} N) = O((M/\log M) \log N)$ tid.

Hvor stor bør M være?

Hvor mange barn skal en indre node ha lov til å ha?

Try algorithm visualizations

HERE

Neste forelesning: 7. september

MAPS OG HASHING