Forskeruddannelsen, Del A
Kvalifikationseksamen d. 24/11-1994.

Gerth Stølting Brodal

"Complexity of Data Structures"

"Der ønskes en præsentation af rapportens centrale resultat, $O(1)$ worst case kompleksisteten for partielt persistente datastrukturer, med vægt på intuitionen bag konstruktionen og dens mulige anvendelser i praksis.

Dernæst ønskes der en fremstilling af problemkredsen omkring fuld persistens, herunder relationerne til fractional cascading"

Disposition

- Partielt Persistente Datastrukturer
- "Planar Point Location"
- "Path Copying"
- "Fat Node"
- "Node Copying"
- The pebble game
- Worst case $O(1)$ opdatering
- Fuldt persistente datastrukturer
- "Fractional Cascading"

Partielt Persistente Datastrukturer

En **partielt persistent** datastruktur er en datastruktur, hvor gamle versioner af datastrukturen huskes, så man senere kan søge i dem.

Den **flygtige** datastruktur er den underliggende datastruktur, der gøres persistent.

Vi vil kræve, at datastrukturer er beskrevet således, at de kan implementeres på en **pointer maskine**. Dvs. en datastruktur er en orienteret graf, hvor antallet af datafelter i hver knude er begrænset af en konstant. Et datafelt kan indeholde data eller en pointer til en anden knude.

Eksempel: Træer

Operationer

Følgende operationer skal understøttes af en persistent datastruktur:

- $\textsc{Read}(v_i, f)$ returnerer indholdet af datafeltet f i den persistente knude givet ved v_i, hvor i angiver, at det er værdien til tid i, vi ønsker.

  Hvis indholdet er en pointer til en knude u, returneres en reference til u_i.

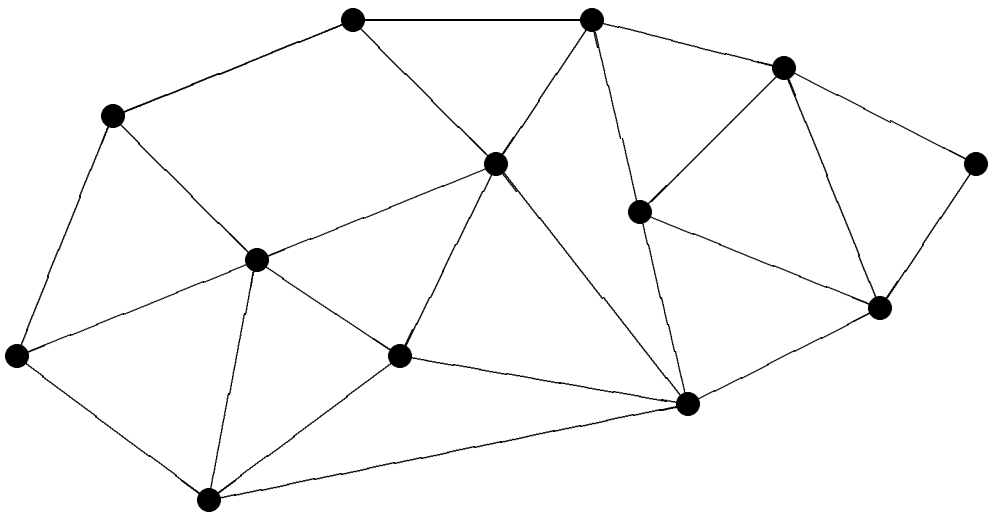
- $\textsc{Write}(v_i, f, value)$ skriver værdien $value$ i knude v's datafelt f. Vi kræver, at i er den aktuelle version af datastrukturen.

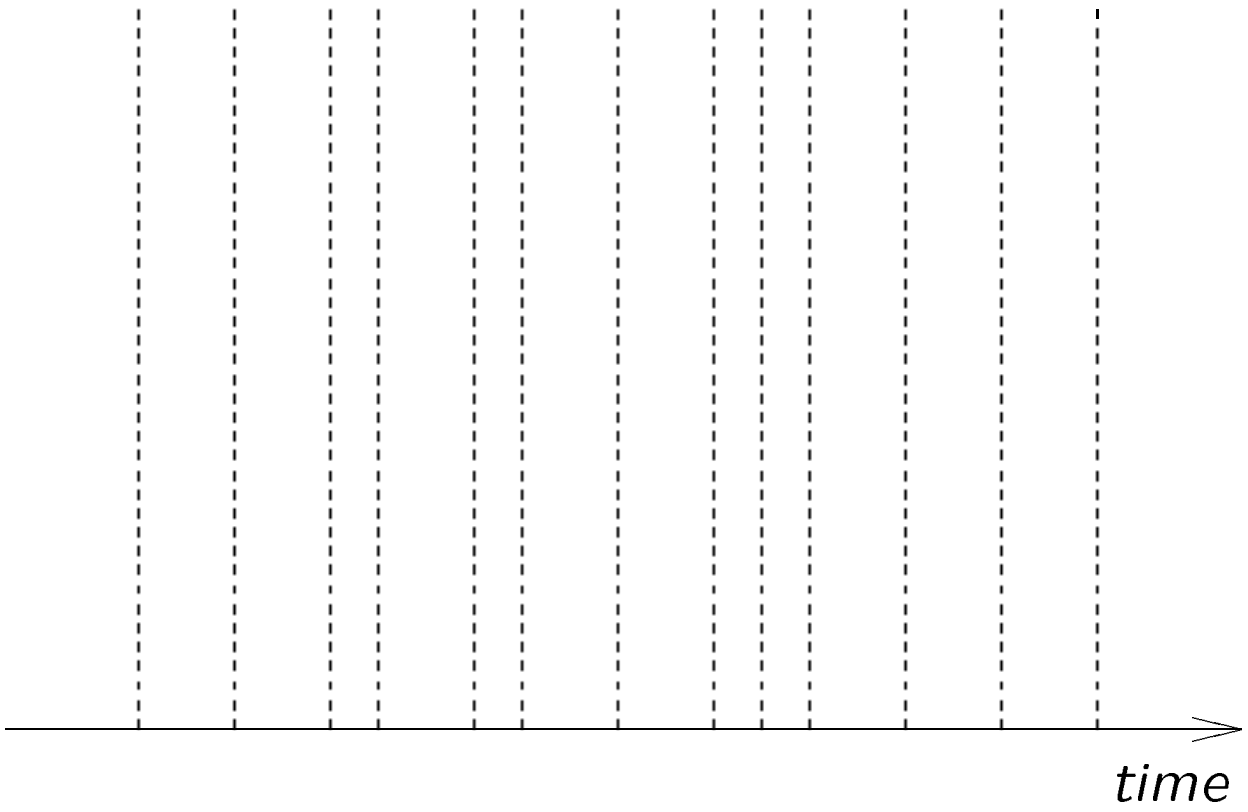
"Planar Point Location"

[Sarnak, Tarjan 86]

Givet en opdeling af planen i polygoner, ønsker vi at kunne besvare spørgsmål på formen:

- Givet (x, y), hvilket område indeholder (x, y)?

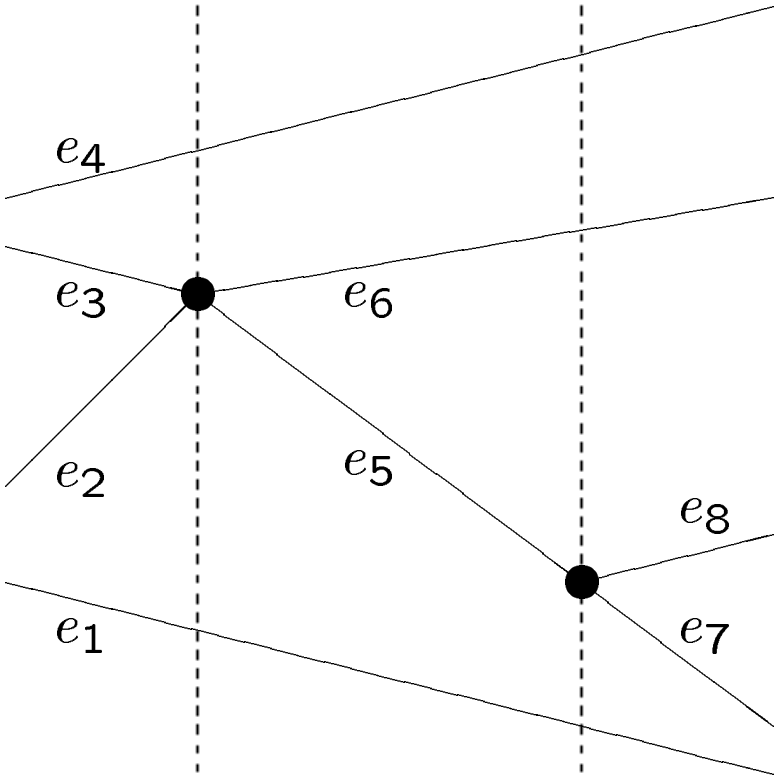



time

NOTE: Planar Point Location

- Liniestykkerne er givet på passende form
- Hver linie kender navnet på områderne på hver side af linien
- Linierne krydser ikke
- Sweep-line algoritme — Generelt?

Planar Point Location (*fortsat*)



$\{e_1, e_2, e_3, e_4\}$ $\{e_1, e_5, e_6, e_4\}$ $\{e_1, e_7, e_8, e_6, e_4\}$

Planar Point Location (*fortsat*)

Præprocessering:

- Sortér endepunkterne mht. x-koordinaten
- Anvend et partielt persistent søgetræ til at opbevare liniestykkerne
- Foretag et sweep igennem grafen, hvor liniestykker indsættes og fjernes fra træet
- "Udførselstid": $O(n \log n)$

Forespørgsel:

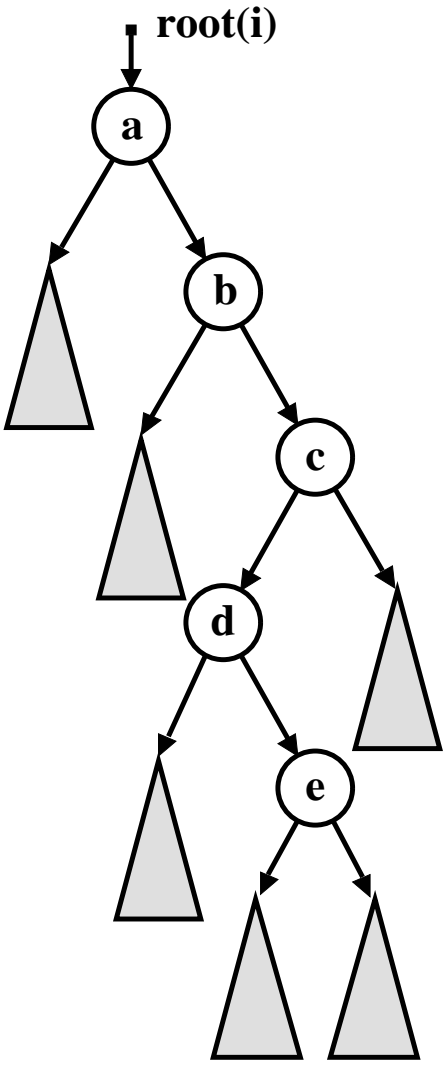
- Find den version af træet, der indeholder (x, y)
- Find de to liniestykker, som (x, y) ligger imellem i den fundne version af træet
- "Udførselstid": $O(\log n)$

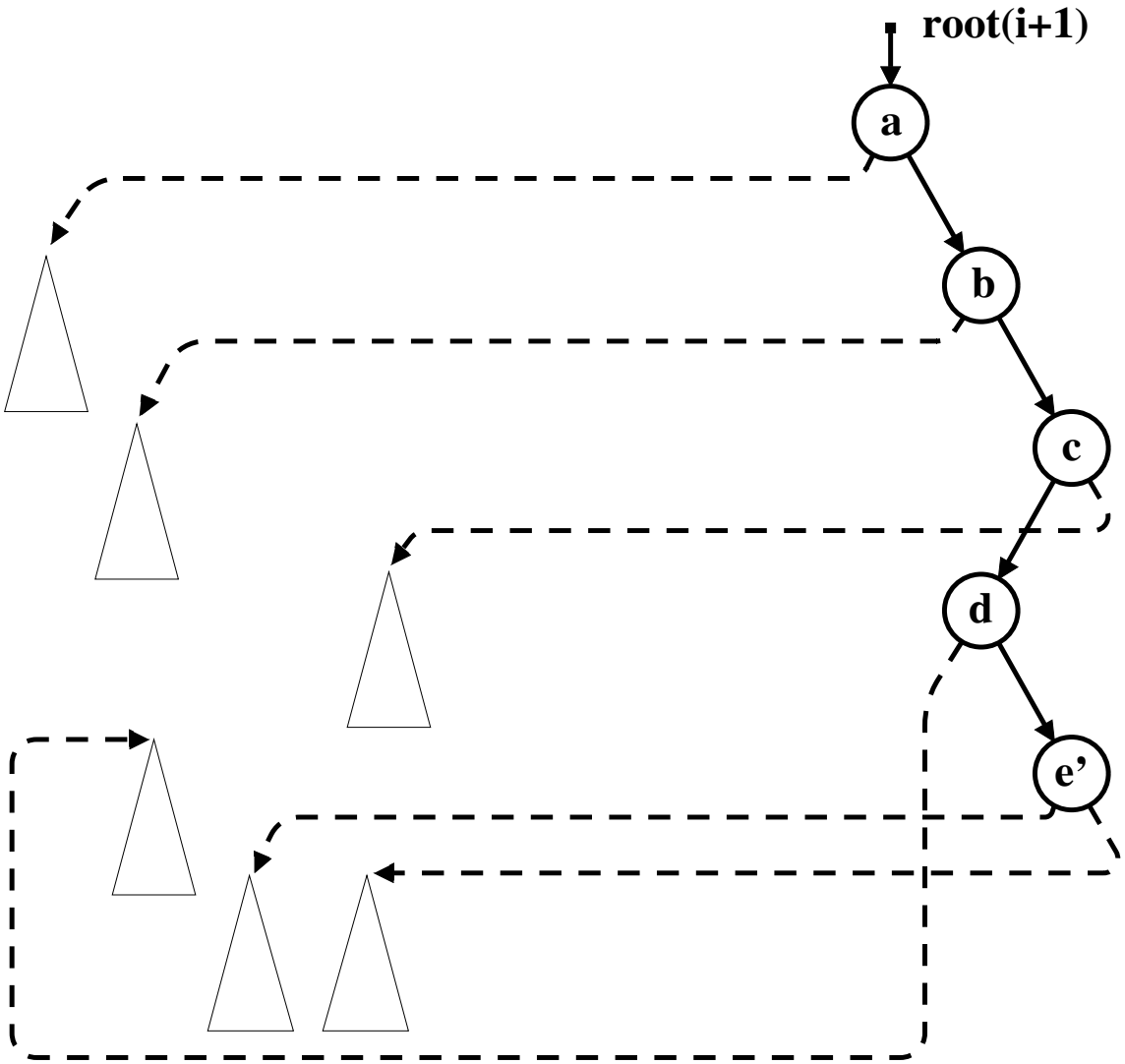
NOTE: Planar Point Location

De nævnte udførselstider gælder kun hvis vi effektivt kan gøre et søgetræ partielt persitent.

“Path Copying”

Den simpleste måde at gøre træer persistente er ved anvendelse af **path copying**.

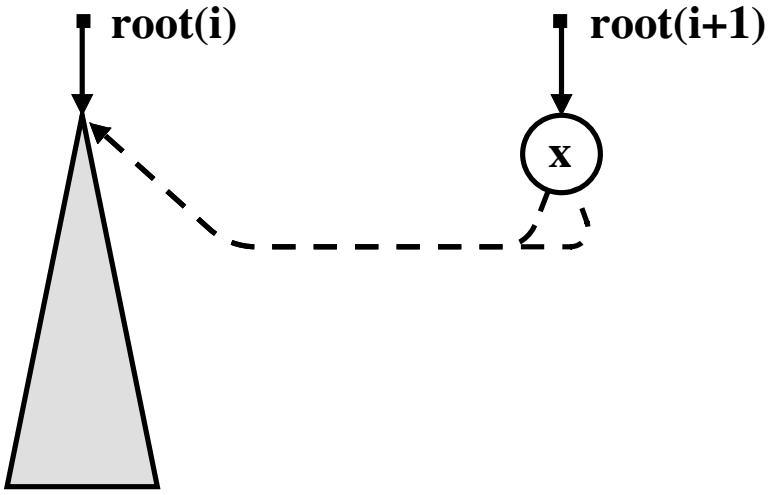



root(i+1)
a
b
c
d
e'

Path Copying (*fortsat*)

Fordele:

- Simpel at implementere
- Man kan kombinere flere forskellige versioner af et træ til et nyt træ (**confluently persistence**)



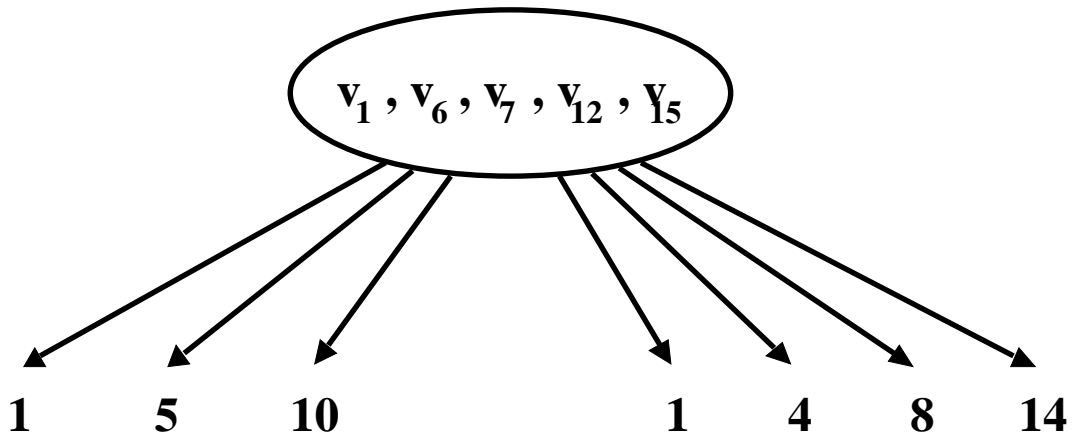
Ulemper:

- Kræver plads proportional med dybden af den foretagne ændring
- Virker kun for træer

“Fat Node”

Datastrukturer kan gøres partielt persistente ved “Fat Node” teknikken. En knude indeholder **alle** de informationer den i tidens løb har indeholdt.

Enhver pointer og værdi er tilknyttet et **tidsstempel**, som angiver, hvornår ændringen er foretaget.



Fat Node (*fortsat*)

Fordele:

- Simpel at implementere
- Kræver $O(1)$ plads per ændring
- Kan anvendes på alle datastrukturer

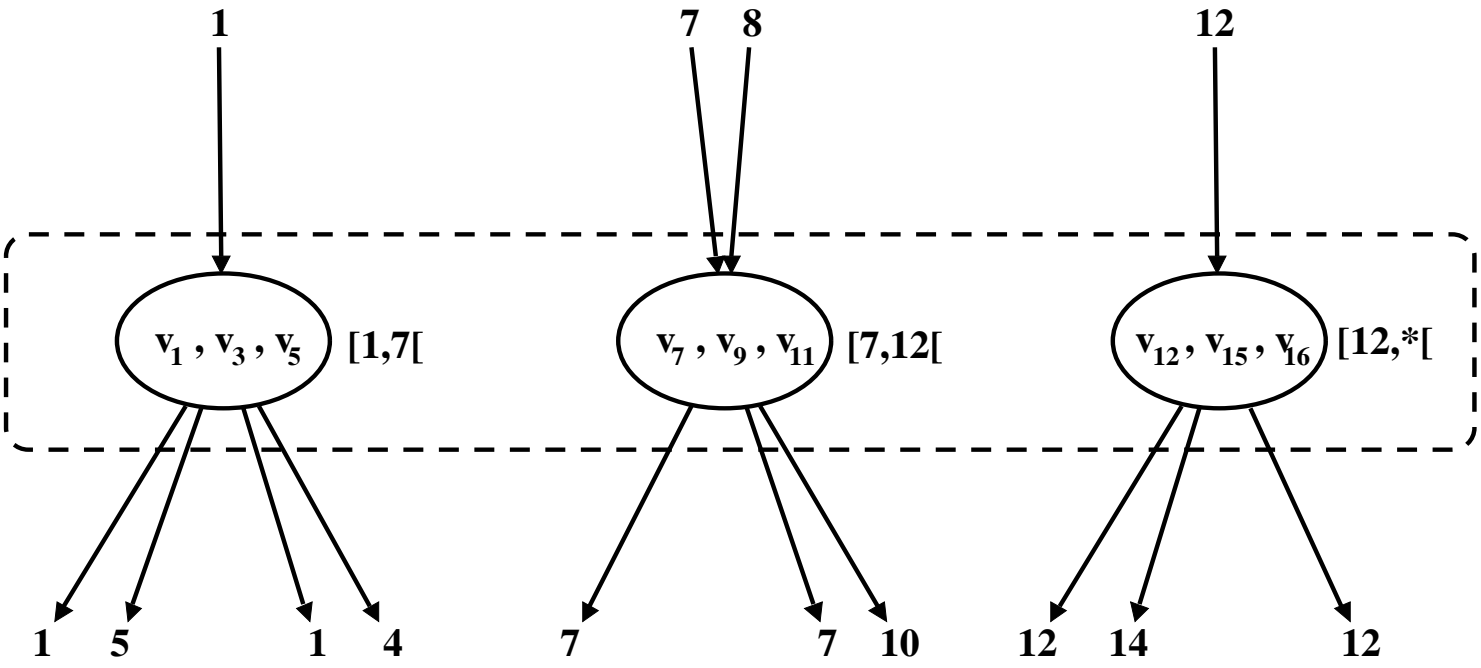
Ulemper:

- Indholdet i en knude kan blive vilkårligt stort
- Der introduceres et **slowdown** på $\omega(1)$ i forhold til den flygtige datastruktur

"Node Copying"

[Driscoll, Sarnak, Sleator, Tarjan 89]

En knude i den flygtige datastruktur er i den persistente datastruktur repræsenteret ved en **familie** af knuder.

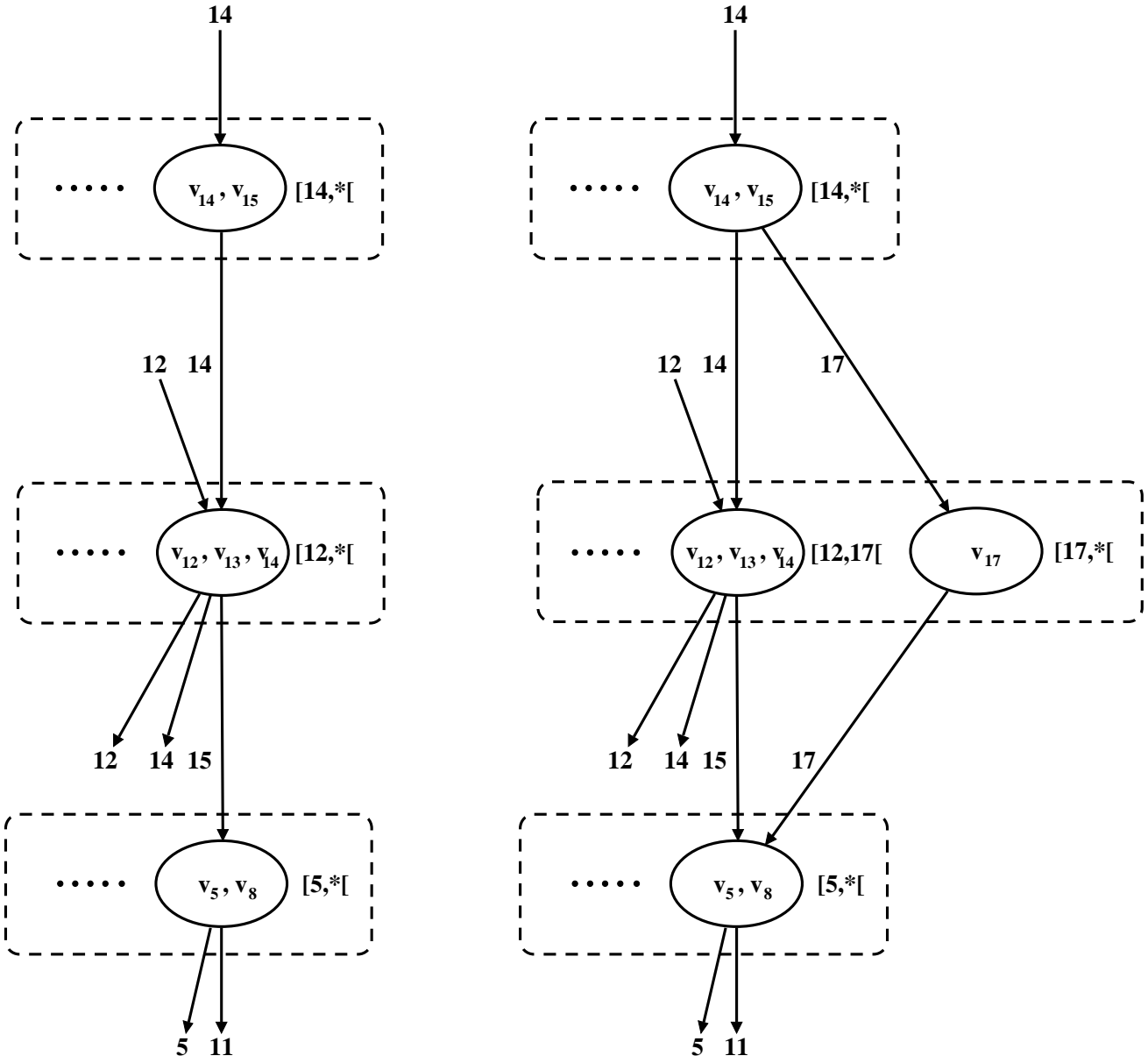


Eksempel på en familie af knuder

Node Copying (*fortsat*)

- Ændringer tilføjes til den sidste knude i familien med det aktuelle tidsstempel.

- COPY(v) tilføjer en knude til familien af knuder, der repræsenterer v og opdaterer de nødvendige pointere.



Node Copying (*fortsat*)

- **Målet**: Begræns mængden af information i hver knude med en konstant

- **Midlet**: Anvend operationen $\textsc{Copy}$

- **Metoden**:

  while $\exists v : P(v) \geq \tau$ **do** $\textsc{Copy}(v)$ **od**

  hvor:

  $P(v) =$ (Mængden af information i den sidste knude i familien, der repræsenterer v)$-d$

  $d =$ Antal datafelter i den flygtige datastruktur

  $\tau =$ En tærskelværdi, hvor $\tau \geq d + 1$

Node Copying (*fortsat*)

Fordele:

- Kræver **amortiseret** $O(1)$ plads og tid per ændring
- Søgninger har slowdown $\Theta(1)$

Ulemper:

- Giver kun en effektiv implementation for datastrukturer med **begrænset indgrad**
- Worst case udførslen for den enkelte ændring kan være $\Theta(n)$, hvor n er størrelsen af den aktuelle datastruktur

"Pebble Game"

[Dietz, Raman 91]

"Node Copying" idéen kan reformuleres i form af et "pebble" spil.

Spillet spilles på en **orienteret graf** $G = (V, E)$, hvor knuderne har begrænset udgrad (d) og indgrad (b). På hver knude v ligger der et antal "pebbles" $P(v)$.

To spillere skiftes til at foretage et træk:

- **I**: Lægger en pebble på en vilkårlig knude v
- **D**: Fjerner alle pebbles fra en knude v og lægger en pebble på alle forgængerne til v ($\textsc{Zero}(v)$)

NOTE: Pebble Game

- Hver knude svarer til en familie i node copying datastrukturen/en knude i den flygtige datastruktur

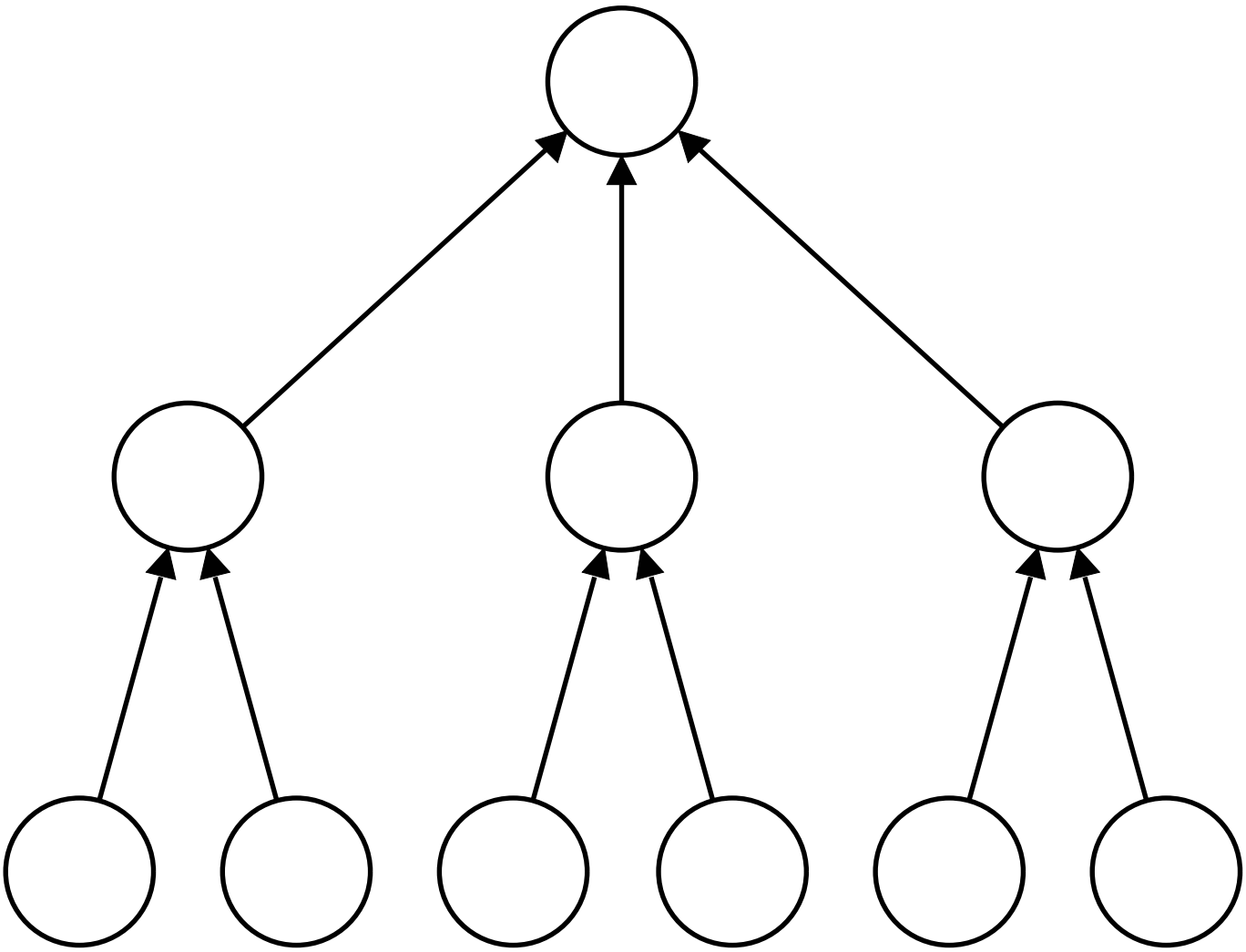
- Pebbles svarer til information i den sidste knude i familien

- Forklar trækkene vha. tegnestifter :-)

- Hvordan virker den traditionelle "node copying" strategi på eksemplet

- Kigger kun på det statiske tilfælde

Pebble Game (*fortsat*)



Pebble Game (*fortsat*)

- **Målet**:

  1. Find en strategi for **D**, der garanterer at antallet af pebbles på alle knuder er begrænset af en konstant

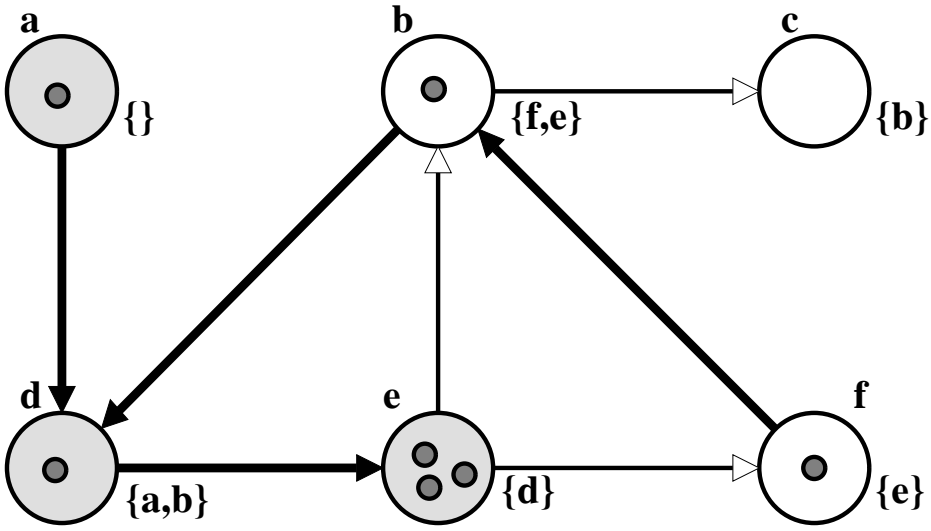
  2. Implementér strategien effektivt

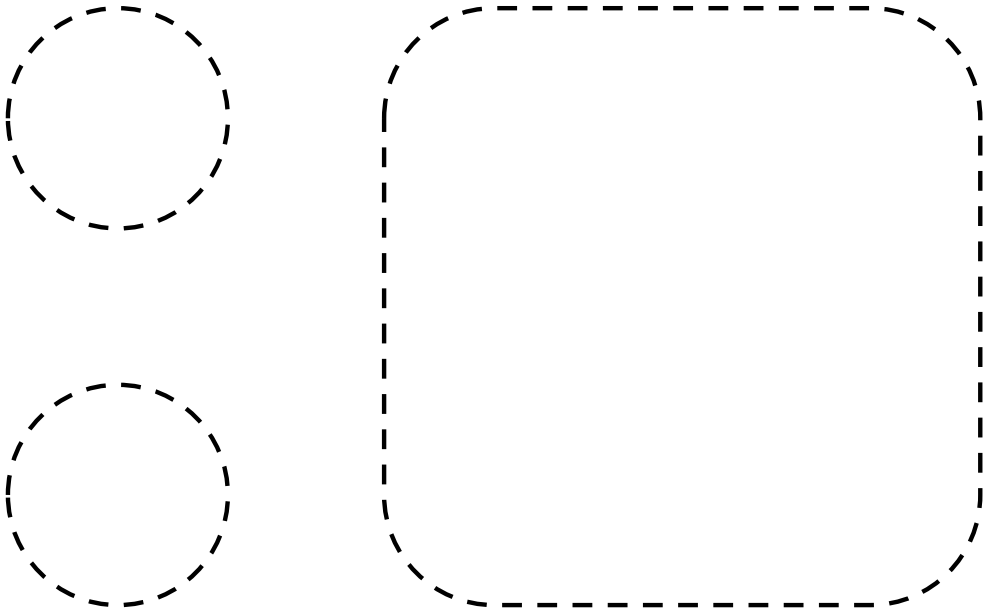
Information tilknyttet grafen

- Kanterne er **hvide** eller **sorte**. Hver knude har højst én indgående hvid kant. Der er ingen hvide cykler.

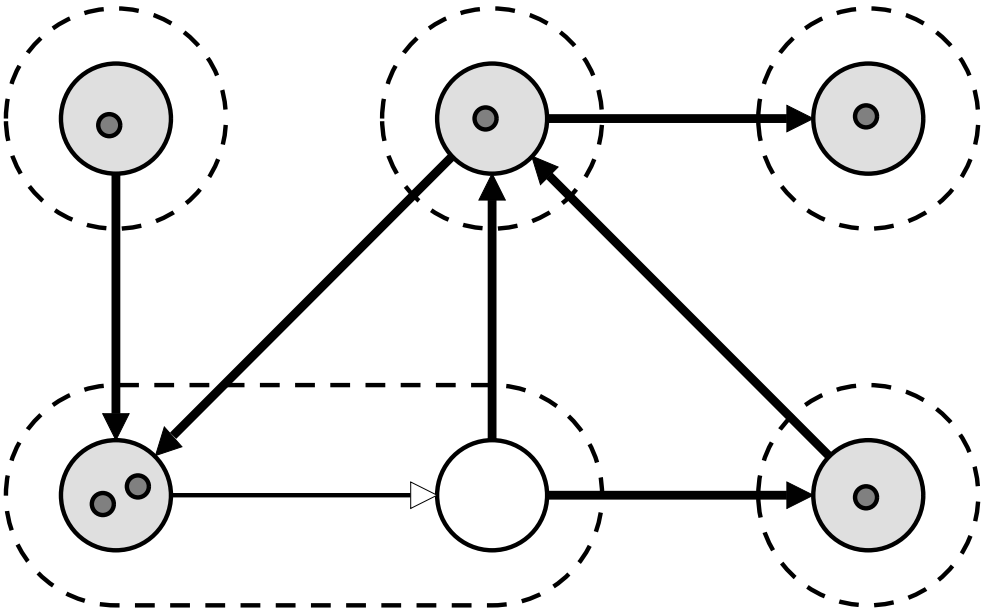
- Knuderne er **hvide** eller **sorte**. En knude er hvid, hviss den har en hvid indgående kant.

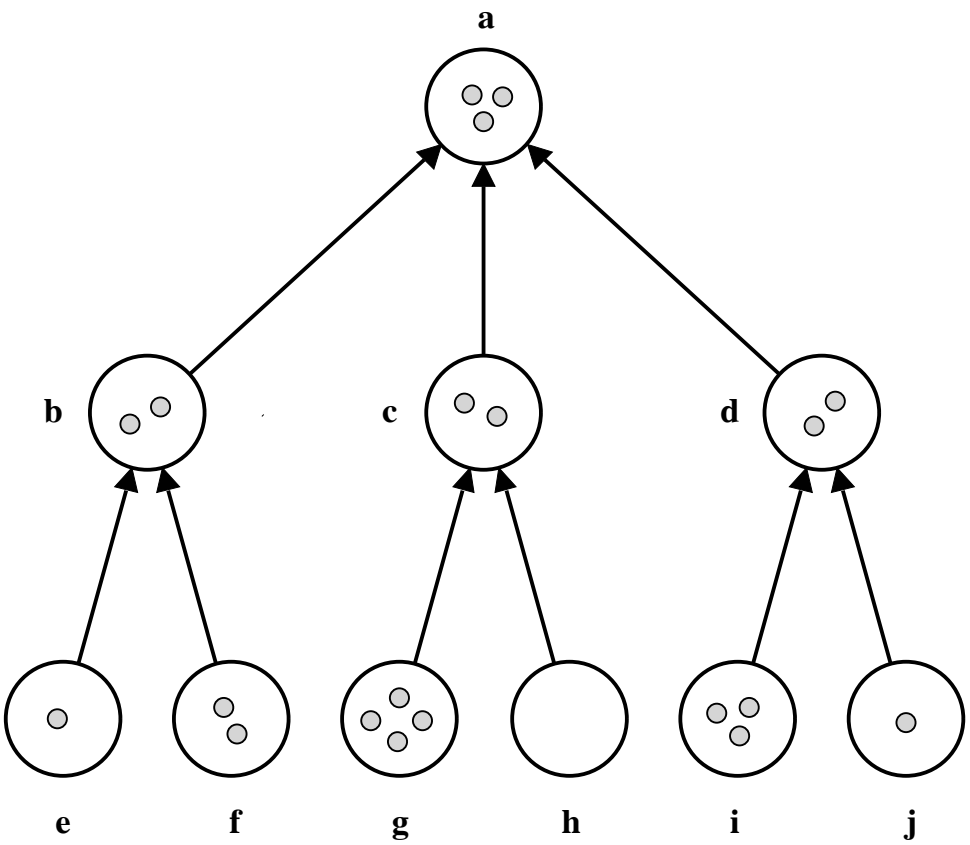
- Hver knude har en **kø** indeholdende dens forgængere.

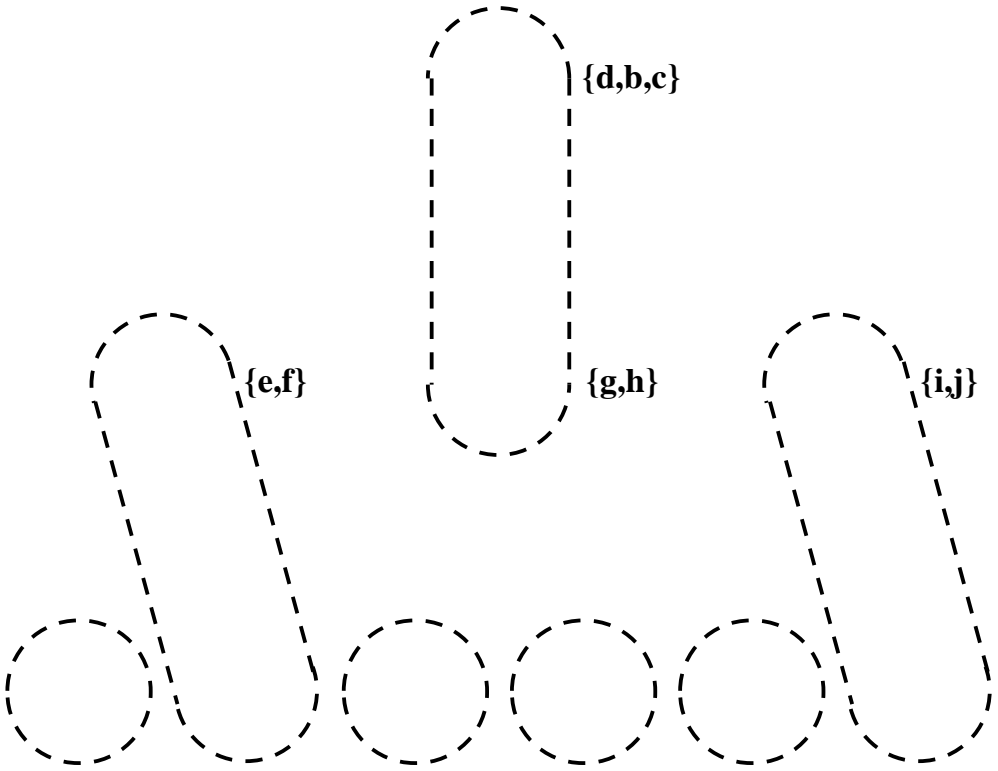

a
{}
b
{f,e}
c
{b}
d
{a,b}
e
{d}
f
{e}

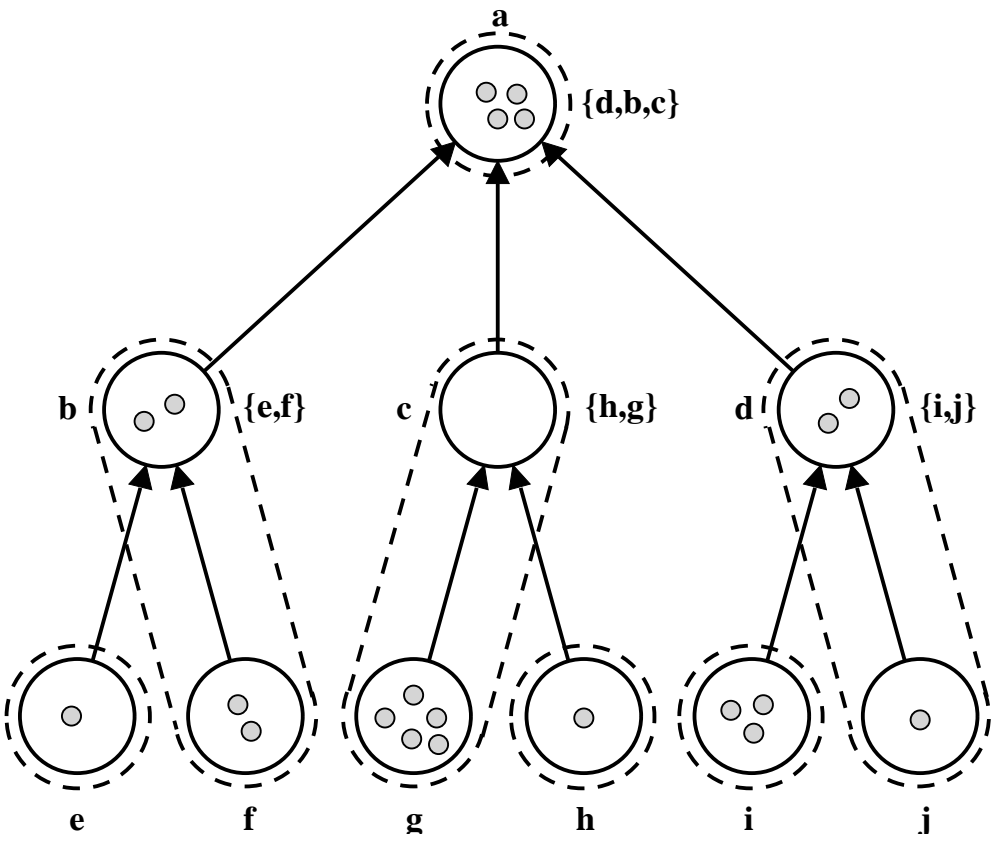


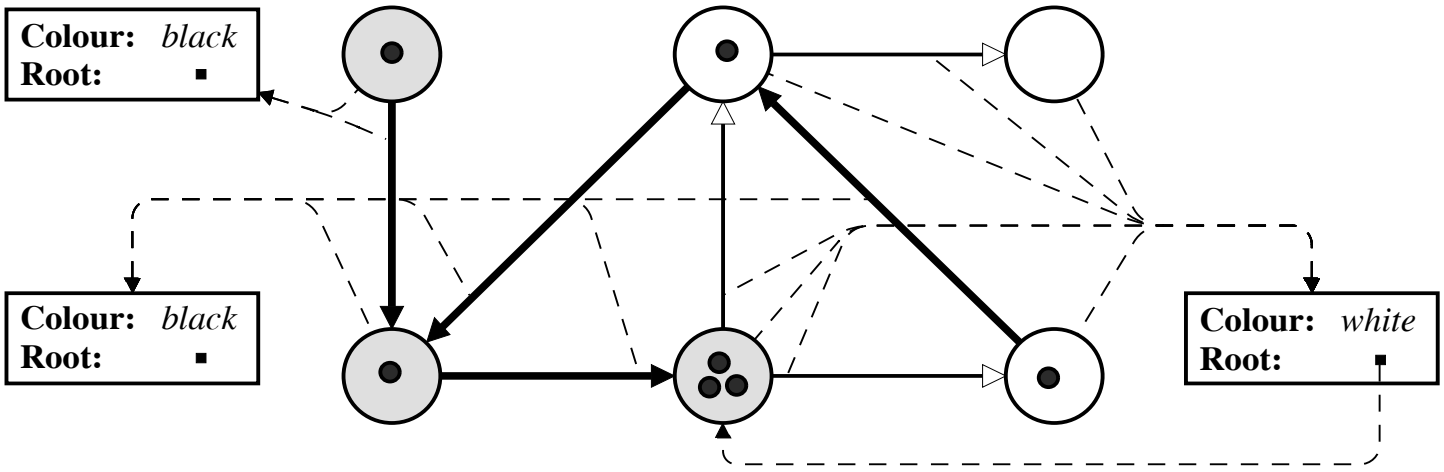
procedure BREAK(C_v)
    $r \leftarrow$ the root of C_v
    colour all nodes and edges in C_v black
    if $Q_r \neq \emptyset$ **then**
        colour r and (ROTATE(Q_r), r) white
    endif
    ZERO(r)
end.



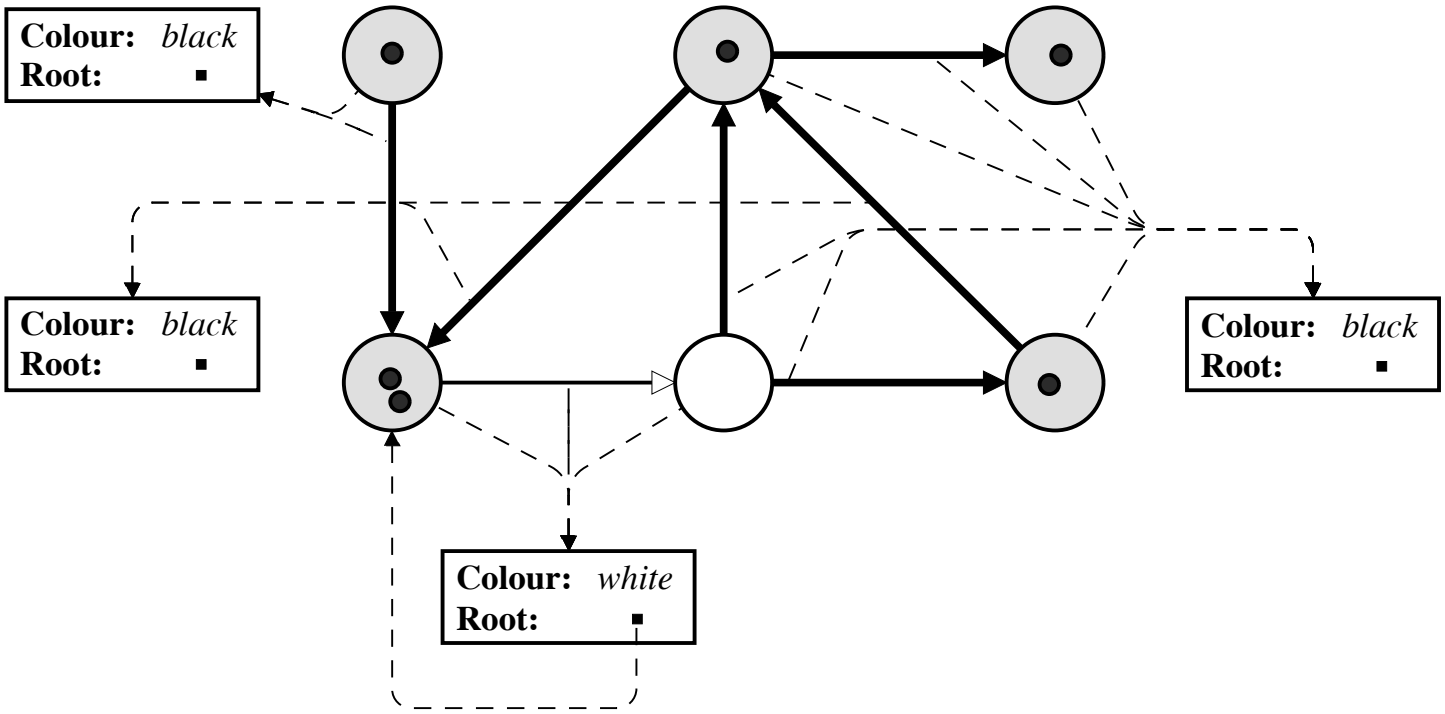

a
b
c
d
e
f
g
h
i
j


{d,b,c}
{e,f}
{g,h}
{i,j}


a
{d,b,c}
b
{e,f}
c
{h,g}
d
{i,j}
e
f
g
h
i
j




```
procedure BREAK(v)
    if v.cr.colour = black then
        r ← v
    else
        r ← v.cr.root
        v.cr.colour ← black
        v.cr.root ← ⊥
    endif
    if r.Q ≠ ∅ then
        u ← ROTATE(r.Q)
        if u.cr.colour = black then
            u.cr ← new-component-record(white, u)
        endif
        r.cr ← (u, r).cr ← u.cr
    endif
    ZERO(r)
end.
```


Colour: black
Root:
Colour: black
Root:
Colour: black
Root:
Colour: white
Root:

Sætning

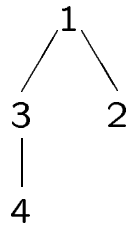
Man kan gøre datastrukturer med begrænset indgrad partiel persistente, sådan at READ og WRITE kan udføres i worst case tid $O(1)$.

Fuld Persistens

[Driscoll, Sarnak, Sleator, Tarjan 89]

- **Alle** hidtidige versioner kan ændres

- Versionerne udgør et **versionstræ**



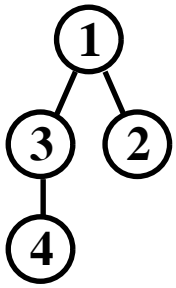
- Et præorder gennemløb af versionstræet giver **versionslisten**

1— 3 — 4 — 2

- En flygtig knude repræsenteres ved en **familie** af knuder

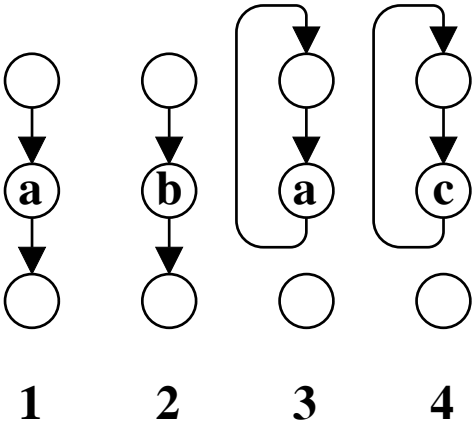
Fuld Persistens (*fortsat*)

Versionstrae

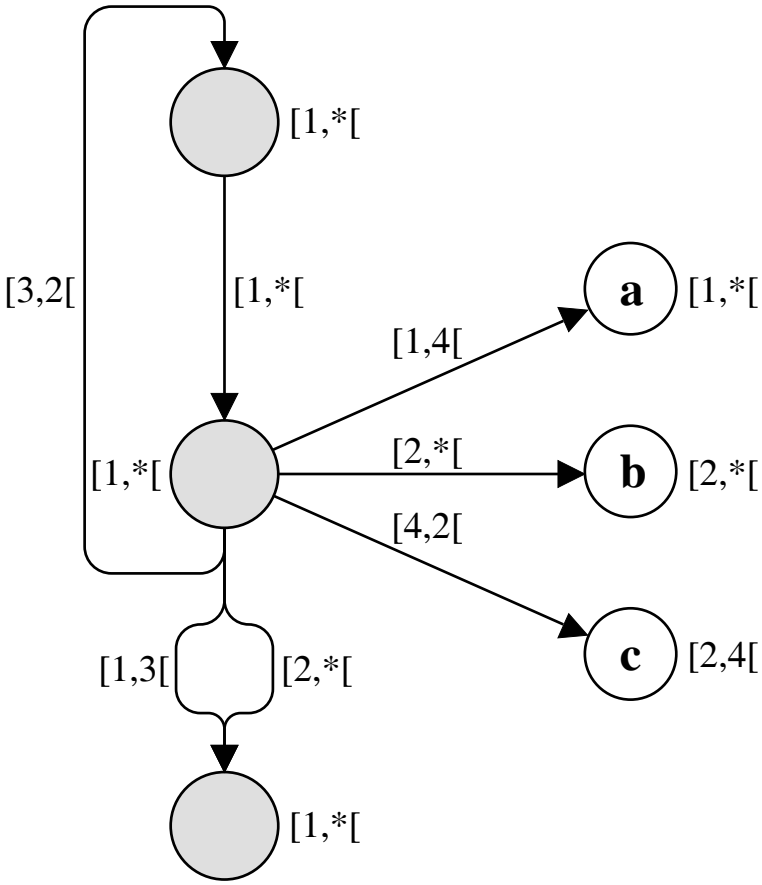


Versionsliste

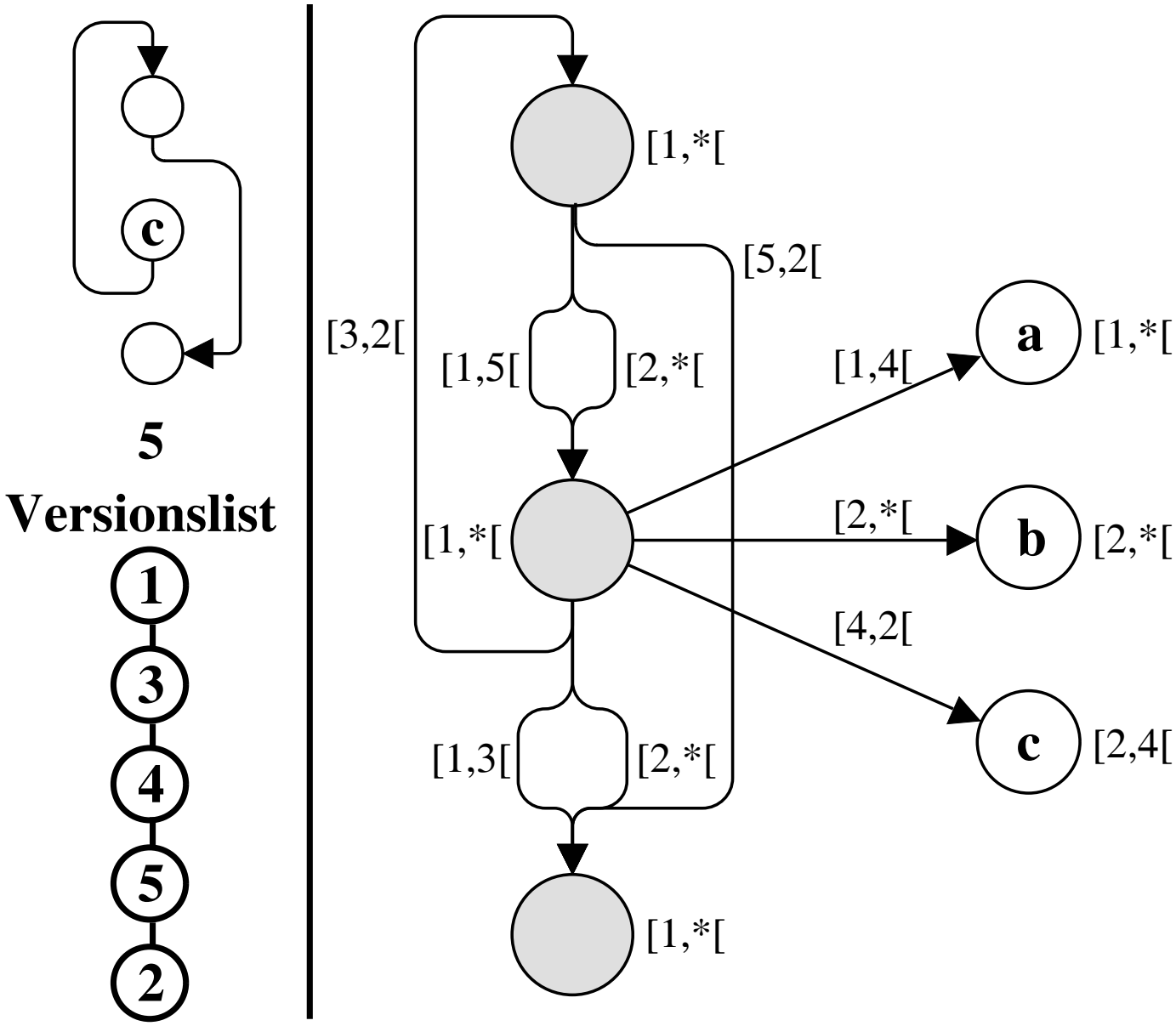
Flygtige Datastrukturer



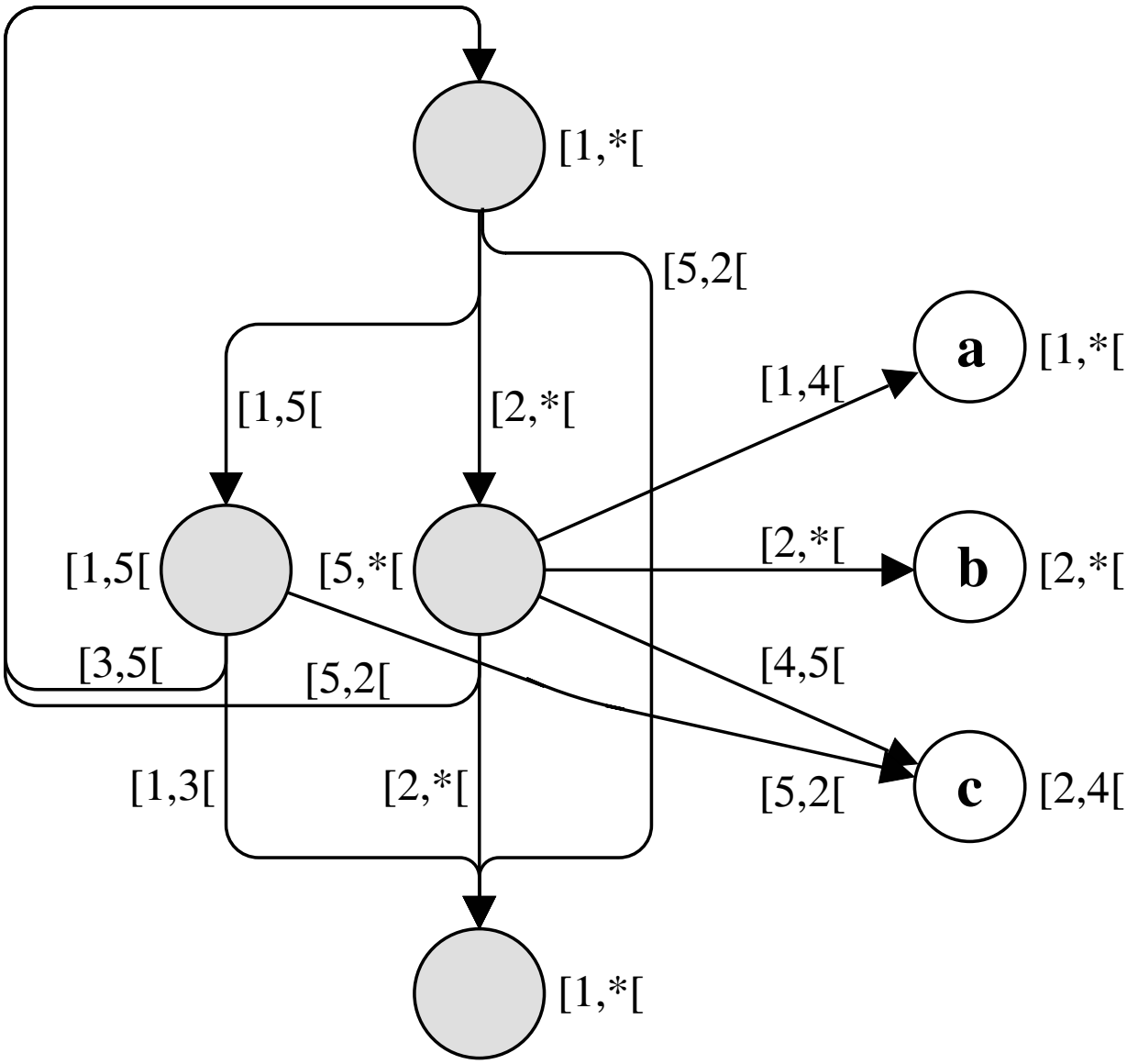
Persistent Datastruktur



Fuld Persistens (*fortsat*)

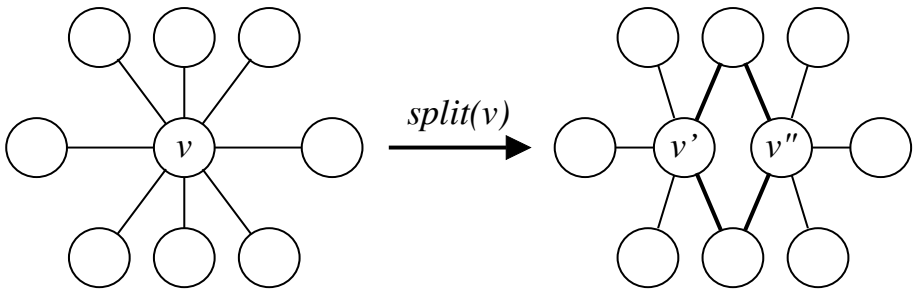


Fuld Persistens (*fortsat*)



Splitninger

- Simplificeret er splitningerne på formen



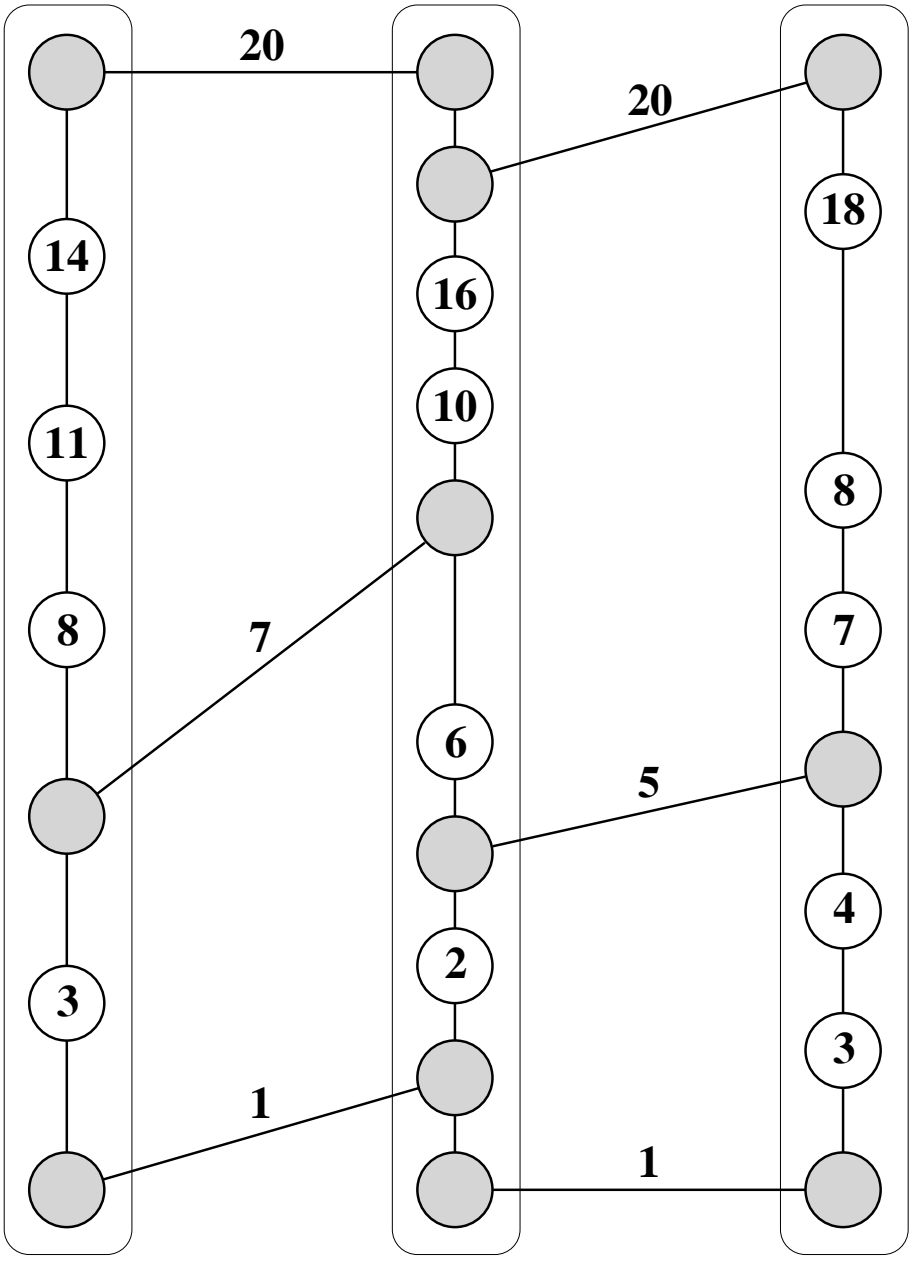
- Antallet af kanter, der splittes i to kanter er begrænset af en konstant, uafhængigt af v's grad

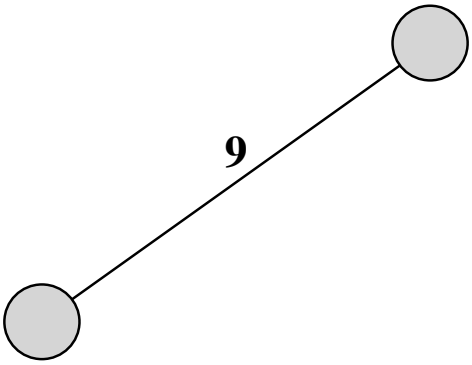
- Problem:

  Hvordan nøjes man med $O(1)$ splitninger per opdatering i en fuldt persistent datastruktur?

Fractional Cascading

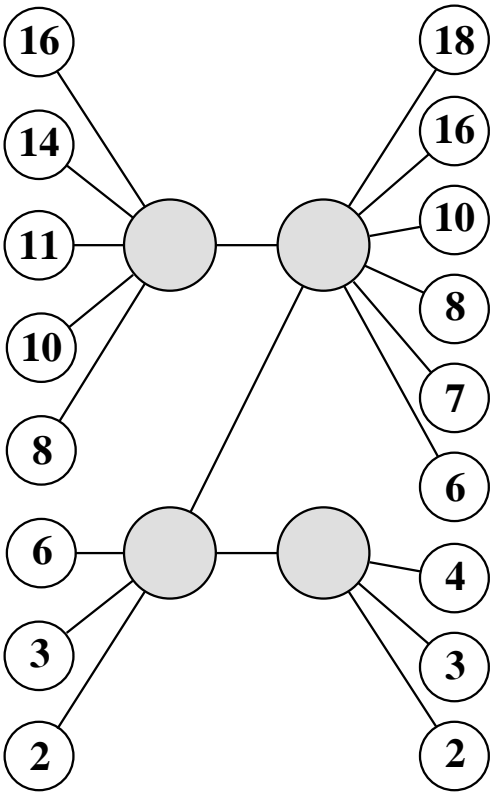
$\{3, 8, 11, 14\}$ $\{2, 6, 10, 16\}$ $\{3, 4, 7, 8, 18\}$

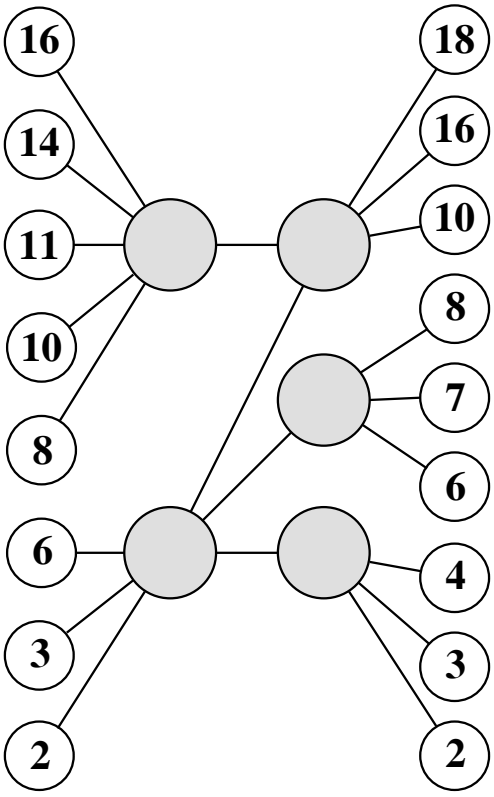



9

Fractional Cascading

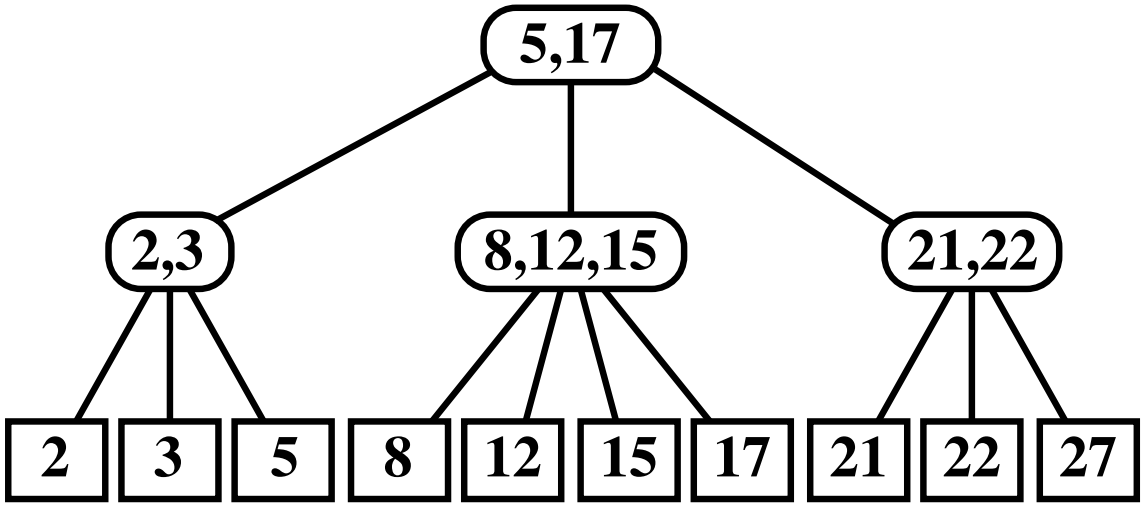
Splitninger i den duale graf

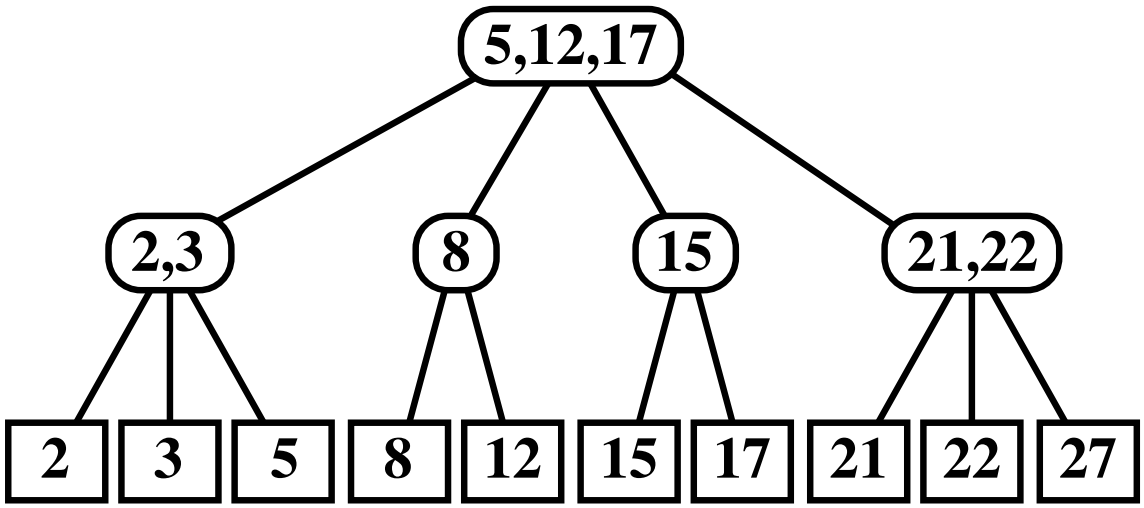




(a, b)–Træer

- Hvordan kan man nøjes med at splitte én knude per indsættelse?





Splitninger

- Det er muligt at nøjes med $O(1)$ splitninger per opdatering i:

  1) (a, b)–træer

  2) Fuldt persistente datastrukturer

  3) Fractional cascading datastrukturen

- Problem:

  Knuden (gappet), der skal splittes, kan ikke findes i tid $O(1)$