

F08 - Träd

5DV149 Datastrukturer och algoritmer
Kapitel 9–10

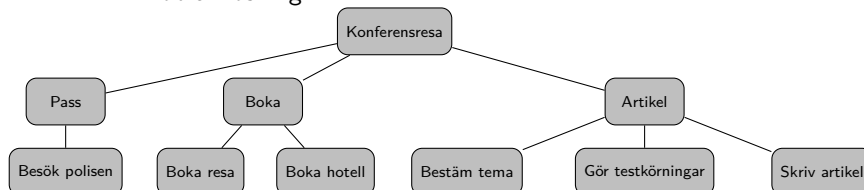
Niclas Börlin
niclas.borlin@cs.umu.se

2024-02-05 Mån

- Modeller för träd
- Tillämpningar av träd
- Organisation och terminologi
- Olika typer av träd
- Trädalgoritmer
- Konstruktioner av träd

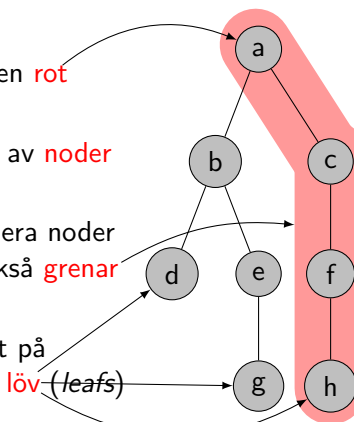
Modeller och tillämpningar

- Modell
 - Ordervägarna i ett regemente eller företag (ordnat träd)
 - Stamtavla/släktträd (binärt träd)
- Tillämpningsexempel inom datavärlden:
 - Filsystem
 - Klasshierarkier i Java/C++
 - Besluts-/sök-/spelträd inom AI
 - Prologs exekvering
 - Problemlösning:



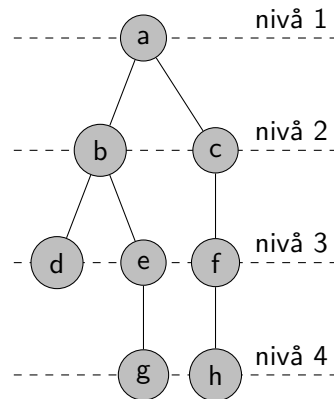
Träd, terminologi

- Varje träd har en **rot** (root)
- Ett träd består av **noder** (nodes)
- Om det finns flera noder så finns det också **grenar** (branches)
- Noder längst ut på grenarna kallas **löv** (leaves)



Träd, organisation

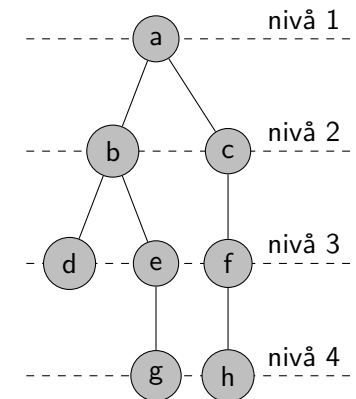
- ▶ Elementen i ett träd kallas för **noder**
- ▶ En nod har en **position** och ev. ett **värde**
- ▶ Värdet på en nod kallas **etikett** (*label*)
- ▶ Ett träds noder finns på olika **nivåer** (*levels*)
- ▶ Ett träd är organiserat som en **föräldra-barn-hierarki**:
 - ▶ Ett **barn** ligger på nivån under dess **förälder**
 - ▶ Alla noder på en nivå med samma förälder kallas **syskon** (*sibling*)
- ▶ Ett **delträd** = en nod och dess avkomma



Djupet för en nod

- ▶ **Djupet** $d(x)$ hos en nod x är antalet bågar från x upp till roten:
 - ▶ "Hur långt är det upp till roten?"

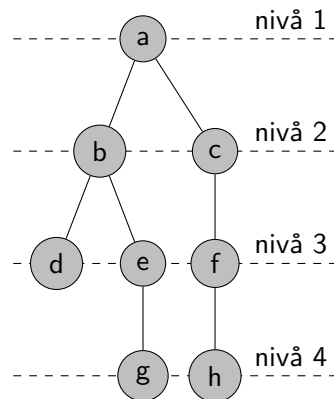
$$\begin{aligned}d(a) &= 0, \\d(b) &= 1, \\d(h) &= 3, \\nivå(x) &= d(x) + 1\end{aligned}$$



Höjden för en nod

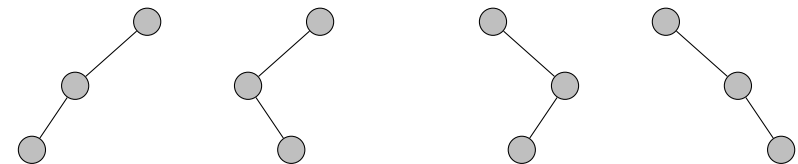
- ▶ **Höjden** $h(x)$ för nod x är antalet bågar på den **längsta grenen** i det träd där x är **rot**
 - ▶ "Hur långt är det ner till lövet?"

$$\begin{aligned}h(T) &= h(\text{roten}) \\h(g) &= 0, \\h(b) &= 2, \\h(a) &= 3 = h(T)\end{aligned}$$



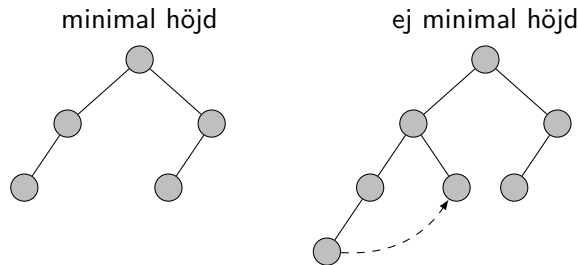
Maximal höjd för ett träd

- ▶ Ett träd med n noder har en **maximal höjd** på $n - 1$
- ▶ Varje nivå har en nod och strukturen liknar en lista
- ▶ Exempel: Några träd med 3 noder och maximal höjd:



Minimal höjd för ett träd

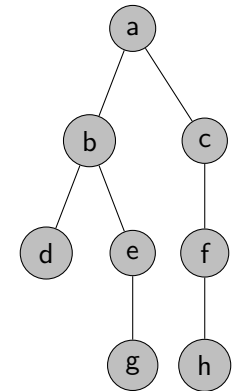
- Begreppet **minimal höjd** för ett träd definieras som att det går ej att **flytta** några noder och få ett träd med **mindre** höjd:



- Minimal höjd för ett träd med n noder beror på det maximala antalet **barn per nod**

Träd, globala egenskaper

- Ett träd har ett **ändligt** antal noder
- Ett träd är en **homogen datatyp**
- Ett träd är en **rekursiv** datatyp; varje delträd är i sig ett träd
- Ett träd saknar **cykler**, dvs. vägen mellan två noder är alltid **unik**



Specifikation av träd

- Navigeringsorienterad
 - Om man arbetar med enstaka träd som **förändras långsamt**, löv för löv, så är **navigeringsorienterad** specifikation bättre.
 - Naturligt med operationer som Insert-node, Delete-node
- Delträdsorienterad
 - Håller man på med träd och delträd som man vill **dela upp** eller **slå samman** är delträdsorienterad bättre.
 - Naturligt med operationer som Join, Split
- Vi kommer att fokusera på den **navigeringsorienterade** specifikationen

Olika typer av träd (1)

- (O)-ordningen för ett träd bestäms av **barnen**:
 - **Ordnat** träd, t.ex. familjeträd:
 - Syskonen är **linjärt ordnade**
 - Syskonen kan representeras av en **lista**
 - **Oordnat** träd, t.ex. filsystemet på en dator:
 - **Ordningen** mellan syskonen är **odefinierad**
 - Syskonen kan representeras av en **mängd**
- **Urträd**:
 - Mer **abstrakt** än de två förra
 - Har en **egen datatyp** som hanterar **syskonen**, t.ex. Tabell
 - Kommer att diskuteras när vi pratar om **Trie**

Olika typer av träd (2)

- ▶ Ett träd kan vara oriktat eller riktat:
 - ▶ **Oriktade** träd
 - ▶ Lika lätt navigera **upp** och **ner** i trädet
 - ▶ **Riktade** träd
 - ▶ Kan bara gå i **en riktning** i trädet
 - ▶ I ett **nedåtriktat** träd saknas Parent
 - ▶ Fungerar för algoritmer som startar i roten
 - ▶ I ett **uppåtriktat** träd saknas Children
 - ▶ Fungerar för algoritmer som startar i löven
- ▶ **Binära** träd, t.ex. stamtafla
 - ▶ Varje nod har **högst två** barn

Gränsyta för Ordnat träd

```
abstract datatype Otree(val)
auxiliary pos
Create() → Otree(val)

Root(t: Otree(val)) → pos
First-child(p: pos, t: Otree(val)) → (Bool, pos)
Next-sibling(p: pos, t: Otree(val)) → (Bool, pos)
Parent(p: pos, t: Otree(val)) → (Bool, pos)

Insert-first-child(p: pos, t: Otree(val)) → (pos, Otree(val))
Insert-next-sibling(p: pos, t: Otree(val)) → (pos, Otree(val))
Delete-node(p: pos, t: Otree(val)) → Otree(val)

Has-label(p: pos, t: Otree(val)) → Bool
Set-label(v: val, p: pos, t: Otree(val)) → Otree(val)
Inspect-label(p: pos, t: Otree(val)) → val

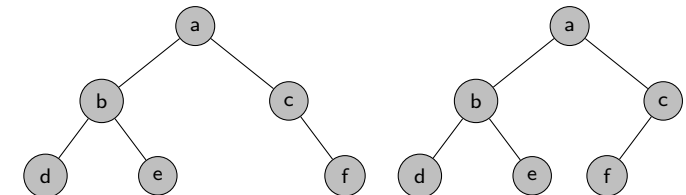
Kill(t: Otree(val)) → ()
```

"Begreppspaus" — Om ordning

- Ordnad** Används för att beskriva olika sätt att ordna **element** före/efter varandra i ett objekt i en datatyp
- Sorterad** När elementvärdena är ordnade enligt någon **ordningsrelation** definierad för elementens **värden** (ex. "äldre än")
- ▶ En sorterad datatyp är alltid ordnad, men ej motsatsen
- Riktad** När det finns en **asymmetri** när det gäller operationer för att **hitta från ett element till ett annat**

Binära träd

- ▶ En nod i ett binärt träd kan ha **högst två** barn
 - ▶ Barnen kallas **vänster-** och **höger-**barn
 - ▶ Ordningen mellan barnen är **odefinierad**, även om träden oftast presenteras med vänsterbarnet "före" (till vänster) om högerbarnet
 - ▶ Två **olika** binära träd kan vara **samma** "ordnade träd med max två barn"



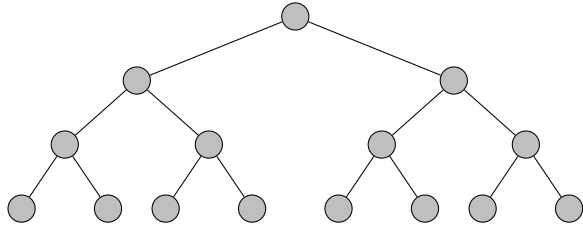
- ▶ Ovanstående träd är **identiska ordnade träd**
 - ▶ I bägge träden har nod **c** ett barn
- ▶ Ovanstående träd är **olika binära träd**
 - ▶ I vänstra trädet har nod **c** ett höger-barn
 - ▶ I högra trädet har nod **c** ett vänster-barn

Höjd för binära träd (1)

- För binära träd T med höjden h gäller att:

- Det minsta antalet noder $n_{\min} = h + 1$
- Det största antalet noder $n_{\max} = 2^{h+1} - 1$

h	n_{\min}	n_{\max}	
0	1	1	= 1
1	2	3	= 1 + 2
2	3	7	= 1 + 2 + 4
3	4	15	= 1 + 2 + 4 + 8
4	5	31	= 1 + 2 + 4 + 8 + 16

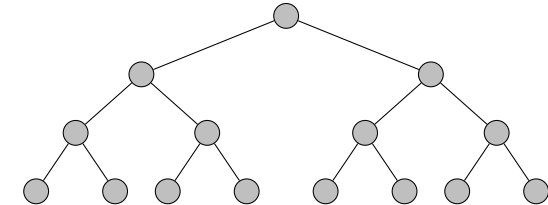


Höjd för binära träd (2)

- Om vi vänder på det:

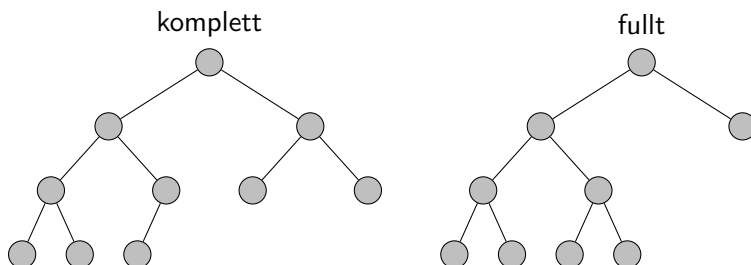
- För ett träd med n noder:
 - Den största höjden $h_{\max} = n - 1$
 - Den minsta höjden är $h_{\min} = \lfloor \log_2 n \rfloor + 1$

n	$\lfloor \log_2 n \rfloor$	h_{\min}
1	0	1
2-3	1	2
4-7	2	3
8-15	3	4
16-31	4	5



Balanserade binära träd

- Man vill ofta ha så **grunda** träd som möjligt:
 - Om vänster och höger delträd har ungefär lika många noder har trädet **balans**
 - I ett balanserat träd är vägen till en **godtyckligt** vald nod $O(\log_2 n)$
- **Komplett** binärt träd (rätt **bra** balans)
 - Fyller på trädet en **nivå** i taget, från **vänster till höger**
- **Fullt** binärt träd (ofta **dålig** balans)
 - Varje nod har antingen **noll** eller **två** barn



Trädalgoritmer

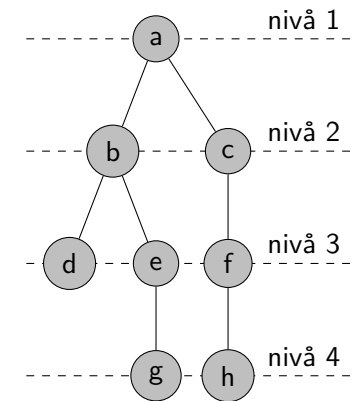
- Basalgoritmer
 - **Konstruera** ett träd
 - Beräkna **djup**
 - Beräkna **höjd**
 - **Slå ihop** två träd
 - **Dela upp** ett träd
 - **Traversera** (förflytta sig i) trädet
 - **Beräkna/evaluera** etikett(-er) i trädet

Traversering av träd

- ▶ Tillämpningar av träd involverar ofta att man
 - ▶ **söker** efter ett element med vissa egenskaper,
 - ▶ **filtrerar** ut element med vissa egenskaper, eller
 - ▶ **transformerar** strukturen till en annan struktur
 - ▶ Exempelvis sortering och balansering
- ▶ Alla dessa bygger på att man **traverserar** strukturen
- ▶ Det finns två grundläggande traverseringsmetoder:
 - ▶ **bredden-först** (*breadth-first*)
 - ▶ **djupet-först** (*depth-first*)

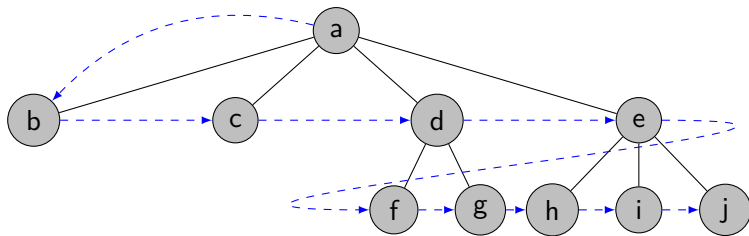
Traversering av träd, bredden-först

- ▶ Trädet undersöks en **nivå** i taget
- ▶ Först roten, sedan rotens barn, dess barnbarn, etc.
- ▶ En **kö** är ofta hjälp i implementationen
- ▶ Varje nod i trädet besöks endast en gång, dvs. $O(n)$
- ▶ Algoritmen normalt **ej** rekursiv



Traversering av träd, bredden-först, exempel

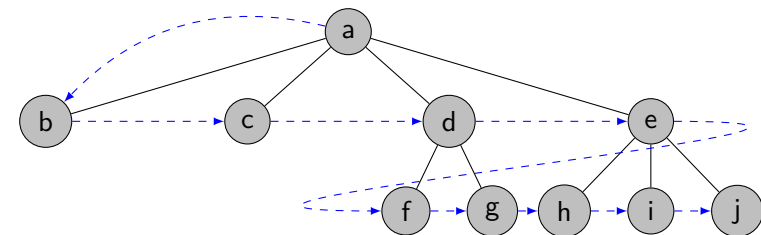
```
Algorithm Traverse-bf-order(T: Tree)
// Input: A tree T to be traversed
for each level L of T do
  for each node n of L do
    Process(n)
```



Ordning: a, b, c, d, e, f, g, h, i, j.

Traversering av träd, bredden-först, detaljerat exempel

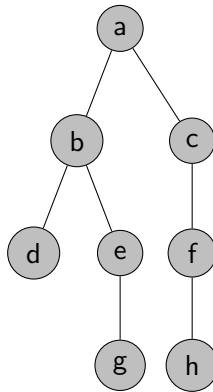
```
Algorithm Traverse-bf-order(T: Tree)
// A tree T to be traversed
q ← Enqueue(Queue-empty(), Root(T))
while not Isempty(q) do
  n ← Front(q)
  Process(n)
  (b, ch) ← First-child(n, T)
  while b do
    q ← Enqueue(q, ch)
    (b, ch) ← Next-sibling(ch, T)
  q ← Dequeue(q)
```



Ordning: a, b, c, d, e, f, g, h, i, j.

Traversering av träd, djupet-först

- Man följer varje **gren** i trädet från roten till lövet
 - En **stack** är ofta till hjälp vid implementationen
 - Varje nod i trädet besöks endast en gång, dvs. $O(n)$
- Tre varianter på traversering:
 - Preorder** **label**, child 1, child 2, ..., child n_i
 - Inorder** child 1, **label**, child 2, ..., child n_i
 - Postorder** child 1, child 2, ..., child n_i , **label**
- Alla varianterna är **rekursiva**

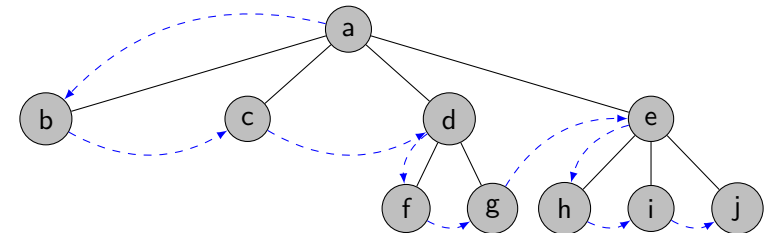


Traversering av träd, djupet-först, preorder

```

Algorithm Traverse-df-pre-order(T: Tree)
// Input: A tree T to be traversed

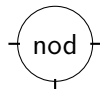
// Do something with the root node first
Process(Root(T))
// Now deal with the children
for each child c of Root(T) do
    Traverse-df-pre-order(c)
    
```



Ordning: a, b, c, d, f, g, e, h, i, j

Traverserings-trick (för utskrivna träd)

- Preorder:**
 - Sätt ett litet streck **till vänster** ("klockan 9") på alla noder
 - Gå "runt trädet", dvs runt alla länkar och noder på "utsidan",
 - När du passerar strecket på en nod, då är det dags att behandla nodens etikett
- Inorder:**
 - Likadant, men sätt strecket **under** noden (mellan första och andra barn-länken)
- Postorder:**
 - Likadant, men sätt strecket **till höger** ("klockan 3") på noderna

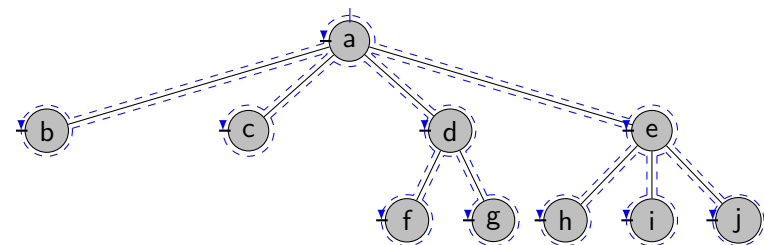


Traversering av träd, djupet-först, preorder

```

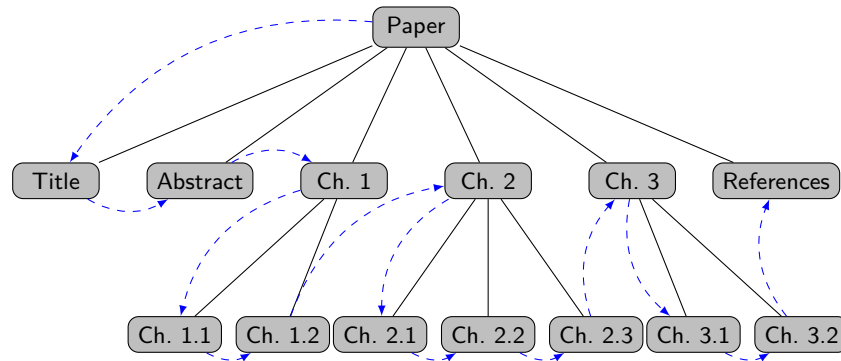
Algorithm Traverse-df-pre-order(T: Tree)
// Input: A tree T to be traversed

// Do something with the root node first
Process(Root(T))
// Now deal with the children
for each child c of Root(T) do
    Traverse-df-pre-order(c)
    
```



Ordning: a, b, c, d, f, g, e, h, i, j.

Preorder — läsa ett dokument

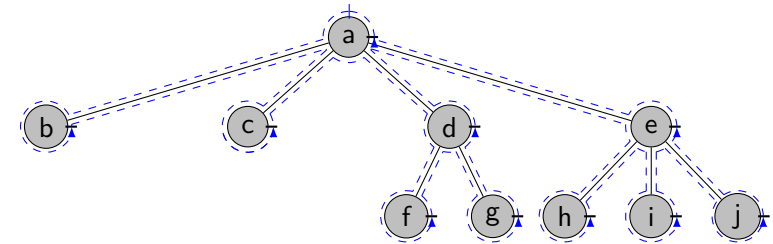


Traversering av träd, djupet-först, postorder

```

Algorithm Traverse-df-post-order(T: Tree)
// Input: A tree T to be traversed

// First deal with all children...
for each child c of Root(T) do
    Traverse-df-post-order(c)
// ...and finally the root
Process(Root(T))
    
```

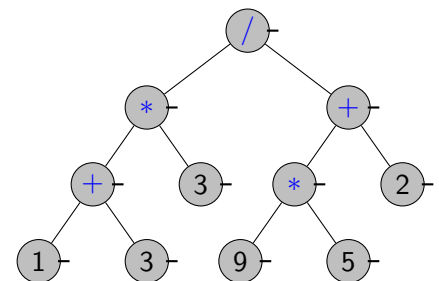


Ordning: b, c, f, g, d, h, i, j, e, a.

Postorder — Beräkna aritmetiska uttryck utan paranteser

```

Algorithm Evaluate-expression(T: BinTree)
if Isleaf(T) then
    return Get-value(Root(T))
else
    x ← Evaluate-expression(Left-child(T))
    y ← Evaluate-expression(Right-child(T))
    op ← Get-value(T)
    return x op y
    
```



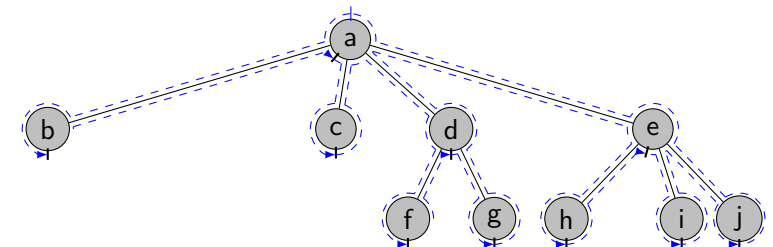
Order: 1, 3, +, 3, *, 9, 5, *, 2, +, /

Traversering av träd, djupet-först, inorder

```

Algorithm Traverse-df-in-order(T: Tree)
// Input: A tree T to be traversed

// Deal with first child first...
Traverse-df-in-order(First-child(T))
// ..then me...
Process(Root(T))
// ..then each of the rest...
for each child c (- first) of Root(T) do
    Traverse-df-in-order(c)
    
```

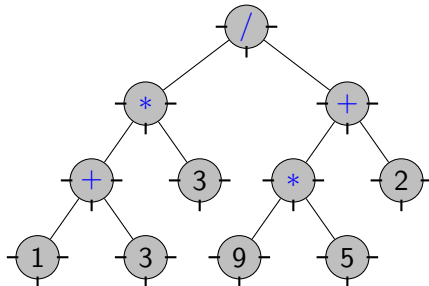


Ordning: b, a, c, f, d, g, h, e, i, j.

Inorder — Skriva ut aritmetiska uttryck

```

Algorithm Print-expression(T: Tree)
Print("(")
if Has-left-child(T) then
    Print-expression(Left-child(T))
Print(Get-value(T))
if Has-right-child(T) then
    Print-expression(Right-child(T))
Print(")")
    
```



Utskrift: (((1)+(3))*(3))/(((9)*(5))+(2)))

Traverseringsordningar binära träd

Pre-order

```

Algorithm Traverse-df-pre-order(T: BinTree)
Process(Root(T))
if Has-left-child(T) then
    Traverse-df-pre-order(Left-child(T))
if Has-right-child(T) then
    Traverse-df-pre-order(Right-child(T))
    
```

Post-order

```

Algorithm Traverse-df-post-order(T: BinTree)
if Has-left-child(T) then
    Traverse-df-post-order(Left-child(T))
if Has-right-child(T) then
    Traverse-df-post-order(Right-child(T))
Process(Root(T))
    
```

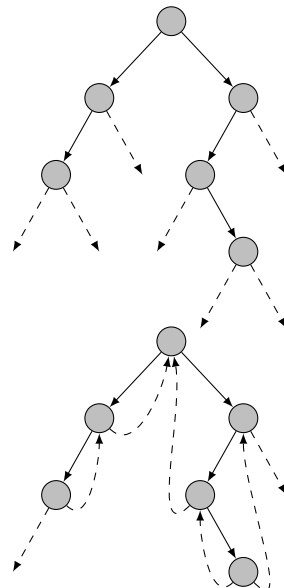
In-order

```

Algorithm Traverse-df-in-order(T: BinTree)
if Has-left-child(T) then
    Traverse-df-in-order(Left-child(T))
Process(Root(T))
if Has-right-child(T) then
    Traverse-df-in-order(Right-child(T))
    
```

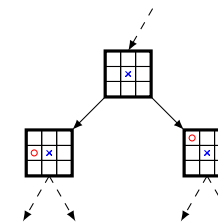
Trädda binära träd

- ▶ Nedåtriktade binära träd har "lediga" länkar
- ▶ Utnyttja dessa för att "trä" **genvägar** i trädet
- ▶ Det är vanligt att skapa **inorder-trädda** träd
- ▶ Detta gör att man kan traversera med hjälp av **iteration** istället för **rekursion**
 - ▶ Enklare algoritmer
 - ▶ Sparar minne



Träd, tillämpningar

- ▶ Konstruktioner av andra typer (speciellt binära träd)
- ▶ Sökträd/beslutsträd/spelträd:
 - ▶ Varje nod symboliserar ett givet **tillstånd**
 - ▶ Barnen symboliserar de olika tillstånd man kan hamna i utifrån förälderns tillstånd

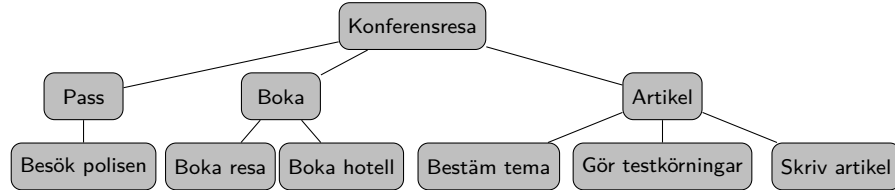


- ▶ Det gäller att hitta **målnoden**, dvs ett tillstånd som **löser problemet**
- ▶ Inte rimligt att bygga upp alla noder (=alla möjliga tillstånd)
- ▶ Ofta används **heuristik**

Tillämpningar

Planträd och OCH/ELLER-träd

- Noderna symboliserar hur man bryter ned ett stort problem i mindre delar och ev. i vilken ordning man bör lösa delproblem



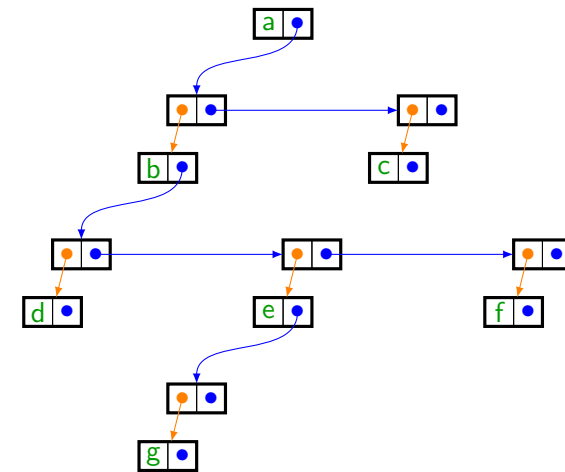
- Ofta använder man OCH/ELLER-träd där man kan ha OCH-kanter eller ELLER-kanter mellan förälder och barn:

OCH alla barn behövs för lösningen
ELLER något barn behövs för lösningen

Konstruktioner av träd (1)

- Nedåtriktat ordnat träd som 1-länkad struktur med lista av barn:

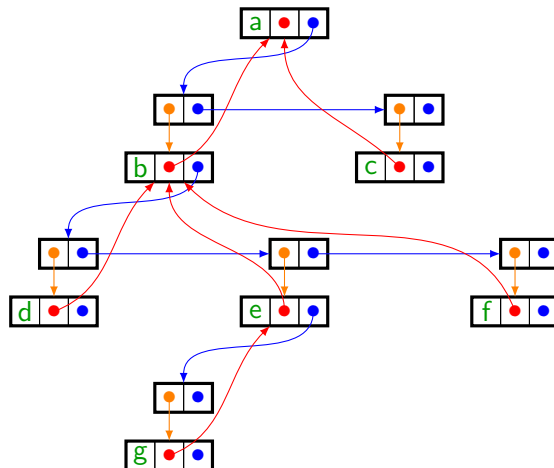
- Noden får en etikett och länk till lista av barn
 - Antalet noder i trädet dynamiskt
 - Antalet barn dynamiskt



Konstruktioner av träd (2)

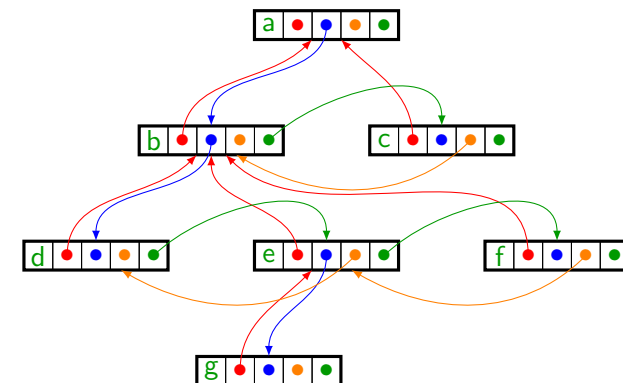
- Utöka till 2-celler så blir trädet oriktat:

- Noden får en etikett, länk till föräldern, samt länk till lista av barn



Konstruktioner av träd (3)

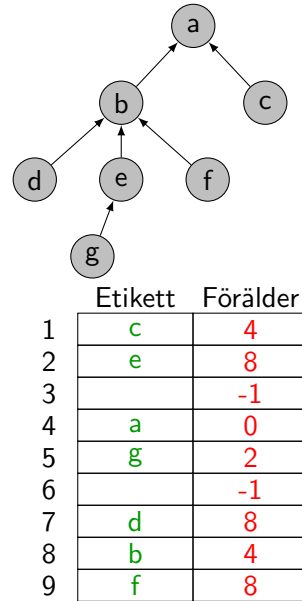
- Oriktat träd med hjälp av 4-cell (etikett, förälder, första barn, föregående syskon, efterföljande syskon)



Konstruktioner av träd (4)

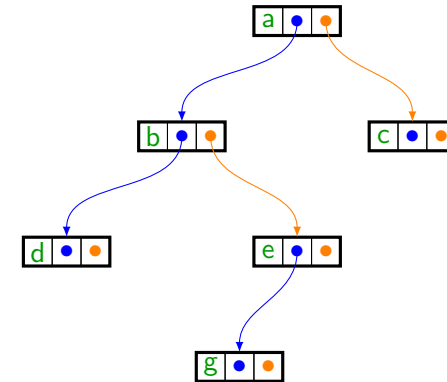
► Oordnat uppåtriktat träd som fält:

- Varje element i en vektor består av ett par: nodens **etikett** och en referens till **föräldern**
 - Tar liten plats
 - Inget bra stöd för traversering (t.ex. svårt avgöra vilka noder som är löv)
 - Maximala storleken på trädet måste bestämmas i förväg



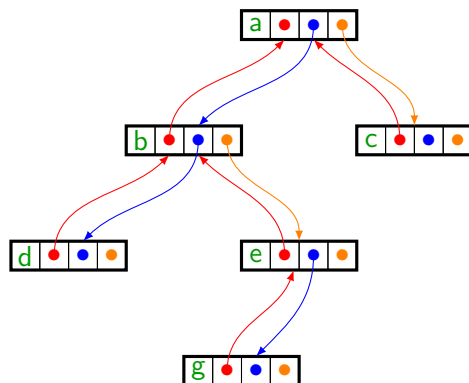
Konstruktioner av binära träd (1)

- Nedåtriktat binärt träd med hjälp av 2-cell (**etikett**, **vänsterbarn**, **högerbarn**)



Konstruktioner av binära träd (2)

- Oriktat binärt träd med hjälp av 3-cell (**etikett**, **förälder**, **vänsterbarn**, **högerbarn**)



Konstruktioner av binära träd (3)

► Binärt träd som fält

- Roten har index 1
- Noden med index i har
 - sitt vänsterbarn i noden med index $2i$,
 - sitt högerbarn i noden med index $2i + 1$,
 - sin förälder i noden med index $\lfloor \frac{i}{2} \rfloor$

- Tar inget utrymme för strukturinformation
- Trädet har ett maxdjup (statiskt fält)
- Krävs "markörer" för null och tom nod
- Ev. slöseri med utrymme

