

Träd:
grundläggande begrepp,
ADT, datastrukturer, BST
TDDE22, 725G97: DALG

Magnus Nielsen

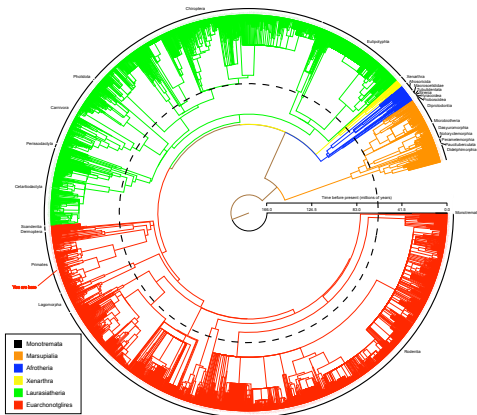
- 1 Grundläggande begrepp
- 2 ADT träd
- 3 Representation av binära träd
- 4 Trädtraversering
- 5 Binära sökträd

Varför träd?

Trädstrukturer uppkommer naturligt i många situationer

- **Filsystem**
- Hierarkiska **klassifikationssystem**
- **Beslutsträd**
- **Hierarkisk organisation** av
 - Organisationer: avdelning, område, grupp
 - Dokument: bok, kapitel, sektion
 - XML-dokument
- För att representera **ordning** eller **prioritet**

Ett superträd över däggdjur



Terminologi

- Ett (*rotat*) *träd* $T = (V, E)$ består av en mängd V av *noder* och *kanter* E , där en kant är ett par $(u, v) \in V \times V$.
- Noder (ibland kallade *hörn*) $v \in V$ lagrar data i ett *förälder-barnförhållande*.
- Ett förälder-barnförhållande mellan noderna u och v visas som en *riktad kant* $(u, v) \in E$, där riktningen är från u till v .
- Varje nod har som mest en föräldranod; kan ha många *syskon*.
- Det finns högst en nod utan förälder – *rotnoden*.

Mer terminologi

- En nods *grad* är antalet barn noden har.
- En nod med 0 barn är ett *löv* eller en *yttre/extern* nod. Övriga noder är *inre/interna*.
- En *stig* är en sekvens av noder (v_1, v_2, \dots, v_k) , där $k > 0$ sådan att v_i, v_{i+1} är en kant för $i = 1, \dots, k - 1$.
- Längden av stigen (v_1, v_2, \dots, v_k) är $k - 1$. Notera att längden av stigen (v_1) är 0.
- En nod n är en *förfader* till en nod v omm det finns en stig från n till v i T .
- En nod n är en *ättling* till en nod v omm det finns en stig från v till n i T .

Ännu mer terminologi

- *Djupet* $d(v)$ av en nod v är längden av stigen från rotnoden till v .
- *Höjden* $h(v)$ av en nod v är längden av den längsta stigen från v till någon ättling till v .
- *Höjden* $h(T)$ av ett träd T är höjden av rotnoden.

Några olika trädtyper

- *Ordnat träd*: linjär ordning mellan varje nods barn
- *Binärt träd*: ordnat träd med $\text{grad} \leq 2$ för varje nod. En nod kan ha ett vänsterbarn och ett högerbarn
- *Tomt binärt träd*(null): binärt träd utan noder
- *Fullt binärt träd*: icke-tomt; graden är antingen 0 eller 2 för varje nod. Följd: antalet löv = $1 +$ antalet interna noder
- *Perfekt binärt träd*: fullt binärt träd, alla löv har samma djup. Följd: antalet löv = 2^h för ett perfekt binärt träd av höjd h
- *Komplett binärt träd*: approximation till perfekt träd för $2^h \leq n < 2^{h+1} - 1$. På avstånd $h - 1$ från rotenoden är alla interna noder till vänster om de externa noderna och det finns som mest en nod med ett barn, vilket måste vara ett vänsterbarn.

- 1 Grundläggande begrepp
- 2 **ADT träd**
- 3 Representation av binära träd
- 4 Trädtraversering
- 5 Binära sökträd

Operationer på en nod v i ett träd T

- *parent*(v) returnerar föräldern till v , **error** om v är rotnoden
- *children*(v) returnerar samling av barn till v
- *firstChild*(v) returnerar första barnet till v eller **null** om v är ett löv
- *rightSibling*(v) returnerar högra syskonet till v eller **null** om det inte finns
- *leftSibling*(v) returnerar vänstra syskonet till v eller **null** om det inte finns
- *isLeaf*(v) returnerar **true** omm v är ett löv
- *isInternal*(v) returnerar **true** omm v inte är en lövnod
- *isRoot*(v) returnerar **true** omm v är rotnoden
- *depth*(v) returnerar djupet av v i T
- *height*(v) returnerar höjden av v i T

Operationer på ett helt träd T

- *size()* returnerar antalet noder i T
- *root()* returnerar rotnoden i T
- *height()* returnerar höjden av T

Och dessutom för ett binärt träd

- *left(v)* returnerar vänstra barnet till v eller **error**
- *right(v)* returnerar högra barnet till v eller **error**
- *hasLeft(v)* testar om v har ett vänstra barn
- *hasRight(v)* testar om v har ett högra barn

- 1 Grundläggande begrepp
- 2 ADT träd
- 3 Representation av binära träd**
- 4 Trädtraversering
- 5 Binära sökträd

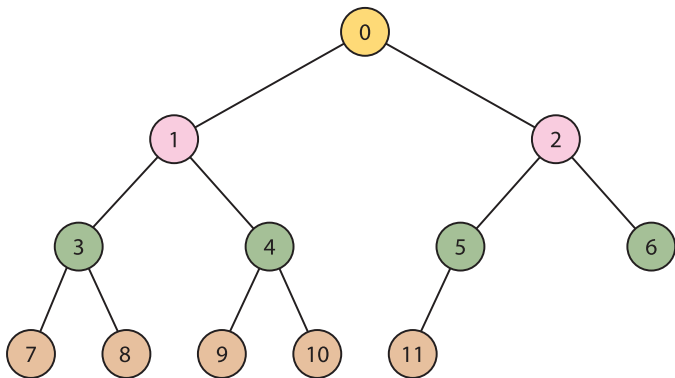
En länkad representation

```
class treeNode<T>  
  nodeInfo: T  
  N: integer  
  children: array[1..M] of treeNode<T>
```

Eller för binära träd

```
class treeNode<T>  
  nodeInfo: T  
  leftChild: treeNode<T>  
  rightChild: treeNode<T>
```

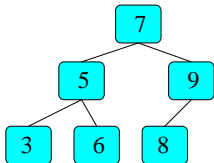
Komplett binärt träd: sekvensiellt minne



Sekvensiellt minne

Använd en $\text{table}\langle \text{key}, \text{info} \rangle [0..n-1]$

- $\text{leftChild}(i) = 2i + 1$ (returnera **null** om $2i + 1 \geq n$)
- $\text{rightChild}(i) = 2i + 2$ (returnera **null** om $2i + 2 \geq n$)
- $\text{isLeaf}(i) = (i < n)$ and $(2i + 1 > n)$
- $\text{leftSibling}(i) = i - 1$ (returnera **null** om $i = 0$ eller $\text{odd}(i)$)
- $\text{rightSibling}(i) = i + 1$ (returnera **null** om $i = n - 1$ eller $\text{even}(i)$)
- $\text{parent}(i) = \lfloor (i - 1) / 2 \rfloor$ (returnera **null** om $i = 0$)
- $\text{isRoot}(i) = (i = 0)$



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

- 1 Grundläggande begrepp
- 2 ADT träd
- 3 Representation av binära träd
- 4 Trädtraversering**
- 5 Binära sökträd

Traversering av träd

Betrakta ett träd T som om det vore en byggnad:

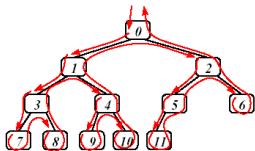
noderna som rum, kanter som dörrar, rotnoden som ingång

Hur utforskar man en okänd (cykelfri) labyrint och tar sig ut igen?

Se till att alltid ha en vägg på höger sida!

Generisk rutin för trädtraversering:

```
procedure VISIT(node  $v$ )  
  for all  $u \in \text{CHILDREN}(v)$  do  
    VISIT( $u$ )
```



Anropa `visit(root(T))` och varje nod i T kommer att besökas exakt en gång!

Traversering av träd

```
procedure PREORDERVISIT(node  $v$ )  
  DOSOMETHING( $v$ )  
  for all  $u \in \text{CHILDREN}(v)$  do  
    PREORDERVISIT( $u$ )
```

▷ före ev. barn

Traversering av träd

```
procedure PREORDERVISIT(node  $v$ )  
  DOSOMETHING( $v$ )  
  for all  $u \in \text{CHILDREN}(v)$  do  
    PREORDERVISIT( $u$ )
```

▷ före ev. barn

```
procedure POSTORDERVISIT(node  $v$ )  
  for all  $u \in \text{CHILDREN}(v)$  do  
    POSTORDERVISIT( $u$ )  
  DOSOMETHING( $v$ )
```

▷ efter alla barn

Traversering av träd (enbart binära träd)

```
procedure INORDERVISIT(node  $v$ )  
  INORDERVISIT(LEFTCHILD( $v$ ))  
  DOSOMETHING( $v$ )  
  INORDERVISIT(RIGHTCHILD( $v$ ))
```

▷ efter alla vänsteråttlingar

Traversering av träd

```
procedure LEVELORDERVISIT(node v)  
   $Q \leftarrow \text{MAKEEMPTYQUEUE}()$   
  ENQUEUE( $v, Q$ )  
  while not ISEMPTY( $Q$ ) do  
     $v \leftarrow \text{DEQUEUE}(Q)$   
    DOSOMETHING( $v$ )  
    for all  $u \in \text{CHILDREN}(v)$  do  
      ENQUEUE( $u, Q$ )
```

Även känd som bredden först.

- 1 Grundläggande begrepp
- 2 ADT träd
- 3 Representation av binära träd
- 4 Trädtraversering
- 5 Binära sökträd

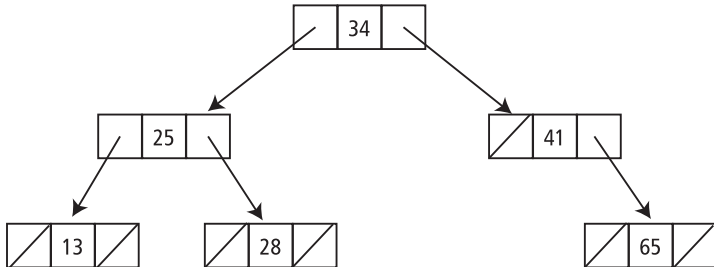
Binära sökträd

Ett *binärt sökträd* (BST) är ett binärt träd sådant att:

- informationen associerad med en nod är linjärt ordnad t.ex. (nyckel,värde).

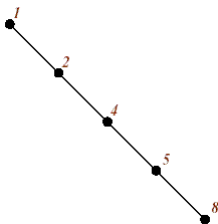
Nyckeln i varje nod är:

- större än (eller lika med) nyckeln hos alla vänsterättlingar, och
- mindre än (eller lika med) nyckeln hos alla högerättlingar.

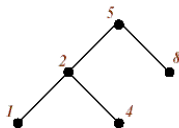


Binära sökträd är inte unika

Samma data kan generera olika binära sökträd



insert: 1,2,4,5,8



insert: 5,2,1,4,8

Lyckad uppslagning

BST i värsta fallet

- BST degenererat till linjär sekvens
- förväntat antal jämförelser är $(n + 1)/2$

Balanserat BST

- djupet av löven skiljer sig inte med mer än 1
- $O(\log_2 n)$ jämförelser

Alltså — Låt oss hålla dem balanserade!

Några vanligt förekommande balanserade träd:

- AVL-träd
- (2,3)-träd, (a,b)-träd,
- ...Röd-Svarta träd, B-träd
- Splay-träd

Nästa gång:
AVL-träd, Multi-way
sökträd, och mer...

www.liu.se