Föreläsning 10

Innehåll

- Träd, speciellt binära träd
  - egenskaper
  - användningsområden
  - implementering

Datastrukturer i kursen

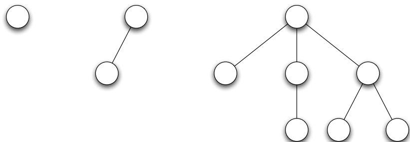
- Hittills har vi använt datastrukturerna **vektor** och **enkellänkad lista**.
  - T.ex. för att implementera de abstrakta datatyperna lista, stack och kö.

I kursen används också

- **träd**
- **binära sökträd och hashtabeller**
  - för att implementera de abstrakta datatyperna mängd (eng. Set) och lexikon (eng. Map).
- **heapar**
  - för att implementera den abstrakta datatypen prioritetskö.

Träd

- Icke-linjär struktur
  - En nod i ett träd kan ha högst en föregångare (förälder), men flera efterföljare (barn).



- Jfr. listor som är linjära strukturer
  - En nod i en lista kan ha högst en föregångare och en efterföljare.

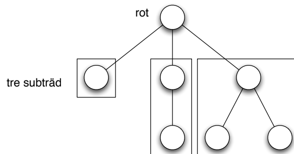
Träd – rekursiv definition

Ett träd är antingen

- tomt (har inga noder)

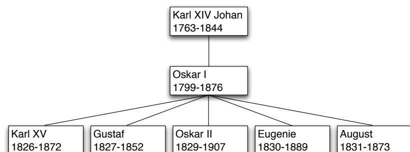
eller

- består av en speciell nod, *roten*, och noll eller flera subträd.



Släktträd

Exempel på användning av träd

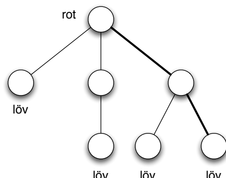


Terminologi lånad från släktträd

- Antag att två noder n_1 och n_2 är förbundna med en båge och n_1 är den övre av dem
  - n_1 är då **förälder** till n_2
  - och n_2 är **barn** till n_1
- Noder som har samma förälder kallas **syskon** (eng: *siblings*)
- Antag att två noder n_1 och n_2 är förbundna med en serie bågar och n_1 är den övre av dem
  - n_1 är då **förfader** (eng: *ancestor*) till n_2
  - n_2 är **avkomling** (eng: *descendant*) till n_1

Terminologi lånad från naturen

- **Rot** – den enda nod som saknar förälder
  - I datavetenskap ritas träden "upp och ned" med roten överst.
- **Löv** – noder som saknar barn
- **Gren** – en serie noder förbundna med varandra



En gren i trädet visas med: ———

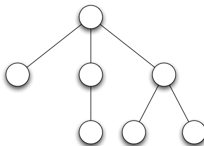
Begreppen nivå eller djup för en nod

- Rotens **nivå** (**djup**) är 1.
- En nod som inte är rot har **nivå** (**djup**) = förälderns nivå (djup) + 1.

nivå = djup = 1

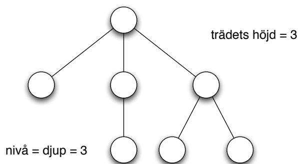
nivå = djup = 2

nivå = djup = 3



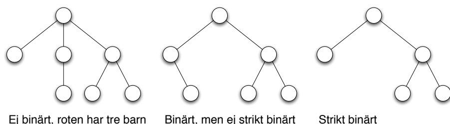
Begreppet höjd för ett träd

- Ett tomt träd har **höjden** 0.
- Ett träd som inte är tomt har **höjd** = maximala nivån (djupet) för noderna i trädet.



Binära träd

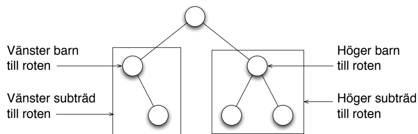
- Ett träd är **binärt** om varje nod har högst två barn (eller ekvivalent, två subträd).
- Ett träd är **strikt binärt** om varje nod har noll eller två barn.
  - D.v.s. alla noder som inte är löv har två barn.



Binära träd – speciell terminologi

För ett binärt träd brukar man tala om

- *vänster* respektive *höger subträd* till en nod
- *vänster* respektive *höger barn* till en nod



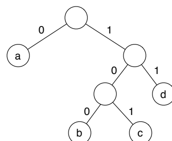
OBS: en nod kan ha höger barn utan att ha vänster barn.

Huffmanträd

Exempel på användning av (strikt) binära träd

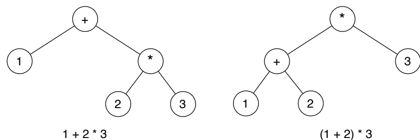
I Huffmankodning byts tecken ut mot koder av olika längd. Vanligt förekommande tecken ska ha kortare kod än mer sällsynta tecken.

Tecken	*Kod*
a	0
b	100
c	101
d	11



Aritmetiska uttryck
Exempel på användning av (strikt) binära träd

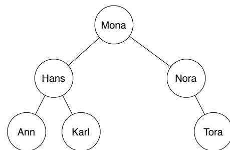
Ett aritmetiskt uttryck med (de binära) operationerna +, −, * och / kan representeras av ett (strikt) binärt träd:



1 + 2 * 3 (1 + 2) * 3

- Operander lagras i löv, operatorer i övriga noder.
- Ett träds värde = operatorn i roten applicerad på värdet av subträden.
- Ett lövs värde är det värde som lagrats i lövet.

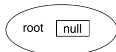
Binära sökträd
Exempel på användning av binära träd



- Lagrar element med **söknyckel** för vilken jämförelseoperationer är definierade. Här: sträng.
- Nycklarna i vänster subträd är mindre än rotens nyckel som i sin tur är mindre än nycklarna i höger subträd.
- Fördel: Enkelt och snabbt att leta upp, sätta in och ta bort element.

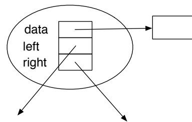
Implementering av binära träd
Klassen BinaryTree

```
public class BinaryTree<E> {
   private Node<E> root;

   public BinaryTree() {
      root = null;
   }

   ... operationer på trädet ...

   private static class Node<E> {
      // se nästa bild
   }
}
```



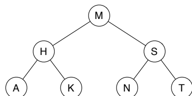
Implementering av binära träd
Klassen Node

```
private static class Node<E> {
   private E data;
   private Node<E> left;
   private Node<E> right;

   private Node(E data) {
      this.data = data;
      left = right = null;
   }

   ...
}
```



Skriv ut trädets noder

```
public void print() {
    print(root);
}

private void print(Node<E> n) {
    if (n != null) {
        System.out.println(n.data);
        print(n.left);
        print(n.right);
    }
}
```

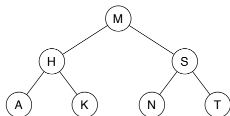


I vilken ordning skrivs noderna ut?

Rekursiva traverseringar av binära träd

- Traversering betyder att vi besöker varje nod i trädet.
- Många operationer på träd kan tolkas som traversering.
  - Under traverseringen utför man operationer på noderna.
- Man kan traversera träd i olika ordningar, t ex följande rekursivt definierade:
  - **preorder**: först roten, sedan vänster subträd i preorder, därefter höger subträd i preorder
  - **inorder**: först vänster subträd i inorder, sedan roten och därefter höger subträd i inorder
  - **postorder**: först vänster subträd i postorder, sedan höger subträd i postorder och därefter roten

Diskutera



I vilken ordning behandlas noderna om man traverserar trädet i

- **preorder**?
- **inorder**?
- **postorder**?

Algoritm för preordertraversering

```
if trädet tomt
   return
else
   besök roten
   traversera vänster subträd
   traversera höger subträd
```

Exempel: Skriv ut innehållet i alla noder i preorder i det träd där n är rot.

```
private void print(Node<E> n) {
   if (n != null) {
      System.out.println(n.data);
      print(n.left);
      print(n.right);
   }
}
```

Algoritm för inorder- och postordertraversering

- Inorder:

```
if trädet tomt
   return
else
   traversera vänster subträd
   besök roten
   traversera höger subträd
```

- Postorder:

```
if trädet tomt
     return
  else
     traversera vänster subträd
     traversera höger subträd
     besök roten
```


Traversera trädet rekursivt – alternativ

Publik metod i trädklassen

Alternativt kan vi placera den rekursiva metoden i nodklassen.

I klassen `BinaryTree` implementerar vi då endast följande publika metod:

```
/** Skriver ut trädet i preorder */
public void print() {
    if (root != null) {
        root.print();
    }
}
```

Metoden `print` i klassen `Node<E>` finns på nästa bild.

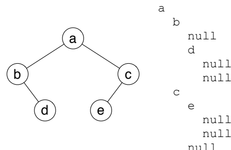
Traversera trädet rekursivt – alternativ

Rekursiv metod i nodklassen

```
/* Skriver ut det träd där noden är rot i preorder. */
private void print() {
    System.out.println(data);
    if (left != null) {
        left.print();
    }
    if (right != null) {
        right.print();
    }
}
```


Exempel: visualisera träd

Visualisera innehåll och form på ett träd genom att översätta trädet till en sträng enligt:



- Strängen beskriver trädet i preorder
- Barn indenteras 2 blanksteg i förhållande till föräldern
- Varje nod på ny rad
- Tomma subträd representeras av delsträngen "null"

Visualisera träd – implementering

Enklare problem – sträng med trädets innehåll

```
public String toString() {
   StringBuilder sb = new StringBuilder();
   buildString(root, sb);
   return sb.toString();
}

private void buildString(Node<E> n, StringBuilder sb) {
   if (n != null) {
      sb.append(n.data);
      sb.append('\n');
      buildString(n.left, sb);
      buildString(n.right, sb);
   }
}
```

- Observera att man skapar ett enda `StringBuilder`-objekt. Det måste skickas med som parameter till den rekursiva metoden.

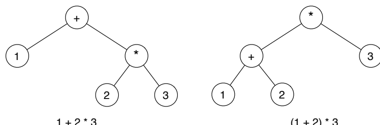
Visualisera träd – implementering

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    buildString(root, 0, sb);
    return sb.toString();
}

private void buildString(Node<E> n, int indent, StringBuilder sb)
    for (int i = 0; i < indent; i++) {
        sb.append(' ');
    }
    if (n == null) {
        sb.append("null\n");
    } else {
        sb.append(n.data);
        sb.append('\n');
        buildString(n.left, indent + 2, sb);
        buildString(n.right, indent + 2, sb);
    }
}
```


Traversering i postorder

Exempel

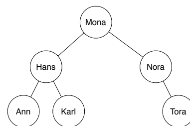
- Om vi har ett binärt träd som representerar aritmetiska uttryck kan vi beräkna trädets värde som en postorder-genomgång.
  - Subträdens värden beräknas först.
  - Därefter "besöks" roten genom att dess operator appliceras på subträdens värde.



Traversering i inorder

Exempel

- Om vi har ett binärt sökträd får vi elementen i stigande ordning vid en inorder-genomgång.



Nivå- för nivåtraversering

Nivå- för nivåtraversering kan implementeras med hjälp av en kö:

```
skapa en tom kö
lägg in roten i kön
så länge kön inte är tom
   tag ut och behandla första noden (actNode)
   om actNode.left != null
      lägg in actNode.left i kön
   om actNode.right != null
      lägg in actNode.right i kön
```


Exempel på vad du ska kunna

- Förklara begreppen träd och binära träd samt begrepp som rot, löv, subträd, förälder, barn.
- Förklara begreppen nivå/djup för en nod och begreppet höjd för träd.
- Implementera träd med en länkad datastruktur.
- Beskriva olika sätt att traversera träd: preorder, inorder, postorder.
- Implementera operationer på träd, speciellt rekursiva metoder.