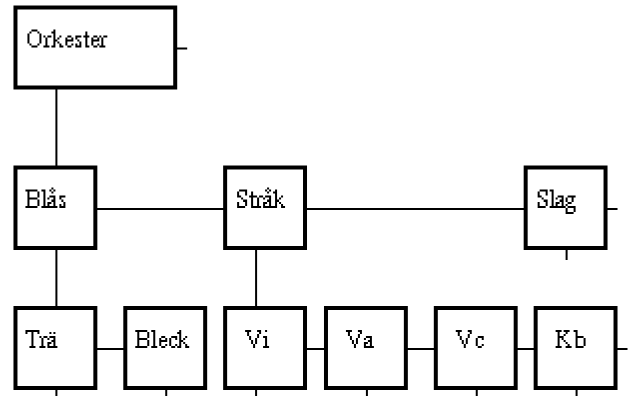
Föreläsning 6: Träd, komprimering

Binärträd och allmänna träd

Stack och *kö* är två viktiga datastrukturer man kan bygga med en länkad lista.

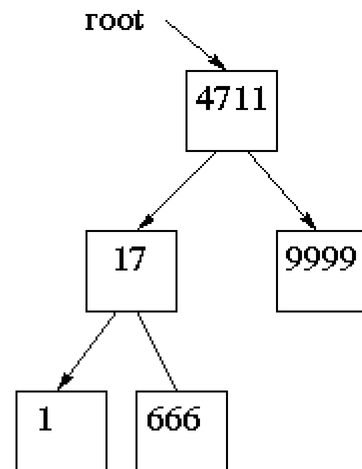
Med **två** referenser i varje objekt kan man emellertid bygga mer intressanta träd, till exempel ett som beskriver en symfoni-orkesters sammansättning.




```
class TreeNode:
  word
  down
  right
```

All systematisk uppdelning kan beskrivas med liknande träd, till exempel ett bokverks uppdelning i delar, kapitel, sektioner osv. Man kan också tänka sej det som ett släktträd med `firstChild` och rightreferensen `nextSibling`. Det är ganska förvånande att det räcker med två referenser i varje objekt, oavsett hur stora barnaskarorna är.

I ett *binärträd* har man högst två barn.




```
class BinaryNode:
  number
  left
  right
```

Högst upp finns konstigt nog trädets *rot* och dit har man alltid en referens `root`. Ett träd är *balanserat* om höjdskillnaden mellan delträden till varje nod är antingen noll eller ett.

Antalet nivåer i trädet avgör hur många objekt det kan innehålla. Ett fullt träd med *k* nivåer innehåller 2 höjt till *k* objekt minus 1. Exempel: k=3 i vår bild ger högst 7 objekt (det finns plats för två till under 9999). Man kan också säga att ett balanserat träd med *N* objekt har cirka log *N* nivåer.

Rekursiva tankar för binärträd

Algoritmer för binärträd blir ofta enklast med en rekursiv tanke.
Fråga: Hur många objekt finns det i trädet?
Rekursivt svar: Antalet objekt i vänsterträdet plus antalet objekt i högerträdet plus 1.
...men ett tomt träd har 0 objekt.

Följande metod gör att `antal(root)` blir 5 för vårt träd.

```
def antal(p) :
  if (p == None) : return 0
  else : return antal(p.left) + antal(p.right) + 1
```

Binära sökträd

I vårt exempelträd ligger små tal till vänster och stora tal till höger. När det är på det sättet har man ett *binärt sökträd*, så kallat eftersom det går så snabbt att söka reda på ett objekt i trädet. Låt oss säga att vi söker efter 666. Vår algoritm blir följande

- Kolla först rottalet.
- Om talet är 666 har vi funnit vad vi söker.
- Om talet är större än 666 letar vi vidare efter 666 i vänsterträdet.
- Om det är mindre än 666 letar vi vidare i högerträdet.
- ...men om vi når en nullreferens `None` så finns inte 666 i sökträdet.

Sökningen kan implementeras rekursivt om man låter anropet `exists(node,word)` returnera `true` ifall ordet finns i det delträd där `node` är rot. Men den kan också göras utan rekursion med hjälp av en stack. Komplexiteten blir densamma.

Det här är ju nästan precis samma sak som binär sökning i en vektor. I båda fallen blir antalet jämförelser cirka log *N*, men binärträdet har två stora fördelar.

1. Insättning av nytt objekt kräver bara log *N* jämförelser mot *N*/2 för insortering i en vektor.
2. Trädet kan bli hur stort som helst men vektorns storlek ges i dess deklaration.

Ett problem med binärträd är att dom kan bli *obalanserade*, och då försvinner den snabba sökningen. Ett riktigt obalanserat sökträd med dom fem talen i exemplet har 1 i roten, 17 till höger, 666 till höger om det osv. Det blir bara en så kallad tarm.

Abstrakta sökträd

Ett binärt sökträd med en ordlista bör ha åtminstone tre metoder

```
class Tree:
  def exists(word):     # ... Returnerar true om ordet finns i trädet
  def insert(word):     # ... Sorterar in ett nytt ord i trädet
  def printTree():      # ... Skriver ut hela trädet
```

Det räcker med dessa metoder för de som vill använda trädet utifrån. Men om man vill implementera metoderna rekursivt måste man kunna göra anrop av typen `exists(node,word)` och därför har man ytterligare tre interna metoder med samma namn men med två parametrar. Den publika metoden som anropas `exists(word)` innehåller bara en sats, anropet till den interna metoden `return exists(root, word)` som sedan gör rekursionen. Variabeln `root` är en intern variabel om pekar ut roten av trädet.

Svårast att implementera är insättningen. En rekursiv tanke är att kolla vilket av delträden ordet ska in i och anropa `insert` med det delträdet.

Binära träd blir lätt obalanserade. Ett sätt att får någorlunda balanserade träd är att efter insättning kontrollera om grenen är obalanserad och i så fall byta plats på noderna och/eller roten. Dessa träd ingår inte i kursen men den intresserade kan läsa vidare om t ex AVL-träd eller röd-svarta träd.

Sökträd med information

Det är sällan man nöjer sej med att veta att en söknyckel finns i databasen; oftast vill man att sökningen ska ge tillbaka information om det sökta. Sökmetoden `getInfo(key)` blir likadan som

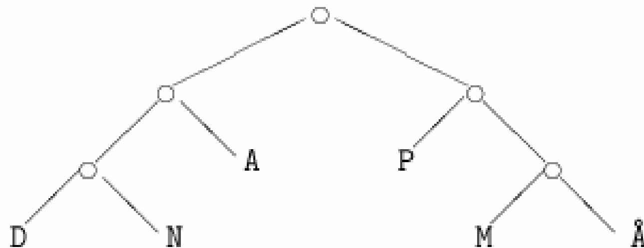
`exists` men returnerar `nod.info` i stället för `true/false`. Trädet kommer att fungera som pythons dictionary. Trädstrukturer är hierarkiska och sådana datastrukturer är mycket vanliga. T.ex.

- Filsystemet använder träd (man kan ha underkataloger i underkataloger).
- En kompilator skapar ett syntaxträd för programmet.
- Databaser använder träd (snabb sökning).
- Schackprogram använder träd för att gå igenom resultaten av möjliga drag.
- Vid datakomprimering kan man använda träd för att få fram en optimal kod

Icke-förstörande komprimering

Huffmankodning

Huffmans metod kodar man varje tecken med ett binärt tal, där vanligare tecken får kortare koder. Huffman är en statistisk metod.



Följdlängdskodning – RLE

I följdlängdskodning, förkortat RLE (Run-Length-Encoding), utnyttjar man att en följd av likadana tecken kan lagras med antal istället för att skrivas ut.

ÅÅÅÅH! JAAAAAAA! AAAAAAAAAAAAH.

Vi ersätter följderna av Å och A med antalet följt av det upprepade tecknet:

4ÅH! J7A! 12AH

Lempel-Ziv

Lempel-Ziv finns i ett otal olika varianter: LZ77, LZSS, LZFG, LZW, LZMW, LZAP, LZY, LZP, osv. Så här fungerar LZW (en variant gjord av T. Welch):

Stoppa in alla bokstäver och tecken i ordlistan `table` och läs in första tecknet från filen i strängen `s`. Nedan visas pseudokod (inte fullt fungerande) för hur det fungerar.

```
for char in file:             # För varje tecken i filen
   if table.has_key(s+c):     # Om teckenföljen finns
      s = s+c
   else:
      print table[s],         # Skriv ut föregående teckenföljd
      table[s+c] = s+c        # Spara denna teckenföljd
      s = c                   # Börja en ny teckenföljd
print table[s]
```

LZ och Huffman används i många komprimeringsprogram, t ex `compress`, `Zip`, `WinZip` och `GZip`

Komprimering av bilder – förstörande komprimering

Bilder tar plats. Det är vanligt att varje bildpunkt (pixel) i en färgbild representeras med ett 24-bitars binärt tal (vilket ger oss åtta bitar för vardera rött, grönt resp blått). Då tar en färgbild 100x100 pixlar 24000 bitar, dvs 24 kB och en bild som täcker en 600x800-skärm tar 11.5 MB. RLE, Huffman och Lempel-Ziv går att använda för att komprimera bilder. Men här kan vi också använda förstörande komprimering för att ta bort information som ögat ändå inte ser.

GIF (Graphics Interchange Format) är ett filformat för bilder där man använder en variant av LZW för att komprimera.

JPEG (Joint Photographic Experts Group) är bättre för foton och andra bilder där närliggande pixlar har liknande färger. Färgbilder delas upp i en belysningsdel och en färgdel, där färgdelen komprimeras med förstörande komprimering eftersom ögat är mindre känsligt för färgförändringar. Sen används en kombination av RLE och Huffmankodning för att koda grupper av pixlar.

Komprimering av rörliga bilder

Det mest kända formatet för rörliga bilder är MPEG (Moving Picture Experts Group). MPEG är egentligen en samling standarder för kombinationer av ljud och video. Komprimeringen av video-delen kan delas upp i *bildkomprimering* av varje enskild bildruta och *tidskomprimering* där man utnyttjar likhet mellan på varandra följande bilder (OBS redigering bli lite knepig. För bildkomprimeringen används i regel JPEG. För tidskomprimeringen finns ett antal olika metoder:

- Koda *likheter* (att en del av bilden ser likadan ut som i förra rutan).
- Koda *förskjutningar* (att en del av bilden har förskjutits sen förra rutan).
- Koda *skillnaden* mellan två bildrutor.
- Koda *förväntad* rörelse.

Komprimering av ljud

Digital lagring av ljud innebär automatiskt en komprimering eftersom vi samplar en analog ljudkurva i ett ändligt antal punkter. Vidare komprimering av digitala ljudfiler kan göras med RLE eller Huffmankodning. Däremot fungerar inte LZ-metoderna särskilt bra, eftersom de bygger på att man hittar upprepningar. Och även om t ex ett musikstycke upprepar sig är det osannolikt att samma upprepningar skulle återfinnas i ljudfilen efter samplingen.

När det gäller ljud kan man också använda förstörande metoder Två exempel på sådana är *tystnadskomprimering* där man ersätter mycket svaga ljud med tystnad och *companding* där man minskar ordlängden för varje ljudpunkt (t ex från 16 till 12 bitar). MP3 (MPEG Audio Layer-3 encoding) använder en kombination av tekniker där man utnyttjar en modell av den mänskliga hörseln samt Huffmankodning.

Felkorrektion

Vill man gardera sig mot fel kan man lägga till redundans (motsatsen till komprimering). Här följer några exempel:

- Kontrollsiffra (t ex sista siffran i ett personnummer).
- Skicka kopior av hela meddelandet, minst tre behövs om man ska kunna korrigera.
- Paritetsbitar, att man lägger till en etta eller nolla till ett binärt tal för att göra det udda. Ett jämnt tal innebär att nån bit är fel.
- Hammingavstånd: Lägg till så många extrabitar till koden så att varje enbitsfel ger en kod som skiljer sig i en bit från den korrumperade koden, men i flera bitar från alla övriga koder.

Entropi

Hur mycket kan man komprimera utan att förlora information? Om det var möjligt att komprimera hur mycket som helst skulle vi kunna få ner varje fil till en bit, men det kan vi uppenbarligen inte. Det finns alltså en undre gräns för hur kompakt man kan få en fil. Om man känner till sannolikheten för varje tecken som ska kodas (som i skräckexemplet ovan) kan man beräkna *entropin* som ger en undre gräns för medellängden hos en kod.