KUNGL
TEKNISKA
HÖGSKOLAN

Pythonkramaren
del två

Datalogi
för teknologer

text: Henrik Eriksson

KTH-CSC
2009

Algoritmer och datastrukturer

Det finns hundratals läroböcker om detta område och Pythonkramaren gör inte anspråk på att vara den bästa. Men det är den.

Innehåll

7 Binärsökning, mergesort och rekursion

– *Kan du slå upp Matti Puntilas telefonnummer?*
– *Javisst, ett ögonblick!* [Många ögonblick går. . .]
– *Nå, kommer det nåt?*
– *Lugn, jag är bara på sidan två!*

Man bör inte använda linjär sökning i telefonkatalogen – det tar för lång tid att hitta den man söker bland tiotusentals andra. Vi ska se att *binär sökning* går betydligt fortare när data är sorterade, till exempel i bokstavsordning.

Och man bör inte använda urvalssortering för att sortera data – det tar för lång tid om man har tiotusentals data. Vi ska se att *merge sort* går betydligt fortare.

Gemensamt för dessa och många andra smarta algoritmer är att dom bygger på en *rekursiv tanke* och att tänka rekursivt är det som brukar ge dataloger dom häftigaste mentala kickarna.

7.1 Binärsökning

När man söker efter Puntila i katalogen ska man inte börja leta vid A utan i mitten och där står Karlsson, Anita. Det var inte rätt namn men nu vet vi att Puntila står i den senare halvan. Mitt i den halvan står Pettersson, Bo och nu har vi bara en fjärdedel av katalogen att leta i. Så här blir pythonkoden.

```
def telefon(person)
    lo = 0                  # index för första namnet
    hi = len(katalog)-1     # index för sista namnet
    while lo<=hi:           # så länge det finns något att leta i
        mid = (lo+hi)//2    # index för mittnamnet
        post = katalog[mid] # mittnamn och nummer
        if person==post[0]: # Om namnen stämmer ...
            return post[1]  # ... returneras numret.
        if person<post[0]:  # Om vår person kommer före mitten ...
            hi = mid-1      # ... räcker det att leta där.
        else:               # Annars letar vi i ...
            lo = mid+1      # ... andra halvan.
    return None             # Personen fanns inte.

person = raw_input("Person: ")
nummer = telefon(person)
```

```
if nummer==None:
    print "Hemligt nummer"
else:
    print "Tel:",nummer
```

Om det finns en miljon personer i katalogen kommer vi nu bara att titta på tjugo av namnen innan vi hittar den vi söker. Varje gång halverar vi ju katalogen, så efter tio halveringar är vi nere i mindre än en tusendel av katalogen ($2^{10} = 1024$) och efter tio halveringar till i en miljontedel av katalogen, det vill säga vi har funnit den vi söker. Om personen inte finns returnerar vi None.

Söker man efter Anita Karlsson behövs inte tjugo namntittar utan bara en, eftersom hon råkar stå i mitten. Men det är så osannolikt att det är bättre att ha det som sista alternativ, så här:

```
- - -
      if person<post[0]:   # Om vår person kommer före mitten ...
          hi = mid-1       # ... räcker det att leta där.
      elif post[0]<person: # Om vår person kommer efter mitten ...
          lo = mid+1       # ... letar vi i andra halvan.
      else:                # Annars måste vi ha ...
          return post[1]   # ... hittat vår person.
- - -
```

Man vill ju också kunna söka namn när man vet nummer. Här kommer en kuggfråga: Räcker det att byta `post[0]` och `post[1]` i koden? Svaret är ja, men bara om databasen först sorteras om efter nummer i stället för efter namn. Men för att sortera en miljon poster duger inte urvalssortering. Den kräver ju att man letar igenom hela databasen en miljon gånger och 10^{12} är ett för stort tal även för en snabb dator. Som tur är finns det snabbare sorteringsmetoder.

7.2 Merge sort

Med *merge* eller *samsortering* menas att låta två sorterade listor smälta samman till en sorterad lista. Man kan till exempel samsortera `[3,7,9]` och `[2,8]` till `[2,3,7,8,9]`.

Idén är nu att man delar upp sin oordnade lista i listor av längd 1 som samsorteras till allt längre ordnade listor och slutligen är hela långa listan ordnad. Anropet `merge(u,v)` med två ordnade listor `u` och `v` ska returnera den samsorterade listan.

```
def merge(u,v):
    m=len(u); n=len(v)
    w=[None]*(m+n)                      # lagom lång tomlista
    i=j=k=0                             # aktuella index i u,v,w
    for k in range(m+n):
        if j==n or i<m and u[i]<v[j]:   # förklaras i texten
            w[k]=u[i]                   # u[i] kopieras till w ...
            i+=1                        # ... och är sen förbrukat
        else:                           # eller också kopieras ...
            w[k]=v[j]                   # ... v[j] till w ...
            j+=1                        # ... och är sen förbrukat
    return w
```

Om listan `v` tar slut, dvs `j==n` ska resten kopieras från `u`, därav det krångliga villkoret.

Vi ska först skapa sorterade listor med längden 2, sedan med längden 4 osv. Om vi kommit till läget att vår lista `d` består av ordnade 4-listor är nästa åtgärd att samsortera `d[0:4]` med `d[4:8]`, därefter `d[8:12]` med `d[12:16]` osv. Det sker väldigt enkelt med anropen

```
d[0:8] = merge(d[0:4],d[4:8])
d[8:16]= merge(d[8:12],d[12:16])
- - -
```

Den snabba sorteringsalgoritm som uppstår kallas *bottom-up merge sort* och arbetar med ordnade högar av storlek `h=1,2,4,8,16,...`. Om det inte går jämnt ut blir den sista högen mindre och det gör inget.

```
h=1                                 # högstorlek 1 först
while h<N:                          # N är len(d)
    a=0                             # a markerar var högen börjar
    while a+h<N:                    # En full hög kan bildas och ...
        b = min(N,a+h+h)            # ... en som kanske inte är det.
        d[a:b] = merge(d[a:a+h],d[a+h:b])
        a = b                       # a flyttas fram
    h = 2*h                         # Gör om med dubbel högstorlek.
```

Det är märkligt att det finns en mycket enklare kod för merge sort som ser helt annorlunda ut, men för att förstå den måste man lära sej att tänka rekursivt. Det är frågan om ett sinnesvidgande mentalt paradigmskifte efter vilket ens gamla jag kan lämnas till personlighetsinsamlingen.

7.3 Rekursiva tankar

Rekursiv kommer från latinet och betyder återlöpande. Definitioner kan vara rekursiva, som när begreppet *avkomling* definieras som *barn eller avkomling till barn*. Att det ord som ska definieras används i definitionen verkar skumt, men i det här fallet fungerar det faktiskt. Vid första genomläsningen inser man bara att avkomlingar bland annat innefattar barnen. Men då återlöper tanken och inser att även barnbarn måste innefattas. Och när tanken löper varv efter varv kommer allt fler generationer att innefattas i begreppet. Det är en *rekursiv tanke*.

- Rekursiv tanke: reducerar problemet till ett enklare problem med samma struktur
- Basfall: det måste finnas ett fall som inte leder till rekursivt anrop

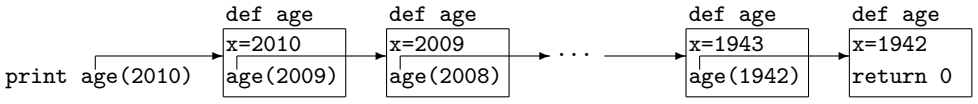
När man börjat tänka rekursivt blir man ganska socialt odräglig:

Oskyldig fråga: Hur många år fyller du i år?
Rekursivt svar: Ett år mer än i fjol.

Om man ger sådana dumsvar blir man inte populär, men faktum är att det går utmärkt att programmera på det sättet. Dock måste man se till att rekursionen avslutas när man kommer ner till något enkelt basfall. I mitt dumsvar skulle jag behöva lägga till "...men år 1942 var jag noll år".
Med koden nedan skulle anropet age(2010) ge värdet 68.

```
def age(year);
  if year==1942: return 0
  else: return 1+age(year-1)
```

Det som händer när funktionen `age` anropar sej själv är inte att man hoppar upp till början av definitionen varv efter varv. Det man anropar är en kopia av `age` och i den har `x` ett nytt värde.



Den sextionionde kopian returnerar värdet noll till den anropande sextioåttonde kopian som plussar på 1 och returnerar det och så vidare tills huvudprogrammet slutligen får värdet 68 och kan printa det.

Den här genomgången av vad som händer i datorn vid rekursion är nyttig att ha sett och förstått en gång, men sedan kan man ha förtroende för att

en korrekt rekursiv tanke alltid fungerar i koden. Och rekursiva tankar finns för många funktioner. Tänk på formeln $\cos 2x = 2\cos^2 x - 1$ som vi alla har stött på någon gång. Den kan användas för att ge ett rekursivt svar på cosinusfrågor.

Oskyldig fråga: Vad är $\cos \pi/3$?
Rekursivt svar: $2\cos^2 \pi/6 - 1$.

Rekursionen återför problemet till att finna cosinus för halva vinkeln. Basfallet är att vinkeln är så liten att approximationen $\cos x \approx 1 - x^2/2$ kan användas. (Det är början på maclaurinutvecklingen för cosinus.)

```
def cos(x):
    if abs(x)<0.0001:
        return 1-x*x/2
    y=cos(x/2)              # ett enda rekursivt anrop
    return 2*y*y-1          # cos(x/2)*cos(x/2) utlöser lavin!
```

Det är svårt att tro att den rekursiva funktionen fungerar, så testa den gärna själv. Om man råkar göra två rekursiva anrop som vart och ett gör två rekursiva anrop osv blir det en lavin med hundratusentals anrop, så det bör man akta sej för.

Många algoritmer, inte bara funktioner, beskrivs enklast rekursivt. Så här beskrivs den sorteringsalgoritm som kallas rekursiv mergesort.

Oskyldig fråga: Hur sorterar jag den här tentabunten i bokstavsordning?
Rekursivt svar: Sortera varje halva för sej och samsortera sedan ihop dom.

Basfallet är att det bara finns en tenta i bunten och koden är så här.

```
def mergesort(bunt):
    N = len(bunt)
    if N<=1: return bunt
    halva = mergesort(bunt[:N//2])
    halvb = mergesort(bunt[N//2:])
    return merge(halva,halvb)
```

Både bottom-up mergesort och rekursiv mergesort har nackdelen att det skapas kopior av listan som ska sorteras. Man skulle kunna tro att själva tentorna kopieras, men så är det inte! Det är bara listobjekten som pekar på tentorna som kopieras och därför är mergesort användbar även för riksskatteverkets tunga register.

7.4 Rekursiva sifferexempel

Oskyldig fråga: Hur många siffror har heltalet n?
Rekursivt svar: En siffra mer än om sista siffran stryks.

Men tal mindre än tio är ensiffriga och det är basfallet.

```
def antalsiffror(n)
  if n<10: return 1
  return 1+antalsiffror(n//10)
```

Oskyldig fråga: Vilken siffersumma har heltalet n?
Rekursivt svar: Sista siffran plus siffersumman om sista siffran stryks.

Men noll har siffersumman noll och det blir basfallet.

```
def siffersumma(n):
  if n==0:
    return 0
  return (n%10) + siffersumma(n//10)
```

Oskyldig fråga: Hur skriver man talet n binärt?
Rekursivt svar: Skriv n/2 binärt, sedan en nolla eller etta beroende på om n var jämnt eller udda.

Men talen 0 och 1 skrivs likadant binärt.

```
def writebinary(n):
  if n==0 or n==1:
    print n
  else:
    writebinary(n//2)
    print n%2
```

7.5 Hemma hos datorn: Anropsstacken

När en funktion anropas läggs en så kallad *aktiveringspost* på den så kallade *stacken*. Underst i stacken finns redan den globala namnlistan. I aktiveringsposten finns *återhoppsadressen* till det ställe i programmet varifrån anropet skedde och funktionens *lokala variabler*. Det är parametrarna i anropet och

övriga variabelnamn som införs i funktionen. Om det sker ett anrop inne i funktionen läggs en ny aktiveringspost överst i stacken osv. Figuren över alla rekursiva anrop av `age` visar sextioåtta aktiveringsposter, var och en med sitt eget `x`. Det är förstås inte nödvändigt att ha hela koden med i varje post – det räcker bra med en återhoppsadress.

Om en funktion är alltför rekursiv blir stacken överfull. Den rymmer knappt tusen poster, så `age(2942)` ger felutskrift. Om man vill vara mer rekursiv än så kan man enkelt öka stackens storlek, men hur det går till avslöjas inte här.

7.6 Varningar och råd

Binärsökning kräver att listan är ordnad efter det man söker på.

Mergesort är mycket snabbare än urvalssortering för stora N.

2^{10} är drygt tusen.

2^{20} är drygt en miljon.

Basfall krävs för att inte rekursion ska bli oändlig.

Lavin kan det bli om en funktion gör två rekursiva anrop.

7.7 Övningar

71. Ge en rekursiv tanke för binärsökning.

72. Ge en rekursiv tanke för antalet siffror i heltalet n.

73. Skriv rekursiv kod för `fakultet(n)`. Kom ihåg att $0! = 1$.

74. Skriv rekursiv kod för `exp(x)`.Några nyttiga egenskaper står här intill.

$$e^{x+y} = e^x \cdot e^y$$
$$e^h \approx 1 + h + h^2/2$$

75. Definiera `binomial(n,k)` med hjälp av rekursionen

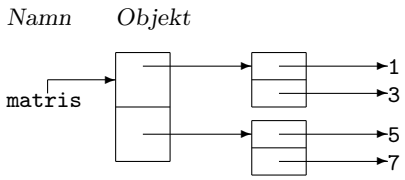
$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}$$

och skriv sedan ut Pascals triangel.

8 Pekare, stackar och köer

Bilder av hur det ser ut i datorns minne brukar visa *pekare*. Så här ser till exempel minnet ut efter satsen

```
matris = [[1,3][5,7]]
```

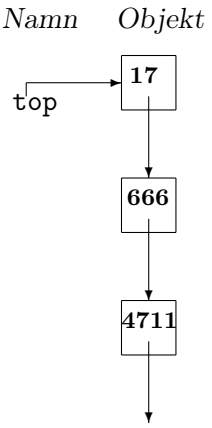


Namnet pekar på ett listobjekt med två fack och varje fack har en pekare till ett annat listobjekt med två fack som slutligen har pekarna till talobjekten.
Det går förstås inga små pilar mellan minnescellerna utan pekarna motsvaras av minnesadresser. Minnets celler är numrerade från noll till någon miljard och jämte namnen i namnlistan står adresserna till motsvarande objekt. Ett listobjekt innehåller bara adresser till andra objekt som i vårt exempel också är listobjekt med adresser till talobjekten.

Datastrukturer med objekt som hänger ihop med pekare kallas *länkade listor*. Vanligast är stackar och köer, så dom ska vi ägna oss åt.

8.1 Stackar

Stacken ser ut att bestå av många listobjekt med två element. I så fall skulle den ha kunnat konstrueras så här.




```
top = [17,None]          # översta objektet
top[1] = [666,None]      # näst översta objektet
top[1][1] = [4711,None] # understa objektet
```

Men så gör man inte! Det är svårt att arbeta med så många index och lätt att av misstag lägga in värden i index `[1]` eller adresser i `[0]`. I stället för index använder man namn, till exempel så här.

```
top = Node()               # ett tomt objekt
top.value = 17             # får ett värde och
top.next = Node()          # en pekare på ett tomt objekt
top.next.value = 666       # som får ett värde och
top.next.next = Node()     # en pekare på ett tomt objekt
top.next.next.value=4711   # som får ett värde
```

Här är `Node` en egen objekttyp eller *klass*, som man också säger. Definitionen kan stå överst i programfilen eller ligga i en egen fil. Vi kallar här namnen

`value` och `next` för *fält* men ibland ser man beteckningar som *attribut*, *medlemmar* eller *klassvariabler*.

```
class Node:           # En hemgjord objekttyp med två fält ...
    value = 0         # ... som får startvärden 0 och None
    next = None
```

Ordet stack betyder trave, till exempel en tallrikstrave på en lunchservering. I en sådan kan man bara komma åt översta tallriken och det är så man använder en stack i datorn också. Eftersom tallrikstraven har en fjäder under traven kallar man alltid dom båda stackoperationerna *push* och *pop*. Så här pushar man värdet `3.14` på stacken.

```
p = Node()       # En tom nod skapas ...
p.value = 3.14   # ... får ett värde ...
p.next = top     # ... pekar på toppnoden ...
top = p          # ... och blir ny toppnod
```

Om denna operation definieras som `push(3.14)` skulle vi kunna bygga hela stacken med fyra anrop.

```
push(4711)
push(666)
push(17)
push(3.14)
```

Vi vill sedan kunna få ut det senast pushade värdet med satsen `x=pop()`. Så här kan definitionerna se ut.

```
def push(x):
  global top         # Annars får push inte ändra globala top
  p = Node()
  p.value = x
  p.next = top
  top = p
def pop():
  global top         # Annars får pop inte ändra globala top
  x = top.value
  top = top.next
  return x
def isempty():       # Returnerar True om stacken är tom
  return top==None
```

Kolla själv att `top` blir `None` när stacken tömts! Det är ofta lämpligt att göra anropet `if isempty():` innan man anropar `pop()`, eftersom det blir felavbrott om man försöker poppa en tom stack.

En hedersregel är att bara använda dessa tre anrop vid hantering av stackar. Går man in och mekar med `top.next.next` osv är det lätt att förstöra sammanhållningen. Och vi tycker synd om dom stackarna som utsätts för det!

En stack kan användas för att vända på ordningsföljder. Exempel:

```
print "Ange resrutt, avsluta med extra returtryckning!"
while True:
    place = raw_input("Plats: ")
    if place == "": break
    push(place)
print "Hemfärden går då följande rutt."
while not isempty():
    place = pop()
    print place
```

8.2 Abstrakt stack

Som vi skrev är det olämpligt att gå in och meka med `top.next` osv. Man bör hålla sej till dom tre anropen och allra bäst är det om man inte ens kan se koden där `top.next` osv förekommer. Alltså lägger man helst den i en fil `stack.py`, som kan importeras av alla som vill använda stacken.

Ofta vill man ha två stackar i samma program och då måste man modifiera anropen så att man även anger den stack man vill pusha eller poppa. Man skulle kunna tänka sej anrop av typen `push(stack1,17)` och `x = pop(stack2)`. Men det blir vackrare om man definierar en hemgjord klass `Stack`. Då blir anropen så här.

```
from stack import Stack
stack1 = Stack()           # Ett stackobjekt ...
stack2 = Stack()           # ... och ett till
stack1.push(4711)          # Betyder push(stack1,4711)
x = stack1.pop()           # Betyder x = pop(stack1)
if stack2.isempty):        # Betyder if isempty(stack2):
    stack2.push(x)         # Betyder push(stack2,x)
```

I filen `stack.py` finns följande kod.

```
class Node:
    value = 0
    next = None
class Stack:
    top = None              # Startvärde är tom stack
    def push(self,x):       # Stackobjektet heter self
        p = Node()
        p.value = x
        p.next = self.top  # Bara top duger inte
        self.top = p
    def pop(self):          # Stackobjektet heter self
        x = self.top.value
        self.top = self.top.next
        return x
    def isempty(self):      # Stackobjektet heter self
        return self.top==None
```

Det enda omvärlden behöver känna till är stackklassens *metoder* `push`, `pop` och `isempty` och hur dom anropas. Ingen användare ska känna till `Node`, `value`, `next` eller `top`.
En tillämpning kan vara att skriva ut en datafil med rader av typen

```
420119-0818 Eriksson, Henrik   pythonkramare
410423-1206 Eriksson, Gerd     matlabbare
 - - -
```

men med damerna först. Då kan man tillfälligtvis pusha herrarna på en stack och på slutet poppa och skriva ut dom.

```
from stack import Stack
manstack = Stack()              # En tom stack skapas.
datafil = open("data.reg")
print "Damer:"
for rad in datafil:             # Gå igenom rad för rad.
    if rad[9] in "13579":
        manstack.push(rad)      # Pusha herrar på stacken.
    else:
        print rad.strip()       # Skriv ut damer.
print "Herrar:"
while not manstack.isempty():
    print manstack.pop()        # Skriv ut herrar.
```

Oskyldig fråga: Hur många element finns det i stacken?
Rekursivt svar: Ett mer än om du poppar stacken.

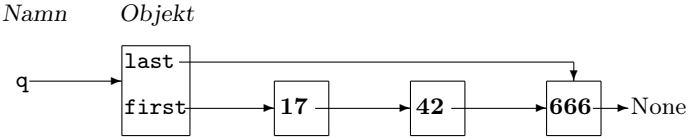
```
def size(stack):
    if stack.isempty():    # Basfallet
        return 0
    x = stack.pop()
    return 1+size(stack)   # Rekursionen
```

Den här rekursiva funktionen ger rätt svar men har en obehaglig bieffekt. Efter anropet `print size(stack)` är stacken tömd! Man måste förstås pusha tillbaka det man har poppat av från stacken, alltså så här.

```
def size(stack):
    if stack.isempty():
        return 0
    x = stack.pop()
    n = 1+size(stack)
    stack.push(x)
    return n
```

8.3 Köer

Kassaköer känner alla till. Regeln är att `queue.put(person)` ställer personen sist i kön och att `person=queue.get()` hämtar personen först i kön. En lämplig struktur med objekt och pekare ser ut så här.



Tänk efter vad som händer med kön i figuren när dom här anropen utförs och vad som skrivs ut. Ledning: Brömsebro!

```
x=q.get()
q.put(x)
x=q.get()
q.put(q.get())
while x>0:
    x=q.get()
    q.put(x-6)
print q.get()+997
```

Det köobjekt som skapas med `q=Queue()` har två pekare, `q.first` och `q.last` som pekar i var sin ände av en länkad lista av nodobjekt. Metoderna `get()` och `isempty()` blir likadan som för en stack, men för att putta in ett nytt värde sist i kön måste man utnyttja `last`-pekaren.

```
class Queue:
    first = None
    last = None
    def put(self,x):
        p = Node()
        p.value = x
        if first==None:        # Vad ska first och last
            - - -              # peka på? Tänk ut det!
        else:
            self.last.next = p
            self.last = p
    def get(self):
        - - -
    def isempty(self):
        - - -
```

Om vi i damernaförstexemplet tänker oss att posterna finns i en kö i stället för på en fil kan vi lösa uppgiften på ett roligare sätt. Personerna hämtas en och en från kön, uppger kön och puttas in sist i kön om könet är manligt. När alla har gåtts igenom innehåller kön alla män som då kan skrivas ut. Men hur ska man veta att första genomgången är klar?

En idé är att lägga in ett markörvärde sist i kön. När markören dyker upp igen är första genomgången klar.

```
\index{markör}
q.put("Markör")
while True:
    x = q.get()
    if x=="Markör": break
    - - -
```

I stället för put och get ser man ofta namnen enqueue och dequeue. Ibland finns också metoden `peek` som returnerar första värdet men låter det stå kvar först i kön. Om nu `x = q.peek()` är samma sak som `x = q.first.value` kan man undra varför `peek` behövs. Svaret är abstraktion! Huvudprogrammet

känner inte ens till namnen `first` och `value`, bara klassen `Queue` och dess metoder.

Det finns ett sätt att lägga till metoden `peek` utan att ha tillgång till `class Queue`. Det är det rekommenderade sättet i objektprogrammering och det kallas *arv*.

```
from queue import Queue
def class Peekqueue(Queue):
    def peek(self):
        return self.first.value

q = Peekqueue()
q.put(17)          # Ärver metoden put
print q.peek()     # Nya metoden peek
```

Den nya klassen ärver allt från den gamla klassen och lägger till nya finesser. Det går också att modifiera metoder från den gamla klassen, men det tar vi upp i ett senare kapitel.

8.4 Ringbuffertar

Om lastnoden kopplas vidare till firstnoden får man en *ringbuffert* eller *cirkulär lista*. Oftast gör man en ring av en viss storlek, och lägger efter hand in värden i dessa noder. När man kommit varvet runt återanvänder man noderna och lägger in nya värden på dom gamla värdenas plats. Jämfört med en vanlig kö sparar man in tiden för att skapa och skräpsamla noder men när ringbufferten blir överfull förstörs dom äldsta värdena. Buffertar finns när data produceras och konsumeras av olika processer, till exempel produceras av ett tangentbord och konsumeras av ett program.

```
class Ringbuffer(Queue):
    first = None               # Nod i tur att läsas
    cursor = None              # Nod i tur att skrivas
    def put(self,x):           # Modifierad put-metod
        self.cursor.value = x
        self.cursor = self.cursor.next
    def isempty(self):         # Modifierad isemptymetod
        return first==cursor
    def setsize(self,n):       # Skapar ring med n noder
        last = Node()          # last är en hjälppekare
        last.next = last       # En ring med en enda nod skapas
```

```
for i in range(n-1):   # Nya noder kläms in efter last
    p = Node()
    p.next = last.next
    last.next = p
self.cursor = p        # Båda pekarna ställs på
self.first = p         # samma ställe
```

För att tvinga användaren att anropa metoden `setsize` före användning av ringbufferten kan man byta namn på metoden till `__init__(self,n)`. Då anropas den direkt när ett objekt skapas. Alltså blir `buffer = Ringbuffer(80)` samma sak som dom båda satserna

```
buffer = Ringbuffer()
buffer.setsize(80)
```

Metoden `__init__` kallas *konstruktor*. Om den finns anropas den alltid när ett objekt skapas. Objektfanatiker brukar tycka att varje klass ska ha konstruktor, men för det mesta går det bra utan. När ett objekt ska innehålla en hel lista (som ringbufferten) skapas listan ofta bäst av en konstruktor.

8.5 Hemma hos datorn: Operativsystemet

Det finns gott om stackar, köer och ringbuffertar i operativsystemet. I förra kapitlet tog vi upp anropsstackarna, och det finns oftast en anropsstack för varje program som körs. Ofta finns det också en avbrottsstack för yttre händelser som avbryter körningen tillfälligt och måste tas om hand. Det gäller insignaler, musklick, tangenttryck osv.

Köer finns för skrivarjobb och för processer som avlöser varandra i exekveringen. I mer avancerade operativsystem kan man ge vissa jobb högre prioritet så att dom smiter förbi andra jobb i kön. I kapitel 10 ska vi se hur prioritetsköer fungerar.

Ringbuffertar används alltid när data skickas mellan datorer och till ansluten hårdvara om signalbehandlingen inte är synkron. En intressant användning är svarta lådan, mest känd från flygplan men vanlig också i datorer. Ständig loggning görs men bara det allra senaste behöver vara kvar för haverikommissionen.

8.6 Varningar och råd

Stackar arbetar LIFO (last in, first out). Försök aldrig ta ut understa tallriken ur en stack!

Köer arbetar FIFO. Försök aldrig komma åt det nyss inlagda.

Tom stack/kö får man inte göra pop/get på. Kolla först med `q.isempty()`.

Abstrakt stack är något som kan anropas med push och pop (med förväntat resultat).

Länkade listor kan implementera abstrakta stackar pch köer, men andra implementationer är möjliga.

8.7 Övningar

81. Skriv några satser som byter plats på dom båda översta värdena i en abstrakt stack.

82. Du vill vända bakochfram på en kö. Till din hjälp har du en stack. Hur gör du?

83. Du vill vända bakochfram på en kö. Till din hjälp har du en rekursiv tanke. Vilken?

84. I en stack finns damer underst och herrar överst, varje grupp för sej i personnummerordning. Du vill ha damerna överst men annars samma ordning. Till din hjälp har du en kö. Hur gör du?

85. En kö innehåller personposter i personnummerordning. Det kan finnas dubbletter. Hur bli man av med dom?

86. En stack av herrar och en kö av damer är båda ordnade i personnummerordning. Hur sorterar man ihop dom i kön utan hänsyn till kön?

87. Två stackar med okänt antal poster finns. Hur lägger man över alla poster i den större stacken?

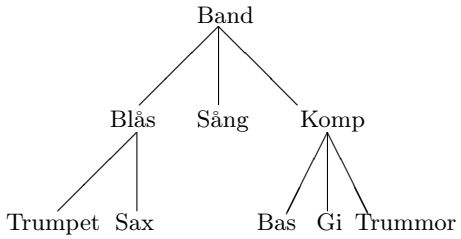
88. Du har en abstrakt stack med n personnamn samt en tom abstrakt kö. Hur många anrop (ungefär) krävs för att bestämma n? Stacken ska vara oförändrad efteråt.

89. Du har en abstrakt kö med n personnamn samt en tom abstrakt stack. Hur många anrop (ungefär) krävs för att bestämma n? Kön ska vara oförändrad efteråt.

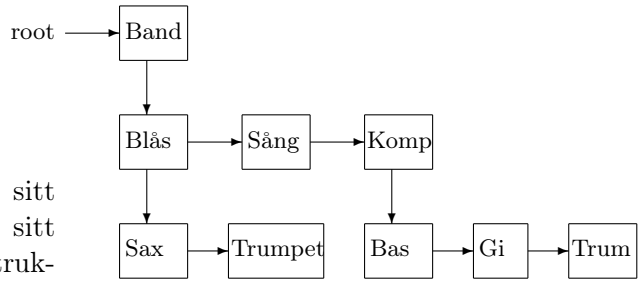
9 Träd och sökning

Datalogiska träd har roten upptill och löven längst ner – så är det bara! Sådana datastrukturer passar för att beskriva en *hierarki*, till exempel den katolska med påven överst och kardinaler, biskopar och vanliga präster på lägre nivåer. Eller den militära med överbefälhavaren överst och generaler, överstar, majorer, kaptener, löjtnanter osv under. Men det är lika användbart för att beskriva ett bokverk i flera delar med många kapitel uppdelade i sektioner och så vidare.

Ett rockband har den här trädstrukturen, där roten har tre barn som i sin tur har barn. Det kan verka som om man skulle behöva nodobjekt med tre pekare för att bygga datastrukturen, men förvånansvärt nog räcker det med två pekare!




```
class Node:
    value= ""
    down = None
    right= None
```

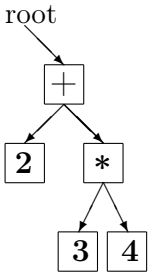


Man pekar alltså nedåt mot sitt äldsta barn och åt höger mot sitt närmaste syskon. Med den datstrukturen kan man framställa alla träd. Och anropet `write(root)` skriver ut hela trädet så som man är van att se innehållsförteckningen i en bok.

```
def write(p):
    if p==None: return
    print p.value
    write(p.down)
    write(p.right)
```

```
Band
   Blås
      Sax
      Trumpet
   Sång
   Komp
      Bas
      Gitarr
      Trummor
```

9.1 Binärträd



I ett binärträd har varje nod högst två barn, `left` och `right`. En användning av binärträd är för aritmetiska uttryck av typen $2 + (3 * 4)$. Observera att parenteser inte behövs i trädet. Om någon frågar vad uttryckets värde är kan man ge ett rekursivt dumsvar: värdet av vänsterträdet plus värdet av högerträdet! Med den här definitionen kommer `value(root)` att ge svaret 14.

```
def value(p):
    if p.value=="+": return value(p.left)+value(p.right)
    elif p.value=="*": return value(p.left)*value(p.right)
    else: return p.value
```

Fyra rekursiva anrop i den korta koden!

Nästan all binärträdsprogrammering är rekursiv och det beror på att binärträdet kan beskrivas rekursivt: Ett binärträd är en rotnod med ett binärträd under sej till vänster och ett annat binärträd under sej till höger. Tomma trädet räknas också som binärträd.

Oskyldig fråga: Hur många noder finns i binärträdet?
Rekursivt svar: Antalet i vänsterträdet plus antalet i högerträdet plus 1.

Basfallet är att ett tomt träd har noll noder.

```
def size(p):                # anropas size(root)
    if p==None: return 0
    x = size(p.left)
    y = size(p.right)
    return x+y+1
```

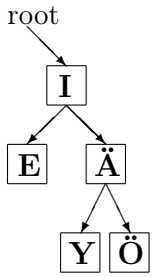
Ett binärträd med n nivåer har $2^n - 1$ noder om alla nivåer är fyllda.

Oskyldig fråga: Hur många nivåer har binärträdet?
Rekursivt svar: Max av antalen nivåer i vänster- och högerträden plus 1.

Basfallet är att ett tomt träd har noll nivåer.

```
def levels(p):              # anropas levels(root)
    if p==None: return 0
    x = levels(p.left)
    y = levels(p.right)
    return max(x,y)+1
```

9.2 Sökträd



Ett sökträd är ett binärträd som är sorterat så att mindre värden finns till vänster och större till höger. Om man har text är det bokstavsordning som gäller. Den som undrar om **U** är en mjuk vokal kollar först rotposten. Där står **I** och eftersom I är mindre än U går man ner till höger. Där står **Ä** och eftersom U är mindre än Ä går man till vänster. Där står **Y** och eftersom U är mindre än Y går man till vänster och når `None`. Alltså finns inte U i trädet.

```
def exists(x):
    p=root
    while p!=None:
        if x==p.value: return True  # Fanns!
        elif x<p.value: p=p.left    # Sök vänster
        else: p=p.right             # Sök höger
    return False                    # Fanns inte
```

Hoppsan, det fanns inte ett enda rekursivt anrop i den koden! Då får vi skriva en sådan version också.

```
def finns(p,x):                           # Anropas finns(root,17)
    if p==None: return False              # Fanns inte
    if x==p.value: return True            # Fanns!
    elif x<p.value: return finns(p.left,x) # Sök vänster
    else: return finns(p.right,x)         # Sök höger
```

Praktiskast är att göra binärträdet abstrakt med metoderna `put(key,info)`, `get(key)`, `exists(key)` och `writeall()`. Söknyckeln `key` kan vara personnumret och `info` är informationen om personen. Används så här:

```
from bintree import Bintree
tree = Bintree()                          # Tomt binärträdsobjekt
tree.put("420119-0818","Pythonkramare")   # Trädet fylls på
 - - -
tree.writeall()                           # Allt i pnr-ordning
pnr = raw_input("Personnummer: ")         # Söknyckel läses in
if tree.exists(pnr): print tree.get(pnr) # Info skrivs ut
else: print "Finns inte"
```

Filen `bintree.py` kan se ut så här ungefär.

```
class Node:
    key = None
    info = None
    left = None
    right = None
class Bintree:
    root = None
    def writeall(self):
        skriv(self.root)            # definieras nedanför klassen
    def exists(self,pnr):
        return finns(self.root,pnr) # definieras nedanför klassen
    def get(self,pnr)
        return alltom(self.root,pnr)# definieras nedanför klassen
    def put(self,pnr,yrke):
        ny = Node()                 # Ny nod skapas...
        ny.key = pnr
        ny.info = yrke
        if self.root==None:         # ...om trädet tomt, lägg in...
            self.root = ny
            return
        p = self.root               # ...börja annars en sökning.
        while pnr!=p.key:
            if pnr<p.key:
                if p.left==None:
                    p.left=ny       # Om tomt läggs nya noden in,
                else: p = p.left    # annars går sökningen vidare.
            elif pnr>p.key:
                if p.right==None:
                    p.right = ny    # Om tomt läggs nya noden in,
                else: p = p.right   # annars går sökningen vidare.

def skriv(p):                       # Skriver trädet i INORDER
    if p==None: return              # Basfall tomt träd
    skriv(p.left)                   # Skriv vänsterträdet
    print p.key,p.info              # Skriv roten
    skriv(p.right)                  # Skriv högerträdet
def finns(p,x): - - -               # Har vi redan programmerat
def get(p,x):   - - -               # Kan du få skriva själv
```

Termerna *preorder, inorder, postorder* betecknar tre olika sätt att skriva ut ett binärträd rekursivt, nämligen roten-vänster-höger, vänster-roten-höger och vänster-höger-roten. För vårt $2 + (3 * 4)$-träd blir utskrifterna så här.

```
 PREORDER       INORDER      POSTORDER
+ 2 * 3 4      2 + 3 * 4      2 3 4 * +
```

Inorder får man om man sätter sej på roten och plattar ut trädet. Det ger en ordnad utskrift av ett sökträd. Preorder gav den bästa utskriften av vårt rockband och alla andra sådana allmänna träd. Det är också den bästa ordningen att spara ett binärträd på fil mellan körningarna. Postorder är bäst när man ska skicka efter katalogvaror men är också den ordning som en HP-kalkylator använder (kallas också omvänd polsk notation).

9.3 Komplexitet

Om man räknar antalet jämförelser som behövs för att söka fram en viss post bland N poster får man sökmetodens *komplexitet*. Den är ett mått på hur lång tid sökningen tar.

	Genomsnitt	Värsta fallet	Kommentar
Dum linjärsökning i lista	N	N	bryter inte vid träff
Linjärsökning i lista	$N/2$	N	bryter vid träff
Binärsökning i lista	$\log N - 1$	$\log N$	ordnad lista
Binärträdssökning	$\log N - 1$	$\log N$	balanserat
Binärträdssökning	$1.4 \log N$	N	obalanserat

Ett binärträd med n fyllda nivåer innehåller ungefär 2^n noder. Samma sak kan uttryckas som att ett balanserat binärträd med N noder har ungefär $\log N$ nivåer. Tvålogaritm är nämligen det som gäller inom datalogi. Oftast hittar man det man söker i ett löv, men ibland hittar man det tidigare och genomsnittet blir därför lite lägre. Ett slumpmässigt byggt träd blir inte helt balanserat och därför blir genomsnittet lite större.

När man bygger ett binärträd gäller det att se till att inte posterna är ordnade. Då får man en lång tarm i stället för ett grenigt träd. Det kan lätt hända om man har personnumret som nyckel eftersom nya poster oftast har sent födelseår. Ett känt trick är då att använda bakvänt personnummer som nyckel.

Det är ganska krångligt att ta bort en nod ur ett sökträd, till exempel en person som inte betalt medlemsavgiften. Därför brukar man låta noden vara kvar men markera i `info`-objektet att det är en spöknod. Om personen betalar avgiften är det lätt att förklara noden levande igen.

9.4 Hemma hos datorn: Filsystem

Ju fler gigabyte minne datorn har, desto viktigare blir det att man snabbt kan hitta den fil man vill komma åt. Filsystemet är ett allmänt träd av kataloger med rotkatalogen högst upp och många nivåer av underkataloger. I varje katalog finns en lista med filnamn och motsvarande minnesadresser. Fram till år 1993 använde dom viktigaste filsystemen linjär sökning i fillistan, men nu har man övergått till binärträd.

Det är ibland nödvändigt att kolla igenom alla filer, till exempel för att jaga virus eller för att flytta på filer så att utrymmet utnyttjas bättre (defragmentering) eller för att säkerhetskopiera (ta backup). Då kan man använda den rekursiva tanken från rockbandsexemplet, och det görs också. I UNIX kan emellertid filkatalogen förutom de egna filerna ha länkar till filer i helt andra kataloger. Det gör en rekursiv genomgång för riskabel – tänk om länkarna skapar en slinga och därmed oändlig rekursion! Lösningen är att ha en stack för länkarna och ett minne för vilka kataloger man redan har varit i. Det kallas djupetförstsökning och kommer i nästa kapitel.

9.5 Varningar och råd

Roten på ett datalogiskt träd finns upptill.

Löven på ett datalogiskt träd finns nertill.

Allmänna träd har pekarna `down` och `right`.

Binära träd har pekarna `left` och `right`.

Aritmetiska uttryck i trädform behöver inga parenteser.

Preorder ger naturlig utskrift av allmänna träd

Inorder ger naturlig utskrift av binära sökträd

Postorder används av HP-kalkylatorn.

9.6 Övningar

91. Förbättra rockbandsutskriften så att det blir snygga indrag.

92. Varför anropas `skriv` via `write`? Varför inte anropa `skriv(tree.root)` direkt?

93. a. Om man skriver ut ett sökträd i inorder på fil och nästa dag bygger upp trädet från filen, hur ser trädet ut?
    b. Samma fråga för preorder.
    c. Samma fråga för postorder.

94. a. På vilket sätt är ett binärträd överlägset en ordnad lista?
    b. På vilket sätt är den ordnade listan överlägsen binärträdet?

95. Den ordnade listan `SAOL` består av cirka 200000 ord. Ge en rekursiv tanke för att bygga ett balanserat träd av den.

96. a. Man tänker skriva ut ett sökträd till en stack och senare bygga upp trädet igen. Vad är lämpligt - pre-, in- eller postorder?
    b. Samma fråga för en kö.

97. a. Ge en rekursiv tanke för antalet löv i ett binärträd.
    b. Ge en rekursiv tanke för antalet löv i ett allmänt träd.

98. Ett binärträd är *tvåbent* om alla inre noder har två barn.
    a. Ge en rekursiv tanke för att kolla om ett träd är tvåbent.
    b. Bevisa att det finns ett löv mer än vad det finns inre noder.
    c. Hitta felet i ditt bevis!

99. Om man med komplexitet menar antalet anrop, vilken komplexitet har
    a. ... utskrift i inorder av ett binärträd med N noder?
    b. ... uppbyggnad av ett sökträd med N noder?

10 Problemträd

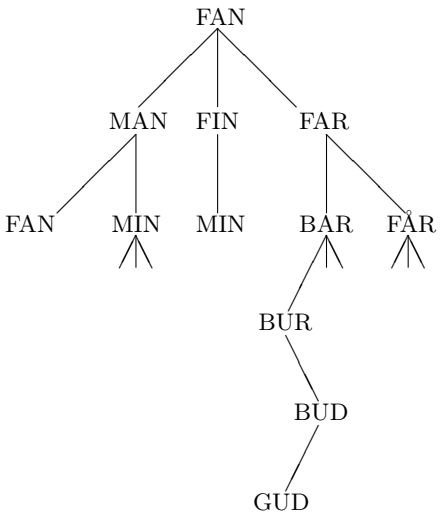
Att hitta billigaste resvägen, att lösa en sudoku, att ta sej igenom en labyrint eller att växla pengar lönsamt – det är några uppgifter som ger upphov till *problemträd*. Det gemensamma är att ett starttillstånd är givet och att man steg för steg tar sej fram till ett önskat sluttillstånd.

Alla tillstånd man kan nå bildar ett träd och det gäller att skapa och kolla noderna på smartaste sätt. Beroende på problemställningen väljer man olika sätt att gå igenom trädet.

10.1 Bredden först

Det gäller att gå från `FAN` till `GUD` via riktiga ord genom att byta ut en bokstav i taget. Stamfar `FAN` har sjutton barn varav tre syns på bilden. Barnbarnen bildar nästa nivå i trädet osv. Sex nivåer ner finns `GUD` via vägen

```
FAN - FAR - BAR - BUR - BUD - GUD
```



Men kanske är detta inte kortaste vägen till gud. För att vara säkra på att vi funnit den måste vi ha gått igenom alla nivåer ovanför först. För en *breddenförstsökning* behövs en kö. Man kunde tro att varje nivå skulle ha sin egen kö, men det räcker med en kö om man gör så här

- Lägg stamfar `FAN` i kön.
- Upprepa följande tills du hittar `GUD`.
  - Ta ut det första ordet ur kön.
  - Skapa barnen och lägg sist i kön.

För att man ska veta hur man kommit till gud är det lämpligt att låta varje ord peka på sin far. Då gör man förstås ett paket som innehåller ett ord och en faderspekare. På det sättet blir problemträdet inte bara en papperskonstruktion utan också en datastruktur.


```
class Node:
    word = ""
    father = None
```

Några av barnen ser ut exakt som en förfader, till exempel `MAN`s son `FAN`. En sådan *dumson* är det ingen mening att gå vidare med, eftersom den kortaste vägen aldrig går genom en dumson. Men hur känner man igen en dumson? Jo, man noterar alla ord man ser i ett minne och jämför sedan varje nyfödd son med det man har i minnet. För att sökningen ska gå fort är ett binärträd lämpligt minne. Och då blir man också av med dumsonen `MIN` som har ett likadant ord till vänster. En väg till gud via det ena `MIN` kan lika gärna gå genom det andra `MIN`.

Bredden först ger alltid den kortaste lösningen. Om man vill hitta kortaste vägen ut ur en labyrint är breddenförst lämpligt, **men** – bara om man har läst in kartan i datorn. Det är en värdelös algoritm IRL! Hur ska man kunna lägga in rummen i en kö? Nej, då är det dags för nästa problemträdsalgoritm.

10.2 Djupet först

Du står i ett rum med flera utgångar och du chansar på en av dom. Den ledde till ett annat rum med flera utgångar och du chansar på en av dom. Hoppsan, det var en återvändsgränd! Tillbaka och pröva ny utgång osv.

Algoritmen heter *djupetförstsökning* och kräver att man minns var man redan har varit. Ett effektivt redskap är ett garnnystan med garnänden fastknuten vid starten. Det kallas *Ariadnes tråd* efter en forntida prinsessa på Kreta som hjälpte sin hjälte Theseus att hitta ut ur minotauruslabyrinten. Datalogen djupetförstsöker i stället genom att byta ut kön mot en stack. Stamfaderns söner pushas, den översta sonen poppas, dennes söner pushas, den översta sonsonen poppas osv.

När man hittar ut ur labyrinten är man inte säker på att ha tagit kortaste vägen, men troligen ger man sej inte in igen för att söka en kortare väg. Den som söker kortaste vägen till `GUD` kan också använda djupetförstsökning, men får då inte avbryta första gången `GUD` visar sej. Det kan ju komma en bättre `GUD` när sökningen fortsätter.

Ibland har problemträdet oändligt många nivåer och då kan det hända att djupetförstsökningen bara fortsätter neråt. Om man inte tar bort dumsöner kan ju vägen bli

```
FAN - MAN - FAN - MAN - FAN ...
```

Ett exempel där djupetförstsökning måste användas är följande. Man har glömt sin portkod (fyra siffror) och måste pröva igenom alla tiotusen kombinationer. En enkel men tidskrävande knapptrycksföljd är då
`0000000100020003000400005...9999`

med fyrtiotusen siffror. Men det finns kanske en smartare följd av typ `000019326607100479218804...8225` med bara tiotusentre siffror där ändå alla 4-följder förekommer bland `0000,0001,0019,0193,...`.

Problemträdets urmoder kan vara följden `0000` och barnen skapas genom att man lägger till en siffra på slutet. Urmodern får dumsonen `00000` och nio andra barn som vart och ett får tio barn osv. Redan efter tio nivåers breddenförstsökning är datorns minne överfullt, och vi som skulle skapa tiotusen nivåer! Men djupetförstsökning fungerar. För att upptäcka dumsöner måste man bara kolla att den nya 4-följden inte finns tidigare i följden.

```
def makesons(s):
    if len(s)==10003:              # Lösningen funnen.
        print s
        quit()
    for tkn in "0123456789":       # Skapa alla söner och,
        if s[-3:]+tkn not in s:    # om dom inte är dumsöner,
            stack.push(s+tkn)      # pusha dom på stacken.

from stack import Stack
stack = Stack()
stack.push("0000")                 # Urmodern pushas först.
while True:
    mor = stack.pop()
    makesons(mor)
```

Ofta kan man klara djupetförstsökningen utan stack, genom rekursion. Programmet blir ofta både kortare och obegripligare.

```
def complete(s):                   # Komplettera påbörjad lösning.
    if len(s)==10003:              # Lösningen funnen.
        print s
        quit()
    for tkn in "0123456789":       # Skapa alla söner och,
        if s[-3:]+tkn not in s:    # om dom inte är dumsöner,
            complete(s+tkn)        # komplettera sonen.
complete("0000")                   # Komplettera urmodern.
```

Om man provkör med tvåsiffrig portkod får man lösningen blixtsnabbt. Men redan med tresiffrig kod kommer felutskriften `Maximum recursion depth exceeded`. Normalt tillåter nämligen Python bara rekursionsdjupet 1000.

I *sudoku* ska man komplettera en delvis ifylld 9×9-sifferruta så att varje rad och kolumn innehåller siffrorna 1 till 9, och samma för nio markerade 3×3-rutor. Den givna matrisen är urmoder och i varje nivå i trädet fyller man i en siffra på den första lediga platsen. Det kan alltså bli upp till nio barn.

Breddenförst eller djupetförst? Eftersom vi inte är ute efter kortaste lösningen (alla är ju lika långa) finns det ingen anledning att söka breddenförst. Risken är att kön sväller över alla gränser, så djupetförst är bäst. Stack eller rekursion? Med bara cirka femtio nivåer är det inget problem med rekursionsdjupet, så då kan man gott satsa på en rekursiv lösning.

Ett enklare problem av samma typ är att sätta ut åtta damer på ett schackbräde utan att två damer står i samma rad, kolumn eller diagonal. Urmodern är ett tomt bräde, första nivån har en dam på första raden, andra nivån damer på första och andra raden etc. En utplacering kan alltså beskrivas med en lista av åtta tal, där `queen[i]` anger vilken kolumn damen på rad `i` står i.

```
n=8
queen=[None]*n

def complete(row):              # Fullborda lösning som har damer på rad 0..row-1.
    if row==n: - - -            # Skriv ut färdig lösning.
    for col in range(n):
        if okay(row,col):
            queen[row]=col
            complete(row+1)

def okay(row, col):             # Kolla att den inte slår tidigare damerna
    for i in range(row):
        if queen[i]==col:       return False   #rakt nedanför
        if queen[i]-col==row-i: return False   #snett nedanför
        if col-queen[i]==row-i: return False   #snett nedanför
    return True

complete(0)                     # Komplettera ett tomt bräde
```

Det visar sej finnas 92 lösningar.

10.3 Bästa först

Labyrintprogrammet kan också användas för alla resruttsproblem. Djupetförstsökning blir då otänkbar eftersom problemträdet har nästan oändligt djup. Breddenförstsökning efter en viss slutstation minimerar antalet stationer som passeras.

Om man i stället vill minimera tiden eller kostnaden för resan kan man varken använda stack eller kö. Det som behövs är en *prioritetskö* och en sån ska ha tre anrop.

```
q.put(pris,x)              # Värdet x prismärks
x = q.get()                # Värdet med lägst pris plockas ut
if q.isempty(): - - -      # Om kön är tom - - -
```

Om det gäller att hitta billigaste resan får `pris` vara totalpriset för resan till platsen, dvs lägst pris har alltid högst prioritet. Trädgenomgången kommer att bli i prisordning, där startplatsen har priset noll osv. När slutstationen dyker upp första gången vet man att man funnit billigaste resvägen.

Snabbaste resan hittar man på samma sätt genom att låta `pris` vara ankomsttiden. Trädgenomgången sker i tidsföljd, där startplatsen har starttiden som ankomsttid osv.

En oväntat svår problemtyp är *kappsäckspackning*. Om man vill packa sina föremål av olika vikt eller volym eller form i så få kappsäckar som möjligt är det en bästaförstsökning med antalet hittills använda kappsäckar som pris. Men om man vill packa en kappsäck så full som möjligt finns det inget användbart prisbegrepp. Eftersom man måste gå igenom hela problemträdet kan man lika gärna göra det djupetförst med rekursion – minst minneskrävande och ofta enklast att programmera. När man inte hittar något bra sätt att gå igenom trädet kan det vara bättre att pröva dynamisk programmering, som tas upp i nästa avsnitt.

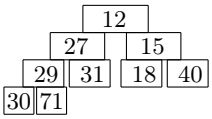
En prioritetskö ger en enkel sorteringsalgoritm. Stoppa in allt i kön så kommer det ut ordnat!

```
for rad in open("persondata.bas"):
    pnr,namn = rad.split()
    q.put(pnr,namn)        # Vad hade q.put(namn,pnr)
while not q.isempty():     # gett för effekt?
    pnr,namn = q.get()
    print pnr,namn
```

Hur programmerar man en prioritetskö? Ett långsamt sätt är att låta varje `get`-anrop leta igenom hela kön. Snabbare är att låta `put` sortera in den

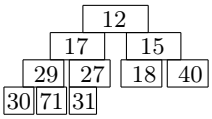
nya posten på rätt ställe. Men i båda fallen blir komplexiteten proportionell mot N, köns längd. Det finns en annan datastruktur som heter *heap* eller på svenska *trappa*, och där både *put* och *get* har komplexiteten $\log N$. Och då ska man påminna sej att om N är en miljon så är $\log N$ bara 20.

10.4 Trappa eller heap



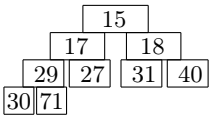
Det som bilden visar är en sorts binärträd med billigaste priset i roten och regeln att en son alltid har högre pris än sin pappa. Alla nivåer utom den understa är fyllda. Bilden visar bara priset men egentligen är det ett prismärkt objekt i varje fack.

`q.put(17,x)`



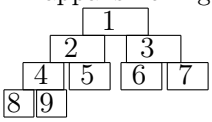
Vi ska fundera ut vad anropet `q.put(17,x)` ska göra. Eftersom alla nivåer utom den understa redan är fyllda lägger vi in 17 på första lediga plats, alltså vänstra platsen under 31. Men pappan ska vara billigare än sonen, så 31 och 17 får byta plats. Nu står 17 under 27 och då får dom byta plats. Sedan står 17 under 12 och det är okej. Den här kedjan av byten kallas *upptrappning*.

`x = q.get()`



Vad ska sedan anropet `x=q.get()` göra? Objektet med priset 12 går till `x` och lämnar efter sej ett tomrum. Det måste fyllas med det sista objektet på understa nivån, alltså 31. Men pappan ska vara billigare än sonen så 31 måste byta plats med sin billigaste son 15. Och sedan måste 31 byta plats med sin billigaste son 18. Sedan är det okej och denna byteskedja kallas *nedtrappning*.

Trappans hemlighet:



Trappans hemlighet är att den egentligen bara är en vanlig indexerad lista där roten har index 1. Man klarar sej helt utan såväl faderspekare som sonpekare eftersom följande gäller.

- Om sonen har index `i` har fadern index `i//2`.
- Om fadern har index `i` har sönerna index `2*i` och `2*i+1`.

Koden för upptrappning ser då ut så här. Det är `t[i]` som ska trappas upp.

```
def upptrappa(t,i):
    j = i//2                        # j är far till i
    while t[j].pris>t[i].pris:      # Om far dyrare än son ...
        t[i],t[j] = t[j],t[i]       # ... byt far och son!
        i = j; j = i//2
```

Nedtrappning av nyinlagd `t[1]` blir så här.

```
def nedtrappa(t,n):
    j = 1; i = 2                    # i är vänstra sonen
    while i<n:                      # trappa ner
        if t[i].pris>t[i+1].pris:  # om högra sonen billigare
            i = i+1
        if t[j].pris>t[i].pris:    # om far dyrare än son
            t[i],t[j] = t[j],t[i]
            j = i; i = 2*j
        else: break
    if i==n and t[j].pris>t[i].pris: # enkelson
            t[i],t[j] = t[j],t[i]
```

Filen `heap.py` innehåller klassen och metoderna.

```
class Node:
    pris = -999999                  # t[0] får negativt pris
    value = None


class Heap:
    N = 1001                        # maxantal
    t = []                          # trappan
    n = 0                           # aktuellt antal
    def __init__(self,N):           # anropas heap = Heap(500)
        self.N = N
        self.t = [None]*N
        for i in range(N):
            self.t[i] = Node()
    def put(self,p,x):
        self.n+=1                   # hoppas att n<N
        self.t[self.n].pris = p     # lägg in sist
        self.t[self.n].value = x
        upptrappa(self.t,self.n)
    def get(self):
        p = self.t[1].pris          # plocka ut första
        x = self.t[1].value
        self.t[1] = self.t[self.n] # ersätt med sista
        self.n-=1
        nedtrappa(self.t,self.n)
        return p,x
```

Metoderna `peek()`, `isempty()` och `isfull()` bör också finnas och består av en självklar sats vardera.

Om upptrappningen går ända upp till `t[1]` kommer `while`-slingan att jämföra `t[1]` med `t[0]`. Därför låter man alltid `t[0]` ha ett oslagbart lågpris.

Om man vänder på alla olikheter och kallar `pris` för `prio` får man en maxprioritetskö där `p.get()` returnerar objektet med högst `prio`. Den är lika vanlig som minpriskön.

En typisk användning av en minpriskö är i *händelsestyrd simulering*. En dag på ICA har bland annat dessa händelsetyper.

1. Ny kund kommer in och börjar handla.
2. Kund ställer sej i kortaste kassakön med ett antal varor.
3. Kassören tar sej an nästa kunds varor.
4. Kund betalar och går
5. En till kassakö öppnas.
6. En kassakö stängs.

Genom simulering får man veta hur långa väntetiderna är vid olika kassabemanning. Idén är att köa framtida händelser med `q.put(time,event)`. När kunder kommer, hur många varor dom köper osv slumpas fram. Kunderna är objekt och kassaköerna är abstrakta köer.

10.5 Dynamisk programmering

Kortaste vägen från fan till gud måste vara ett steg längre än kortaste vägen till den närmaste av guds grannar och det är ju en rekursiv tanke. Men oanvändbar eftersom den i onödan går igenom alla jättelånga ordkedjor också. Dynamisk programmering kallas det när man först löser dom enklare fallen och använder resultaten från dom för att komma vidare. Först noterar vi i ordlistan att gud själv har avståndet noll men alla övriga oändligt avstånd. Sen går vi igenom ordlistan och sätter avståndet $n + 1$ på alla grannar till ord med avståndet n, men vi höjer aldrig ett redan satt avstånd. När fan får sitt avstånd är det vårt svar, men på vägen har vi fått reda på avståndet till gud för en massa andra ord.

I det här fallet ger inte dynamisk programmering något annat än vad vår tidigare breddenförstsökning gav. Men kappsäckspackning är ett bättre exempel. Anta att kappsäckar får väga 20 kg på flyget och att vi vill packa

så fullt det går med tre varutyper som väger 457 g, 843 g och 900 g. Om vi visste att det bästa vore att ta tio stycken av den tyngsta skulle vi ha ett enklare kappsäcksproblem att lösa, nämligen att packa en kappsäck som får väga 11 kg med varor av dom två första slagen. Nu vet vi inte att det ska vara just tio av den tyngsta, så vi får lösa tjugoen olika kappsäcksproblem med kappsäckar på 20 kg, 19.1 kg, 18.2 kg etc. Vart och ett av dom kan vi på samma sätt reducera till ett antal kappsäcksproblem med mindre kappsäck och bara ett varuslag.

I efterhand ser man oftast att dynamisk programmering lika gärna kunnat göras med till exempel breddenförstsökning, men det är ändå ett bra sätt att tänka.

10.6 Hemma hos datorn: Prioritet och resursdelning

I datorn pågår flera processer samtidigt och det kommer signaler från yttre källor som musen, tangenterna och internet. Vad som ska göras först bestäms av prioriteten. Processer som virusskanningar har låg prioritet och vissa tangenter som ctrl-C och ctrl-alt-del har hög prioritet. Så det finns förstås prioritetsköer i operativsystemet.

Men förvånansvärt ofta blir det problem när flera processer använder samma resurs, till exempel en databasfil. Tänk på en bankomat som kollar att det finns pengar på ett konto och sedan gör uttaget. Om en högre prioriterad process bryter in mitt i och mekar med kontot blir det fel och det får inte hända. Därför låser databasen kontot under den tid som bankomattransaktionen pågår. En transaktion som ska föra över ett belopp från ett konto till ett annat måste låsa båda kontona. Då händer det att två processer låst var sitt konto och båda försöker komma åt det andra. Det kallas *baklås* (eng. deadlock) och måste undvikas på något sätt, till exempel så att processen väntar en viss tid och sedan gör något annat ett tag. Men då kan det bli ett *framlås* (eng. livelock) om båda processerna ändrar sej på samma sätt. Två hungriga japaner med var sin ätpinne är exempel på baklås och två artiga japaner som går åt sidan för varandra i en trång korridor blir ett framlås.

10.7 Varningar och råd

Breddenförstsökning ger kortaste lösning i problemträdet.

Djupetförstsökning ger vänstraste lösning i problemträdet.

Bästaförstsökning ger billigaste lösning i problemträdet.

Kö/stack/trappa ger genomgång bredden/djupet/bästa först.

Rekursiv djupetförstsökning behöver ingen stack.

Prioritetskö implementeras med en trappa, som är en delvis ordnad lista.

Trappans komplexitet är $\log N$ för varje `put/get`.

10.8 Övningar

101. Du står vid en sjö med en sjulitershink och en tolvlitershink och vill mäta upp exakt en liter vatten. Rita problemträd och ange datastrukturer!

102. Växelkurserna mellan n valutor bildar en $n \times n$-matris. Finns det någon lönsam växlingskedja från kronor till kronor? Rita problemträd och ange datastrukturer!

103. Vilken är längsta ordkedjan av typen

     ```
     Å -> ÅR -> VÅR -> VÅRD -> VÅRDA -> VÅRDAG
     ```

     och hur blir problemträdet och datastrukturerna?

104. Programmera en maxtrappa med en `peek`-metod och använd den i nästa uppgift.

105. Stadens silhuett skapas av rektangulära hus, beskrivna av tre tal: `xstart,y,xslut`, alltså vänsterkant, höjd och högerkant. Rita silhuetten!

     Ledning: Lägg först in horisonten som ett hus. Lägg husen i en mintrappa med `xstart` som pris, ta ut ett hus, avläs `x` och rita! Lägg tillbaka det i mintrappan men nu med `xslut` som pris. Lägg också huset i maxtrappan med`y` som prio. Peeka, avläs `y` och rita. När högsta huset i maxtrappan återkommer ut mintrappan tas det bort och nya höghuset ger nytt `y`, om det inte är ett dött hus för då tas det bara bort.

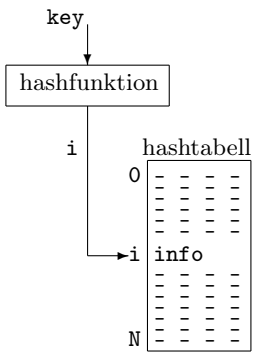
     Grafikfönster får man enklast med `import turtle` och sedan ritar man med `turtle.goto(x,y)`.

11 Sökning och sortering

Hur kan Google leta igenom många miljarder webbsidor på en sekund? Ett binärträd med alla ord ur dokumenten skulle få ett femtiotal nivåer och därmed vara både långsamt och otympligt. Lösningen är hashning, den kanske viktigaste datalogiska uppfinningen.

Hashning ger snabbaste sökningen efter ett ord i en databas men samtidigt sämsta överblicken över hela databasen. Det viktigaste redskapet för att ordna och samordna databaser är sortering och det är häpnadsväckande hur många olika metoder man kommit på för detta.

11.1 Hashtabeller



När man googlar på ett ord eller drar ett bankkort i en bankomat skickas ordet eller magnetkoden till en *hashfunktion* som på något sätt räknar fram och returnerar ett heltal i mellan noll och N. I datorn finns en *hashtabell* indexerad från noll till N och på tabellplatsen med index i finns det vi söker, alltså webbsidan eller bankkontot. Träff med en gång utan en enda jämförelse, det verkar för bra för att vara sant. Och det är det också!

Tillsammans med informationen `info` lagras alltid söknyckeln `key`. Då kan man kolla att man hittat det man sökte efter genom att jämföra söknyckeln och den lagrade, och oftast räcker denna enda jämförelse.

En hashfunktion ska alltså förvandla text till tal och naturligast är att använda bokstävernas ordningsnummer i ASCII-alfabetet. Här är en enkel hashfunktion.

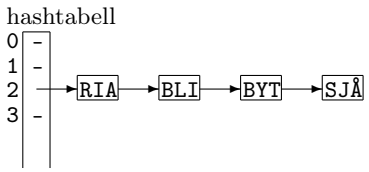
```
def hash(word):    # Summerar bokstävernas ASCII-koder
    h = 0
    for tkn in word:
        h+=ord(tkn)
    return h
```

Funktionen har en uppenbar brist, nämligen att orden ORM, MOR och ROM får samma hashvärde 238 så att det bli *krockar* i hashtabellen. Bokstävernas position måste alltså påverka deras bidrag till hashvärdet. En annan brist är att långlånga ord kan få så stort hashvärde att dom hamna utanför tabellen. Så här fixar vi båda bristerna.

```
def hash(word):    # Proffsig hashfunktion för N=1023
    h = 0
    for tkn in word:
        h = (h*155+ord(tkn))%1024
    return h
```

Nu får orden helt olika hashvärden (926, 546, 26) och värdet kan aldrig bli större än 1023, hur långt ordet än är. För datorn är det väldigt enkelt att räkna ut `k%1024` – det betyder nämligen "tio sista bitarna i `k`". Hashfunktionen blir alltså lite snabbare om tabellstorleken `N` är en tvåpotens. Läroböcker i datalogi skriver ofta att `N` bör vara ett primtal, men det är föråldrat. Talet 155 är större än största förekommande `ord(tkn)` men kan annars väljas fritt. Det bör dock vara ett udda tal, annars kan det bli underliga effekter. Byt till exempel 155 mot 32 och räkna ut hashvärden för ORM, ARM och ÄRM. Förklara resultatet!

En hashfunktion är *perfekt* om den aldrig ger några krockar men det är svårt att hitta en sådan. Om man hashar alla 749 svenska trebokstavsord med den proffsiga hashfunktionen ovan får man 181 krockar. Ungefär vart fjärde ord krockar alltså med ett redan inhashat ord och det är det typiska om man har en hashtabell med cirka femtio procents luft. Med det menar man att tabellstorleken 1024 är cirka femtio procent större än datamängden 749 (om nu $50 \approx 37$) och det är en vanlig tumregel för hur stor hashtabell man ska ta till.



Värsta krocken är mellan orden RIA, BLI, BYT och SJÅ som alla får hashvärdet 2. Problemet löser man med krocklistor enligt figuren. Noderna i listan har fälten `key`, `info` och `next` och man pushar in ett nytt objekt först i listan.

```
def put(key,info)
    nod = Node()
    nod.key = key
    nod.info = info
    i = hash(key)
    nod.next = hashtabell[i]   # Som man pushar något
    hashtabell[i] = nod        # på en stack, alltså.
```

Krocklistor är överlägset andra sätt att hantera krockar men är ändå inte det vanligaste och skälet till det är att det i gamla programspråk som C är svårt att spara en datastruktur med pekare på fil. I Python går det utmärkt

(se `pickle` i dokumentationen) men vi ska i alla fall nämna hur man annars gör.

Vi antar alltså att vi hashat `key` till ett index `i` som visar sej vara upptaget. *Linjär probning* kallas det när man då går vidare till `i+1` och kollar om den platsen är ledig. Annars går man till `i+2` och så vidare. Den som senare söker efter `key` hamnar förstås också på index `i` men upptäcker vid jämförelse att det inte är rätt nyckel som finns där. Då kollar man på `i+1` osv. Det här pågår tills man antingen hittat rätt eller också kommit till ett tomt fack. Då är man helt säker på att den sökta nyckeln inte finns.

Linjär probning leder ofta till hopklumpning, så att vissa nycklar måste krockas väldigt långt bort och då får lång söktid. Man kunde tro att det skulle bli bättre om man gick två steg i taget, alltså probade indexen `i+2`, `i+4`, `i+6` osv. Det hjälper inte ett dugg! Men om man ökar stegen efter hand, till exempel probar `i+1`, `i+3`, `i+6`, `i+10` osv blir det sällan hopklumpning. Det kallas *kvadratisk probning*.

Dubbelhashning är linjär hashning med steget `h`, där steget är olika för olika nycklar och bestäms av en annan hashfunktion. Då finns ingen benägenhet för hopklumpning, men en del lurigheter finns. Steget får givetvis inte vara noll men inte heller något annat jämnt tal om N valts som en tvåpotens. Då kan nämligen tabellen verka fullsatt när bara dom jämna indexen är slut.

11.2 Quicksort

Den snabbaste sorteringsalgoritmen heter quicksort. Den bygger på en överraskande vardagslivsalgoritm som kallas *Damerna först*. Anta att en lång personlista ska sorteras om så att damerna kommer först. Den bästa algoritmen är så här enkel:

DAMERNAFÖRSTSORTERING

- Sätt vänster pekfinger på första och höger på sista personen.
- Rör vänster pekfinger åt höger tills det pekar på en man.
- Rör höger pekfinger åt vänster tills det pekar på en kvinna.
- Byt plats på dom utpekade personerna.
- Upprepa tills fingrarna korsats, då står alla damer först!

Enkelt att programmera, men den här koden är inte helt felfri:

```
v=0; h=N-1
while True:
    while tab[v]==D: v+=1
    while tab[h]==H: h-=1
    if v>h: break
    tab[v],tab[h] = tab[h],tab[v]
```

För hur går det om listan bara innehåller herrar? Lätt att fixa förstås, men vi skjuter upp det så länge.

QUICKSORT

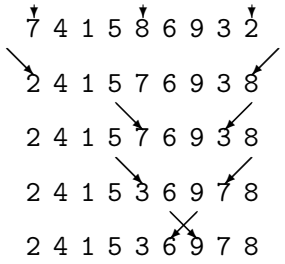
- Välj ett värde godtyckligt.
- Kalla mindre värden för damer och större för herrar.
- Gör damernaförstsortering.
- Quicksortera damerna inbördes.
- Quicksortera herrarna inbördes.

Damernaförstdelen kallas partitionering och delar listan i två delar som sorteras rekursivt var för sej. Ett lämpligt basfall är att dellistan har mindre än två element.

```
def quicksort(v,h):     # Sorterar dellistan mellan index v och h
    if v+2>h: return    # Inget att sortera
    m = partition(v,h)  # Damerna har nu index v till m
    quicksort(v,m)      # Damerna sorteras inbördes
    quicksort(m+1,h)    # Herrarna sorteras inbördes
```

Om partitioneringen är perfekt ger den lika många damer som herrar och då kommer rekursionsdjupet att bli $\log N$. Varje element jämförs en gång på varje nivå, så totalt blir det $N \log N$ jämförelser. Men har man riktig otur blir det bara en dam och resten herrar vid varje partitionering och det ger N rekursionsnivåer och totalt N^2 jämförelser! Sådan enorm otur har man väl inte? Jodå, om man som partitioneringsvärde väljer det första i listan och om hela listan råkar vara sorterad från början görs en onödig quicksortering med just N^2 jämförelser. Det är ju katastrof om det är en miljonlista! För en nästan sorterad lista är vanlig insättningssortering överläget.

Median-of three undviker katastroffallet genom att som partitioneringsvärde använda det mellersta av tre värden, nämligen dom som står först, mitt i och sist i listan. Man sorterar dom med det minsta först, det största sist och det mellersta i mitten. I exemplet blir 7 partitioneringsvärde. Fingrarna som rör sej mot varandra fastnar då på 7 och 3 och bytet görs. När fingrarna nästa gång fastnar på 9 och 6 har dom korsats och partitioneringen är klar.



Om alla element är lika skulle fingret kunna köra över kanten, därför är det säkrast att låta båda fingrarna fastna på partitioneringsvärdet.

11.3 Räknesortering och hålkortssortering

Quicksort, merge sort och trappsortering har alla i genomsnitt komplexitet proportionell mot $N \log N$, men quicksort är oftast den snabbaste algoritmen. Den kräver inte heller extra minnesutrymme utan gör bara byten inom listan. Men teoretiskt värsta fallet för quicksort är i alla fall proportionellt mot N^2; det kan vara värt att minnas.

Det är inte svårt att matematiskt bevisa att ingen sorteringsalgoritm kan vara snabbare än just $N \log N$. Vi ska nu se på två algoritmer som paradoxalt nog är snabbare än den teoretiska gränsen och har komplexitet som är proportionell mot N.

Exempel: För astrologiska syften vill man ha befolkningsdatabasen sorterad på stjärntecken. Hur många jämförelser krävs?

Quicksort skulle behöva $N \log N \approx 9000000 \cdot 23$, alltså tvåhundra miljoner jämförelser. Vi klarar det på arton miljoner jämförelser så här:

- Räkna hur många som är födda i varje tecken (N jämförelser).
- Avsätt i en lista rätt antal platser för varje stjärntecken.
- Gå igenom befolkningen och sätt in varje person i rätt listdel (N jämförelser).

Det kallas *räknesortering* (distribution count) och är bara användbart när man har ett litet antal kategorier, i exemplet tolv stycken.

Exempel: För datalogiska syften vill man ha befolkningen sorterad på personnummer. Hur många jämförelser krävs?

Quicksort skulle fortfarande behöva tvåhundra miljoner jämförelser. Vi klarar det enkelt på åttioen miljoner så här.

- Varje person ska vara ett hålkort i en enorm kortbunt.
- Dela upp bunten i tio småbuntar efter nionde siffran i personnumret.
- Lägg ihop dom i en jättebunt i ordning efter niondesiffran
- Gör om samma sak för åttonde, sjunde osv siffran i personnumret.
- Simsalabim – hela befolkningen är ordnad efter personnummer!

Hålkortstiden, som tog slut på 1970-talet, byggde på IBMs hålkortsmaskiner. Dom utförde den här sorteringen rent mekaniskt under starkt oväsen och därför kallas den *hålkortssortering* eller radix sort. Den är bara användbar om alla nycklar har samma längd och får då linjär komplexitet. Om nyckellängden är tio tecken blir komplexiteten $10N$ men för personnummer är kontrollsiffran ointressant för sorteringen.

11.4 Hemma hos datorn: Unix-rehash

När en unixanvändare skriver kommandon som `emacs pnyxtr.py` måste operativsystemet leta i många kataloger bland tusentals kommandofiler tills det hittar `/usr/local/bin/emacs`. För att sökningen ska gå blixtsnabbt hashas vid inloggningen alla körbara filnamn och informationen om var dom finns bifogas. Om man flyttar en kommandofil eller lägger till en ny katalog i sökvägen aktualiserar man hashtabellen med `rehash`.

Linjär probning har i alla år dömts ut av datalogiläroböckerna, men nu upptäcker man att den ger snabbare sökning än den hajpade dubbelhashningen. Förklaringen är att processorer nu använder *cache*, snabbminne. När ett värde hämtas ur det vanliga minnet hämtar man samtidigt ett helt avsnitt av minnet och lägger i cachen. Då nya värden efterfrågas kollar datorn först om dom kanske redan ligger i cachen. I så fall går åtkomsten tio gånger så fort. Vid linjär probning ligger positionerna med index `i+1`, `i+2` osv redan i cachen.

Att hasha något betyder egentligen att göra pyttipanna av det. Dom snabbaste hashfunktonerna gör just det, nämligen hackar upp och mixar bitarna i ordets binära kod. Att använda multiplikation, som vi har gjort här, är långsammare men lättare att analysera.

11.5 Varningar och råd

Perfekt hashning gör att varje sökning endast kräver en jämförelse.

Femtio procents luft i hashtabellen ger cirka 1.5 jämförelser i snitt.

Krockar hanteras bäst med krocklistor men probning är fortfarande vanligt.

Damernaförstsortering har komplexitet N och minimalt antal byten.

Quicksort har komplexitet $1.4\,N \log N$ i snitt men i värsta fall $N^2/2$.

Insättningssortering är ändå bäst när det nästan är sorterat från början.

Räknesortering vid ett litet antal kända kategorier har komplexitet $2N$.

Hålkortssortering när alla nycklar har längd k har komplexitet kN.

11.6 Övningar

111. En databas över världens största diamanter (omkring hundratusen olika) är sorterad efter försäljningspris (dollar) men du ska sortera om den efter vikt (karat). Vilken sorteringsmetod föreslår du?

112. Hur tar man bort en post ur en hashtabell
a) med krocklistor?
b) med linjär probning?

113. Man vill hasha sina fyratusen matrecept på uppsättningen av ingredienser. Om man söker på mängden {mjöl,ägg,mjölk,margarin} ska man hitta pannkakor, plättar, våfflor och crêpes. Föreslå hashfunktion!

114. En miljon dumbolotter säljs varje månad och för varje lott sparas lottnumret och köparen i en post. En lista med en miljon poster finns i datorn vid dragningen då tusen vinstnummer slumpas fram, ett efter ett.

    För varje vinst måste den osorterade listan letas igenom. Hur många jämförelser blir det? Lönar det sej att sortera listan först?

115. Femtio gånger per sekund når spänningen i vägguttaget ett maxvärde och via en AD-omvandlare skickas det till en pc. Programmet ska varje dygn rapportera dom tio högstanoteringarna och när dom inföll. Följande datastrukturer har föreslagits för uppgiften:

- Lista med 4,32 miljoner poster som insättningssorteras.
- Lista med 4,32 miljoner poster som quicksorteras vid dygnets slut.
- Hashtabell med storlek 6,48 miljoner hashad på spänningsvärdet.
- En trappa med plats för 4,32 miljoner poster.
- En trappa med plats för tio poster.
- Ett binärträd, sorterat efter spänningsvärdet.

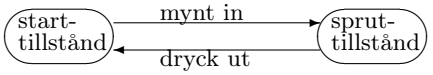
Ranka strukturerna från oduglig till jättebäst!

116. Alla nyfödda vägs och mäts. Exempelvis var Elon Kann 54 cm lång och vägde 3980 g. Databasen med tio miljoner poster ligger ordnad på personnummer, men man vill sortera om den så att korta barn kommer före långa barn. eftersom längd anges i hela centimeter låter man för lika långa barn vikten avgöra, med lätt före tung. Vilket sorteringsförfarande är snabbast?

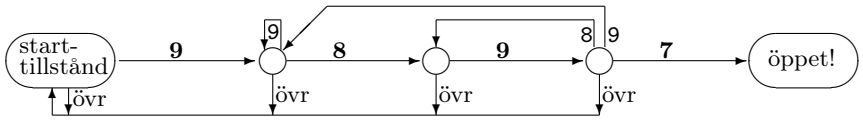
12 Automater och syntax

År 50 i Alexandria utkom Herons bok *Automata*. Stadsborna kände redan till automaterna där man stoppade i ett mynt och fick en läskande dryck eller en dockteatershow. Mer allmänt är en automat någonting som kan befinna sej i olika *tillstånd* (eng. states) och där olika *händelser* (eng. events) ger *övergång* (eng. transition) till nytt tillstånd och eventuellt till en *aktivitet* (eng. action).

Dryckesautomaten kan då beskrivas som en graf med två tillståndsnoder och två övergångspilar.



En låsautomat med portkoden `9897` blir mycket krångligare.



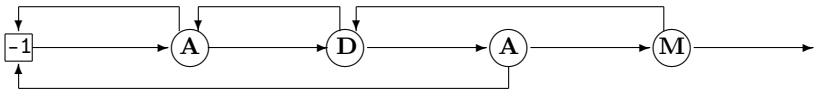
Från starttillståndet för `9` vidare till nästa tillstånd. Härifrån för `8` vidare, `9` ligger kvar och övriga för tillbaka till starttillståndet. Från nästa tillstånd är det bara `9` som för vidare, allt övrigt återgår till start. Krångligast är det i det sista tillståndet. Då har följden `989` mottagits och `7` för till målet men `8` backar bara ett steg. Varför? Jo, följden `989897` ska ju öppna porten.

Om tillstånden numreras `0,1,2,...` kan hela automaten definieras av sin transitionsfunktion. Anropet `transition(state,tkn)` returnerar tillståndet automaten övergår till om den befinner sej i `state` och tar emot `tkn`. Programslingan blir så här enkel.

```
while state!=finalstate:
    tkn = input()                # någon sorts input
    state = transition(state,tkn)
```

12.1 Knuthautomaten

Om automaten läser igenom en text kommer den att stanna första gången som teckenföljden `9897` dyker upp. Det blir en effektiv textsökning, men att konstruera automaten är komplicerat. Knuth, datalogins fader, kom på ett enklare sätt att rita automaten. Låt oss säga att det gäller att leta i bibeln efter ordet `ADAM`. Knuthautomatens pilar är omärkta och det finns en bakåtpil från varje tillstånd. Bibelläsning sker bara i framåtpilarna. Man börjar i tillståndet `-1` och upprepar följande.



- Läs en bokstav, gå samtidigt åt höger. Stanna om bokstaven står där.
- Backa annars, jämför åter med samma bokstav. Stanna om det stämmer.
- Backar annars osv, eventuellt ända till -1.

För bibeltexten `VADAN SKAPADES ADAM?` går automaten genom tillstånden

`* 0* 0 1 2 310* 0* 0* 0* 0 10* 0 1 2* 0* 0* 0 1 2 3 4`

där `*` är tillståndet `-1`. Om `next[i]` är dit man backar från tillstånd `i` beskrivs alltså hela automaten av listan `next`. Så här blir koden.

```
word="ADAM"
n=len(word)
i=-1
next=[-1,0,-1,1]
for tkn in bible.read():
    i+=1
    if i==n:
        print "Found",word
        break
    while i>=0 and tkn!=word[i]:
        i=next[i]
```

Vilken `next`-lista som gäller för ett visst sökord finns det en enkel algoritm för att räkna ut. Men det är också roligt att kunna tänka fram det och det roliga unnar vi läsaren. Så här kan man tänka:

- `next[0]=-1` och oftast blir övriga `next[i]=0`.
- `next[i]=j` om senaste `j` bokstäverna är början på sökordet.
- Men om `word[i]==word[j]` sätts i stället `next[i]=next[j]`.

Knuthautomaten kallas också KMP för att inte medförfattarna Morris och Pratt ska glömmas. Men det var Boyer och Moore som kom med den lysande

idén att tjuvtitta `n` steg framåt i texten. Så kan inte en automat göra, men om hela texten finns tillgänglig går det bra. Om vi söker efter ADAM i en lång text tjuvtittar vi på fjärde bokstaven, sedan på åttonde bokstaven och så vidare. Så länge dessa bokstäver inte är A,D eller M kan vi hoppa vidare (kom på varför!). Annars får automaten stega sej fram som förut.

Tjuvtitt och hopp gör KMP mycket snabbare för långa sökord. När man inte kan hoppa `n` steg framåt kan man faktiskt ändå hoppa en liten bit framåt, beroende på vilken bokstav i ordet man stött på. Detta görs i den algoritm som kallas *Boyer-Moore*, men nästan all förbättring sker redan om den här koden läggs in.

```
good = [False]*256
for tkn in word:                          # Bokför tecknen i sökordet
    good[ord(tkn)] = True
j = -1                                    # Aktuell position i bibeln
while n-i<len(bibeln)-j:                  # Om resten av ordet får plats
    if i==-1 and not good[ord(bibeln[j+n])]:
        j = j+n                           # Hoppa!
```

Rabin-Karps textsökningsalgoritm bygger på hashning. Först hashar man sökordet `ADAM`, sedan hashar man varje fyrbokstavsavsnitt ur bibeln och ser om något får samma hashvärde som sökordet. Innan man ropar hej måste man kolla att det inte är ett annat ord som bara råkar ha samma hashvärde.

Det här verkar bli orimligt mycket räknearbete, men det finns ett trick. Vi utgår från den hashfunktion vi tidigare använt.

```
def hash(word):    # Proffsig hashfunktion för N=1023
    h = 0
    for tkn in word:
        h = (h*155+ord(tkn))%1024
```

Först hashas `ADAM` till 111. Bibeltexten börjar `VADAN`, så vi ska räkna ut

$$\texttt{VADA} \Rightarrow 86 \cdot 155^3 + 65 \cdot 155^2 + 68 \cdot 155 + 65 = 321825480 = 712 \mod 1024$$

Den här uträkningen kan vi återanvända när vi hashar `ADAN`.

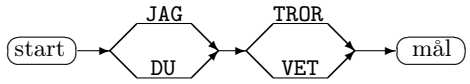
$$\texttt{ADAN} \Rightarrow 65{\cdot}155^3{+}68{\cdot}155^2{+}65{\cdot}155{+}78 = 155{\cdot}712{+}78{-}65{\cdot}155^4 = 112 \mod 1024$$

Man räknar ut 155^4 en gång för alla, så varje hashfunktion kräver bara två multiplikationer, oavsett hur långt sökordet är. Det blir ändå lite långsammare än KMP med tjuvtitt och hopp, men det finns en användning där

Rabin-Karp är överlägsen. Man kan leta efter många ord samtidigt genom att först hasha in dom i en hashtabell och sedan tar det inte längre tid än att leta efter ett enda ord. En typisk användning är att kolla hur mycket två uppsatser överlappar, en annan är att jämföra brottsplatsspår med DNA-registret.

12.2 Syntax

Svenska och Python är exempel på *språk*. Med det menar datalogen bara en uppsättning texter. En automat med startnod, målnod och textpilar definierar ett språk, nämligen de texter som bildas när man går längs pilarna från start till mål. Den här automaten definierar ett språk som består av fyra meningar: JAG TROR, JAG VET, DU TROR och DU VET.



En automat som programmeras efter denna graf kommer att gå från start till mål om den får någon av dessa fyra meningar som input. Men vad händer om den får meningen JAG ANAR? När ingen pil stämmer med inputten låter vi automaten avsluta körningen med felutskrift.

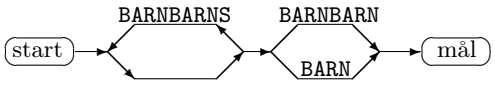
I stället för att rita en graf kan man definiera språket med en *syntax*. Detta grekiska ord uttalas på svenska *syntáx*, betyder grammatik och kan se ut så här.

```
<mening> ::= <subj><pred>
<subj>   ::= JAG | DU
<pred>   ::= TROR | VET
```

Uttydning: En mening är ett subj följt av ett pred, där ett subj är JAG eller DU och ett pred är TROR eller VET. Det här sättet att skriva en syntax kallas BNF, Backus-Naur-form, och uppfanns 1960 för att beskriva programspråket ALGOL. Vi teknologer fick ett tjockt häfte skrivet i BNF, sen var det bara att skriva algolprogram. Tyvärr fanns ingen kompilator men det gjorde inte så mycket eftersom det inte heller fanns någon dator.

Ett språk kan innehålla oändligt många ord. Så blir det om automaten har slingor eller om syntaxen är rekursiv. Svenska språket har till exempel oändligt många ord för ättlingar.

```
<ätt>::= <bbb><bb> | <bbb><b>
<b>  ::= BARN
<bb> ::= BARNBARN
<bbb>::=   | BARNBARNS<bbb>
```



Att barnbarnsslingan kan gås hur många varv man vill är lätt att förstå, men hur fungerar den rekursiva definitionen av `<bbb>`? Det står ingenting före eller-tecknet så `<bbb>` kan vara tom. Därav följer att den kan vara `BARNBARNS` och av det följer rekursivt att den kan vara `BARNBARNSBARNBARNS` osv.

Det första språk vi stötte på ska nu bli oändligt.

```
<mening> ::= <subj><pred> | <subj><pred><bisats>
<subj>   ::= JAG | DU
<pred>   ::= TROR | VET
<bisats> ::= ATT <mening>
```

JAG TROR ATT DU VET ATT JAG VET ATT DU TROR är en av dom oändligt många meningarna i detta språk.

Ofta är det rätt enkelt att skriva ett program som läser en text och kontrollerar om den följer syntaxen. Man brukar börja med att splitta texten i ord och lägga dom i en kö. Sedan definierar man för varje syntaxbegrepp en funktion som glufsar i sej det ur kön och protesterar om det var fel.

```
def readsubj:
    word = q.get()
    if word=="JAG": return
    if word=="DU": return
    print "fel subjekt:",word
def readpred:
    word = q.get()
    if word=="TROR": return
    if word=="VET": return
    print "fel predikat:",word
def readbisats:
    word = q.get()
    if word=="ATT": readmening()
    else: print "fel bisats:",word
def readmening:
    readsubj()
    readpred()
    if not q.isempty(): readbisats()
```

Felutskrifter i varje funktion är inte proffsigt. Det sköts bättre i huvudprogrammet så här.

```
def readsubj:
    word = q.get()
    if word=="JAG": return
    if word=="DU": return
    raise Exception("fel subjekt:"+word)
 - - -
try:
    readmening()        # Följer den syntaxen?
except Exception,fel:   # Annars hoppar vi hit
    print fel
```

Parsning är dataslang för algoritmer som läser text enligt en syntax. Det sätt vi har visat heter *rekursiv medåkning*. Det fungerar inte om man råkat skriva syntaxen vänsterrekursivt, exempelvis så här.

```
<zzz> ::=  | <zzz>Z
```

Noll eller flera `Z` alltså, men den här funktionen funkar inte.

```
def readzzz:
    if q.isempty(): return
    readzzz()              # Oändlig rekursion!
    tkn = q.get()
    if tkn=="Z": return
    raise Exception
```

Om syntaxen skrivits högerrekursivt hade det inte blivit något problem.

```
<zzz> ::=  | Z<zzz>
```

Eller-tecken i syntaxen fordrar att programmet kan avgöra vilket fall som är aktuellt. Ibland måste man då tjuvtitta ett eller flera tecken framåt. Syntaxen för barnbarnsbarnen är så illa skriven att man måste titta nio tecken framåt för att skilja mellan `BARNBARN` och `BARNBARNS`. Så här kan man slippa tjuvtitta.

```
<ätt>  ::= BARN<fler>
<fler> ::=   | BARN<flest>
<flest>::=   | S<ätt>
```

Ett språk som kan definieras av en automat kallas *reguljärt* och då finns alltid en syntax. Men vissa syntaxer har ingen automat, till exempel den här.

```
<aj> ::= AJ | A<aj>J
```

Rekursiv medåkning fungerar bra, men det går åt allt mer minne till rekursionen ju längre ord man ska kolla. En automat har inget annat minne än att den vet vilket tillstånd den befinner sej i. Eftersom ingen verklig dator har oändligt minne kan teoretiskt ingen dator parsa detta enkla språk!

Alan Turing tänkte sej en dator som kunde beställa fram mer minne när det behövdes. Den kallas nu turingmaskin och en sådan klarar alla språk definierade med BNF-syntax. Men på annat sätt kan man definiera språk där inte ens turingmaskinen kan avgöra om ett givet ord ingår i språket. Ett sådant problem kallas *oavgörbart*. Huruvida ett visst pythonprogram kommer att avslutas eller kommer att loopa oändligt är en känd oavgörbar fråga.

12.3 Hemma hos datorn: Kompilering

Varje programsats och varje kommandorad som ska förstås av datorn måste följa en given syntax. Då är det möjligt för ett översättningsprogram att tolka raden och göra om den till instruktioner som datorn kan utföra. Vad händer i ett unixsystem om användaren ger kommandot `python pnyxtr.py`? Först ska operativsystemets kommandotolk dela upp kommandoraden i två ord identifiera det första ordet som namnet på en körbar fil, starta pythontolken och skicka in det andra ordet som input. Om det första ordet varit felstavat hade kommandotolken gett felmeddelande.

Pythontolken läser igenom `pnyxtr.py`, delar upp den i ord, översätter tal till binära tal och kollar att raderna följer syntaxen. Om den inte hittar något fel laddas den inbyggda namnlistan in (`print` osv) och körningen kan börja. Sats för sats översätts till maskininstruktioner och utförs.

Program skrivna i C tolkas inte utan *kompileras*. Kompilatorn översätter hela programmet till en fil med maskininstruktioner som kan köras vid ett senare tillfälle. Processorns instruktioner är i binärkod och inte avsedda för mänskliga ögon men för datorn går det mycket fortare att köra en binärfil än att via pythontolken få instruktionerna sats för sats. En annan fördel med kompilerade filer är att dom står på egna ben och inte är beroende av att datorn har en pythontolk. I windowsmiljön är dom klickbara. En nackdel med kompilerade filer är att dom bara kan köras på den processor dom är kompilerade för. Olika processorer har nämligen olika instruktioner.

En värre nackdel ser man när man får meddelanden av typen **Fatal error!** Den kompilerade filen har inga programsatser och variabelnamn, så man får ingen hjälp att hitta felet.

Kompilering och interpretering (tolkning) använder sej ofta av rekursiv medåkning för att tolka syntaxen och samtidigt skapa motsvarande maskininstruktioner. Det finns så bra C-kompilatorer att det kan vara vettigt att först kompilera Python till C (det gör Pyrex) och det kan också kompileras till Java eller Javascript eller till Nokiatelefonkod. Att skriva en bra kompilator räknas som den största och roligaste uppgift en programmerare kan få.

Syntaxbegreppet är viktigt för mycket annat än programkod. Poster i databaser och informationspaket på nätet är exempel på det. Ofta används *protokoll* för kombinationen av syntax och semantik (betydelse). Ett protokoll talar inte bara om hur något ska se ut utan också hur det ska tolkas.

12.4 Varningar och råd

Automater kan övergå mellan olika tillstånd.

KMP-automat för textsökning har backvektorn `next`.

Boyer-Moore tjuvtittar, *Rabin-Karp* hashar!

Backus-Naur-form, BNF, beskriver syntax för ett språk rekursivt.

Högerrekursiv syntax tolkas av *rekursiv medåkning*.

12.5 Övningar

121. Gör en KMP-automat för ordet `ANANAS` och provkör den på texten `BANANANANASSALLAD`.

122. En DNA-molekyl är en lång text i alfabetet `CGAT`. Om man söker efter en viss tjugobokstavssekvens i en text med många tusen bokstäver, vilken metod är bäst och vilken är sämst?

123. I Monty Python serverades rätterna `SPAM` och `SPAM AND SPAM` och `SPAM, SPAM AND SPAM` osv. Skriv en syntax och ett rekursivt medåkande program för syntaxkoll.

124. Skriv en syntax för varupris av typen `17.50` och `4711.00`.

125. Skriv en syntax för aritmetiska uttryck av typen `12*((3+4)*5+(6*7)*8+9)`. Använd begreppen `<term>`, `<faktor>` och `<uttryck>` där ett parentesuttryck räknas som faktor.

126. Skriv ett program som kollar syntaxen ovan och sedan beräknar uttryckets värde.

Sakregister