Länkade listor, stackar och köer

I fortsättningen ska vi ta upp några olika *abstrakta datatyper*. De kan ses som enkla verktyg i en verktygslåda som ska göra det lättare att programmera. Några av dessa datatyper är *länkade listor, stackar och köer*. De här begreppen dyker upp lite varstans inom datalogi, programmering och operativsystem, så det kan löna sig att vara säker på grunderna.

En länkad lista (i sin enklaste version) består av *noder* som var och en "pekar" på nästa nod i listan. Varje nod har ett innehåll (som i boken kallas "cargo") och en referens till nästa nod i listan.

Vi kan börja med att skriva en klass som beskriver en nod i en länkad lista.

En nodklass

(Det här exemplet kommer ur boken.)

```
class Node:
        def __init__(self, cargo=None, nextNode=None):
                self.cargo=cargo
                self.nextNode=nextNode

        def __str__(self):
                return str(self.cargo)
```

Nod-objekt kan nu användas för att skapa listor.
Nedanstående kommandon skapar tre noder och kopplar ihop dem med varandra.

```
>>> from Node import *
>>> node1=Node(1)
>>> node2=Node(2)
>>> node3=Node(3)
>>> node1.nextNode=node2
>>> node2.nextNode=node3
```

Skriva ut lista och ta bort noder

Så här kan vi skriva ut en lista:

```
def printList(node):
      while node:
            print node,
            node=node.nextNode
      print
```

Man stegar sig alltså fram i listan via `nextNode`-pekarna. Den första noden är ett slags referens till hela listan; via den kan man nå alla andra noder genom att stega sig fram. Vi har sett hur man lägger till nya noder i en lista. Men hur tar man bort noder?

```
def removesecond(node):
      if list==None: return
      first=list
      second=list.nextNode
      # gör så att första noden pekar på den tredje
      first.nextNode=second.nextNode
      # låt andra noden peka ut i intet
      second.nextNode=None
      return second
```

Abstrakta datatyper: stackar

"Stack" betyder ju ungefär "hög" eller "trave" på svenska och den här datatypen tänker man sig ungefär som en trave med tallrikar, där den tallrik man senast lade dit är den första man plockar ut. Man säger att det är en LIFO (last in, first out)-struktur. Stackar brukar ha åtminstone två metoder, `push()` och `pop()`. `push()` lägger något (ett objekt, en sträng, ett tal...) på stacken. `pop()` plockar ut det som ligger överst - dvs. det som lades dit senast. Det kan också vara användbart att ha en metod `isEmpty()` för att kolla om stacken är tom och en metod för att skriva ut stackens innehåll.

I Python är det ganska lätt att skriva en stack-klass; det finns redan bra funktioner för att lägga till och ta bort objekt så man behöver inte göra så mycket själv. I boken (kap. 18) finns ett exempel på en stack-klass. Man kan också helt enkelt låta `Stack` vara en klass som ärver från `list`.

En stack-klass

```
import sys

class Stack(list):
        def push(self,x):
                self.append(x)

        # pop() finns redan för list i Python!

        def isEmpty(self):
                return len(self)==0

        def write(self):
                for i in self:
                      print i,
```

Funktionen `append()` lägger till ett objekt i slutet av en lista, så vi använder den som `push()` i vår stack-klass. `pop()` finns redan i Python, så den behöver vi inte ens skriva! Vi lägger till några metoder för att kolla om stacken är tom och för att skriva ut den.

Test av stack-klassen

Nu kan vi testa om stacken fungerar som den ska. När man plockar ut objekt man lagt in ska de komma ut i omvänd ordning. Vi testar med ett par tal och sedan ett helt ord.

```
from Stack import *
s=Stack()
s.push(3)
s.push(9)
s.push(13)
s.push(14)
print s.pop()
print s.pop()
# Vända på en sträng
t=Stack()
ord = "palindrom"
for a in ord:
        t.push(a)
while not t.isEmpty():
        sys.stdout.write ( t.pop() )
print
```

Utskriften blir:

```
14
13
mordnilap
```

Stack implementerad med länkad lista

Här är ett exempel på hur man kan göra en stack med länkade listor. Det antas att Node-klassen vi gjorde innan finns.

```
class Stack:
   top=None
   def __init__(self,num):
   # Här måste man lägga något i stacken då den skapas
   # Hur skulle man skriva annars?
      self.top=Node(num)
      self.length=1
   def push(self,num):
      self.top=Node(num,self.top)
      self.length+=1
   def pop(self):
      if self.length==0:
         print"Empty stack"
         return
      else:
         v=self.top.cargo
         self.top=self.top.nextNode
         self.length-=1
         return v
```

Köer

En kö är ungefär som en stack, men när man plockar ut saker ur den får man den första saken som stoppades in, inte den sista som i stackar. Precis som i en riktig kö alltså - den som ställde sig först kommer (förhoppningsvis) fram först. Köernas metoder för att lägga in och ta ut objekt heter ibland `enqueue()` och `dequeue()`, eller `get()` och `put()` - i boken har de valt namnen `insert()` och `remove()`. Boken (kap. 19) innehåller en kö som är gjord (dvs *implementerad*) med hjälp av länkade listor. Det kan vara lärorikt att titta på det exemplet och förstå hur det funkar. Men vi kan köra på samma modell som tidigare och låta vår kö ärva av `list`.

```
class Queue(list):
        def put(self,x):
                self.append(x)
        def get(self):
                return self.pop(0)
        def isEmpty(self):
                return len(self)==0
        def write(self):
                for i in self:
                      print i,
```

Den här klassen är nästan likadan som `Stack`, förutom att vi varit tvungna att skriva en liten metod för att plocka ut element.

Test av köklass

Vi testar köklassen för säkerhets skull:

```
from Queue import *
s=Queue()
s.put(3)
s.put(7)
s.put(13)
print s.get()
print s.get()
```

Resultatet blir som väntat:

```
3
7
```

Köklass med länkade listor

```
class Queue:
   front=None
   back=None
   def __init__(self,num):
   # Kräver att man lägger något i kön när den skapas
      self.front=self.back=Node(num)
      self.length=1
   def put(self,num):
      if self.length==0:
         self.front=self.back=Node(num)
         self.length=1
      else:
         n=Node(num)
         self.back.nextNode=n
         self.back=n
         self.length+=1
   def get(self):
      if self.length==0:
         print"Empty queue"
         return
      else:
         v=self.front.cargo
         self.front=self.front.nextNode
         self.length-=1
         return v
```

Att vända på en kö

Hur vänder man på en kö? Om man tar ut elementen ett efter ett och lägger dem i en annan kö, så hamnar de ju i samma ordning igen. Om vi vill vända på kön behöver vi en stack.

```
q=Queue()
s=Stack()
q.put(2)
q.put(5)
q.put(10)
print "Kön före vändning"
q.write()
while not q.isEmpty():
        s.push(q.get())
while not s.isEmpty():
        q.put(s.pop())
print "Kön efter vändning"
q.write()
```

Resultatet blir:

```
Kön före vändning
2 5 10
Kön efter vändning
10 5 2
```