Föreläsning 4: Abstrakta datastrukturer, kö, stack, lista

- Abstrakt stack
- Abstrakt kö
- Länkade listor
- Objekt och klasser

Abstrakta datatyper
Det är ofta praktiskt att beskriva vilka operationer man vill kunna göra på sina data utan att behöva tänka på hur operationerna ska implementeras (dvs hur programkoden ska se ut). En datastruktur som beskrivs på detta sätt kallas för en abstrakt datatyp.
En abstrakt datatyp

- Anger inte lagringssättet
- Specificerar operationer för åtkomst och ändring

Exempelvis har dictionary flera metoder vars implementation vi inte känner till.

```
>>> dict.values()
[12332, 231444, 53231]
>>> dict.keys()
['kalle', 'beata', 'cecilia']
```

Stack
En stack fungerar som en trave tallrikar - det man lägger överst på stacken är det som kommer att tas bort först. För en abstrakt stack finns följande operationer:

`push(x)`	Lägg x överst på stacken.
`x=pop()`	Hämta det som ligger överst på stacken.
`IsEmpty()`	Undersök om stacken är tom.

Den som använder datatypen behöver inte bry sig om hur den representeras. Funktionerna läggs i en klass som opererar på data. Så här skulle ett gränssnitt för en stack kunna se ut:

```
Stack:
    push()
    pop()
    isEmpty()
```

Skriv programmet **LadiesFirst.java** som läser en fil av typen med personnummer och namn och först skriver ut alla kvinnor och sedan alla män i filen.

120203-1114 Albus Dumbledore
330401-6402 Minerva McGonagall
920731-3131 Harry Potter

Männen måste tillfälligt läggas i ett förvaringsutrymme medan filen läses igenom. Man skulle kunna använda en vektor men vi ska i stället använda en *stack*. Man lägger en textrad på stacken med anropet `stack.push("En textrad")` och man hämtar en textrad från stacken med `raden=stack.pop()`. Med koden `if (stack.isEmpty()) ...` ska man kunna kolla om stacken är tom.

```
import sys
file_name = sys.argv[1]
try:
    infil = file(file_name)
except:
    print "No such file", file_name

stack = Stack()
for row in infil:
    if (int(row[9]) % 2 == 0):
        print row
    else:
        stack.push(row)

while not stack.isEmpty():
    print stack.pop()
print "done"
```

Här har vi programmerat abstrakt, som om push och pop vore fungerande metoder. Stackimplementationen kommer lite senare!

Kö
En kö fungerar som man förväntar sig, dvs det man stoppar in först är det som tas ut först. För en abstrakt kö finns följande operationer:

`enqueue(x)`	Stoppa in x sist i kön.
`X=dequeue()`	Hämta det som står först i kön.
`IsEmpty()`	Undersök om kön är tom.

En kö kan t ex användas för att hantera skrivarköer. Den som skrev ut först ska också få ut sin utskrift först. I labb 2 ska ni använda en kö för att förbereda en kortkonst!

Länkade listor
En länkad lista består av ett antal objekt, *noder* som är sammanlänkade genom att varje nod refererar till nästa nod. Dessa referenser kallas ofta *next-pekare*. Klassen `Node` skulle kunna användas för att skapa en länkad lista med heltal.

```
class Node:
    data = None
    next = None

    def __init__(self, d):
        self.data = d
        self.next = None
```

Klasser

Med klasser kan man kapsla in data och funktioner som hör ihop.

Metoden `__init __` är en specialmetod som kallas konstruktor. Det är den som anropas när man skapar en klassen t.ex. om man skriver

```
x = Nod(”lite innehåll”)
```

Funktioner i en klass anropas med genom att använda punkt. Exempel i lab 2 finns t.ex en `dictionary` som anropar medlemsfunktionen `has_key`

```
todo.has_key(’check’)
```

Funktionen has_key måste veta vilken dictionary den ska titta. Variablen `todo` måste skickas med i anropet. Hur görs det? Jo, alla medlemsfunktioner (metoder) i en klass tar en extra parameter `self`.

Stack
Här kommer nu klassen Stack, en stack implementerad med en länkad lista. Stacken kan också implementeras på andra sätt, t ex med pythons vektor.

```
class Stack:

    def __init__(self):
        self.top = None

    def pop(self):
        tmp = self.top
        self.top = tmp.next
        return tmp.data

    def push(self, data):
        ny_nod = Node(data)
        ny_nod.next = self.top
        self.top = ny_nod

    def isEmpty(self):
        return (self.top == None)

    def __str__(self):
        s = "--- Stacken ser ut så här ---\n"
        p = self.top
        while p != None:
            s += p.data + "\n"
            p = p.next
        s += "----------------------------"
        return s
```

Vissa medlemsfunktioner är speciella Python kan tolka specialmetoder i klasser. Dessa metoder är omgärdade med två understrykningstecken __ . Metoden `__str__` används t.ex. av funktionen print för att skriva ut objekt. Genom att definera __str__ i våran stackklass kan man nu skriva ut den med `print`.

Kö

En *kö* kan implementeras som länkad lista med samma noder men med en pekare även på sista noden. Där ska nämligen nya noder stoppas in.

```
class Queue:

    def __init__(self):
        self.first= None
        self.last = None

    def enqueue(self, data):
        ny_nod = Node(data)
        if (self.isEmpty()):
            self.first = ny_nod
            self.last = ny_nod
        else:
            self.last.next = ny_nod
            self.last = ny_nod

    def dequeue(self):                  # Ska skrivas i lab3

    def isEmpty(self):                  # Ska skrivas i lab3
```

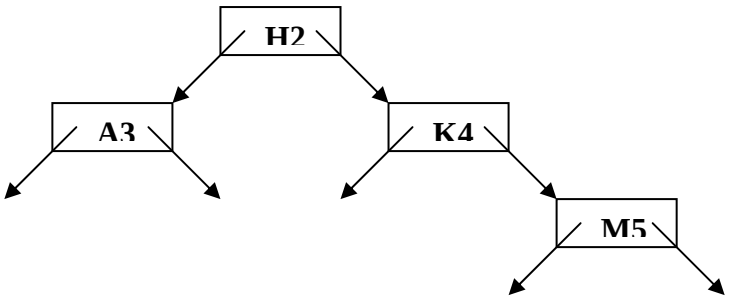
Träd

På samma sätt som man kan göra en enkellänkad lista med nodklasssen kan man göra lite mer komplicerade strukturer som träd med en tränodsklass. Dessa noder har två pekare, en höger- och en vänsterpekare.

```
class Treenode:

    def __init__(self, d):
        self.data = d
        self.left  = None
        self.right = None
```

Träd



Länkad lista