**Föreläsning 1: Introduktion till kursen**

**Algoritmer**

En beskrivning, i ett ändligt antal steg, av hur man löser ett givet problem.

**Exempel:**  
Skriv en algoritm för att läsa in gps-data för platser i Kina från en fil ([kina.txt](https://kth.instructure.com/courses/7489/files/1570153/download)) och lagra dom i en lista.   
Platserna är i filen ordnade med sydligast latitud först, men i listan vill vi att dom ska ligga i ordning från nordligaste till sydligaste.

Algoritm 1

1. öppna filen
2. läs in rubrikraden
3. läs in första raden
4. lägg raden sist i listan
5. upprepa punkt 3 och 4 tills alla rader i filen lästs in
6. vänd listan

|  |  |
| --- | --- |
| **Implementation med for-slinga** | **Implementation med while-slinga** |
| print("Algoritm 1 med append och reverse") filnamn = "kina.txt" rader = [] with open(filnamn, encoding = "utf-8") as landfil:  for rad in landfil:  rader.append(rad.strip()) rader.reverse() print("Antal rader", len(rader))  for i in range(5):  print(rader[i]) | print("Algoritm 1 med append och reverse + while") filnamn = "kina.txt" rader = [] with open(filnamn, encoding = "utf-8") as landfil:  rad = landfil.readline()  while rad:  rader.append(rad.strip())  rad = landfil.readline() rader.reverse() print("Antal rader", len(rader))  for i in range(5):  print(rader[i]) |

 Algoritm 2

Här är en annan algoritm för samma problem:

1. öppna filen
2. läs in rubrikraden
3. läs in första raden
4. lägg raden *först* i listan
5. upprepa punkt 3 och 4 tills alla rader i filen lästs in

|  |
| --- |
| **Implementation med insert** |
| print("Algoritm 2 med insert") filnamn = "kina.txt" rader = [] with open(filnamn, encoding = "utf-8") as landfil:  for rad in landfil:  rader.insert(0,rad.strip()) print("Antal rader", len(rader))  for i in range(5):  print(rader[i]) |

Ett extra exempel (pythonrepetition)

|  |  |
| --- | --- |
| **bok.py** | **bokdata.csv** |
| class Bok: def \_\_init\_\_(self, datarad):  datarad = datarad.strip()  datarad = datarad.split(",")  self.titel = datarad[0] def \_\_str\_\_(self):  return self.titel  with open("bokdata.csv", encoding = "utf8") as fil:  titelrad = fil.readline()  print(titelrad)  datarad = fil.readline()  b = Bok(datarad)  print(b) | Boktitel,sidantal,pris, typ De grå molnen,457,150,skönlitteratur |

**Föreläsning 2: Abstrakta datatyper**

Abstrakta datatyper

* Abstraktion
* Gränssnitt (Interface)
* Abstrakta datatyper
  + Kö (Queue)
  + Stack
  + Deque
* Länkade listor
  + Stackimplementation

Abstraktion

Abstraktion innebär att dölja detaljer. En användare behöver inte känna till detaljer för att kunna utnyttja en datastruktur eller algoritm och en konstruktör behöver inte veta var informationen kommer ifrån eller vad resultaten ska utnyttjas till.

Exempel:

Man behöver inte veta exakt hur data lagras för att använda operationerna.

Likadant i Python - vi vet inte hur strängar, listor och uppslagslistor är definierade, ändå kan vi använda dom. Det här är ett exempel på abstraktion. Om implementationen av listans metoder ändras (i en ny Python-version) behöver vi inte bekymra oss, alla våra program som använder listor fungerar ändå. Vi använder listan som en abstrakt datastruktur.

Några fördelar med abstraktion:

* Användaren och konstruktören kommer överens om vad som ska kunna göras och vilka data som ska utnyttjas (d v s ett gränssnitt).
* Sedan kan konstruktören ägna sig åt detta problem och användaren fortsätta som om det var löst (och utnyttja gränssnittet i sin del av programmet).
* Användaren kan inte missbruka detaljinformation (utan måste hålla sig till gränssnittet).
* Konstruktören kan förbättra konstruktionen utan att användarens kod behöver ändras.

Dessutom:

* Det är lättare att överblicka program om delproblem är lösta separat.
* Konstruktören kan anpassa den abstrakta datatypen/algoritmen till olika användare utan att förstöra en bra konstruktion.

**Gränssnitt (Interface)**

Begreppet *gränssnitt* används i många sammanhang, men det handlar alltid om någon form av "kontakt" eller "kommunikation". Man talar tex om gränssnittet mellan olja och vatten som skiktat sig i en behållare och om grafiska användargränssnitt som underlättar kommunikationen mellan användare och datorprogram.   
Här gäller det gränssnitt inuti program; hur en viss del av koden kommunicerar med resten av programmet.

**Abstrakta datatyper**

Heltal, flyttal, textsträngar och vektorer är datorns datatyper. Verklighetens datatyper är många fler, till exempel pengar, temperaturer och datum. Det är frestande att låta **pengar** representeras av heltal, **temperaturer** av flyttal (grader Celsius) och **datum** av textsträngar ("2016-08-31"), alltså av *konkreta datatyper*, men det är inte så bra. Datorn kan inte lagra hur stora heltal som helst, så när man kommer upp i stora belopp beter sig inte programmet som man tänkt sig. Temperatur anges i Fahrenheit i USA, så där räknar programmet fel. Och misstaget att ha datum som konkret datatyp kostade hundratals miljarder i omprogrammering vid tusenårsskiftet.

En ***abstrakt datatyp:***

* Anger inte lagringssättet
* Specificerar operationer för åtkomst och ändring

Så här skulle exemplen se ut med abstrakta datatyper.

saldo = kronor() # Ett objekt av den abstrakta datatypen kronor

saldo.set(2000) # Sätter nytt värde

saldo.plus(1500) # Ändrar värdet

print(saldo.get()) # Åtkomst av värdet

- - -

T = temperatur() # Ett objekt av den abstrakta datatypen temperatur

T.setC(11.5) # Sätter nytt värde i grader Celsius

print(T.getF()) # Åtkomst i grader Fahrenheit

- - -

d = datum() # Ett objekt av den abstrakta datatypen datum

d.set(2016,09,03) # Sätter ett värde

if d.helgdag(): # Användbara anrop finns

Specifikationen av vilka anrop som finns kallas datatypens gränssnitt och är det enda användaren behöver känna till. Hur data representeras konkret och hur metoderna implementerats behöver användaren inte veta. Om implementationen ändras påverkar det inte gränssnittet, så ingen användarkod behöver ändras.

Abstrakt 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(): Plocka ut och returnera det som står först i kön.
* isEmpty(): Undersök om kön är tom.

I labb 2 ska ni använda en kö för att förbereda en kortkonst. En kö kan t ex användas för att hantera skrivarköer (se [exempel  (Länkar till en externa sida.)Länkar till en externa sida.](http://interactivepython.org/courselib/static/pythonds/BasicDS/SimulationPrintingTasks.html)i Miller-Ranum). Den som först begärde utskrift ska också få ut sin utskrift först.

Abstrakt 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(): Plocka ut och returnera det som ligger överst.
* isEmpty(): Undersök om stacken är tom.

Abstrakt deque

Deque (double-ended queue)

För en abstrakt deque finns följande operationer:

* addFront(x): Lägg in x först.
* addRear(x): Lägg in x sist.
* x = removeFront(): Plocka ut och returnera det som ligger först.
* x = removeRear(): Plocka ut och returnera det som ligger sist.
* isEmpty(): Undersök om dequen är tom.

Ett tentatal

När influensavaccinet anländer till vårdcentralen bildas en lång kö utanför av folk som vill bli vaccinerade.

Syster Maud inser att vaccinet inte kommer att räcka till alla och ser att många i början av kön är unga och friska. Hon tror att dom som är i minst behov av vaccination hamnat först (dom sprang snabbast).  
Hon vill se till att personer som är 65 eller äldre vaccineras först. Hur ska hon göra?

Skriv programmet **MaudVaccinerar.py** som läser filen *patienter.txt* av typen

Usain 29

Annegret 65

Stanislawa 104

Frankie 47

Inge 74

och skriver ut alla under 65 först.

Yngre måste tillfälligt läggas i ett förvaringsutrymme medan filen läses igenom, till exempel i en stack. Man lägger ett objekt på stacken med anropet **push(p)** och man hämtar ett objekt från stacken med **p = pop()**.

Algoritm

1. Skapa en tom stack
2. Öppna filen
3. För varje patient i filen:
   * Om patienten är yngre än 65:
     + Pusha hen på stacken
   * annars:
     + Vaccinera
4. Nu är filen genomläst. Så länge stacken inte är tom:
   * Poppa ett element ur stacken
   * Vaccinera

|  |
| --- |
| from stack import Stack  class Patient:  def \_\_init\_\_(self, namn, ålder):  self.namn = namn  self.ålder = int(ålder)  def \_\_str\_\_(self):  return self.namn + " " + str(self.ålder) + " år"  def main():  korridor = Stack()  with open("patienter.txt", encoding = "utf8") as register:  for rad in register:  lista = rad.split()  namn = lista[0]  ålder = lista[1]  p = Patient(namn, ålder)  if p.ålder < 65:  korridor.push(p)  else:  print("Vaccinerar:", p)  print("Om vi hinner tar vi också: ")  while not korridor.isEmpty():  p = korridor.pop()  print("\t", p)  main() |

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

I vilken ordning kommer personerna ut?

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*.

class Node:

def \_\_init\_\_(self, x, next = None):

self.data = x # Kan referera till värde av valfri typ

self.next = next

När en nod skapas med n = Node(rad) har n.next värdet None, dvs pekar inte på någonting.

Vad innebär följande?

\* p.data  
\* p.next  
\* p = None  
\* p = p.next

Vi skapar en Stack-klass. Både Stack-klass och Nod-klass kan ligga i filen stack.py

För en stack behövs bara en referens **top** till den översta noden, sedan kommer man åt övriga noder genom att följa **next**-pekarna.

class Stack:

def \_\_init\_\_(self):

self.top = None

def push(self,x):

"""Lägger x överst på stacken """

ny = Node(x)

ny.next = self.top

self.top = ny

def pop(self):

"""Plockar ut och returnerar det översta elementet """

x = self.top.data

self.top = self.top.next

return x

def isEmpty(self):

"""Returnerar True om stacken är tom, False annars"""

if self.top == None:

return True

else:

return False

class Node:

def \_\_init\_\_(self, x, next = None):

self.data = x

self.next = next

En *kö* kan implementeras likadant som en stack. Nu vill man ha en pekare i var ände på kön.

Den som hette **top** i stacken kallar vi **first** och så har vi **last**som pekar på sista noden. Där ska nämligen nya noder stoppas in.

class Queue:

def \_\_init\_\_(self):

self.first = None

self.last = None

def enqueue(self,x):

"""Stoppar in x sist i kön """

ny = Node(x)

if self.first == None: # Om kön är tom blir det på ett sätt...

- - - # ...som du får tänka ut själv.

else: # Annars blir det på ett annat sätt..

- - - # ...som du också får lura ut själv.

def dequeue(self):

"""Plockar ut och returnerar det som står först i kön """

- - -

def isEmpty(self):

"""Returnerar True om kön är tom, False annars """

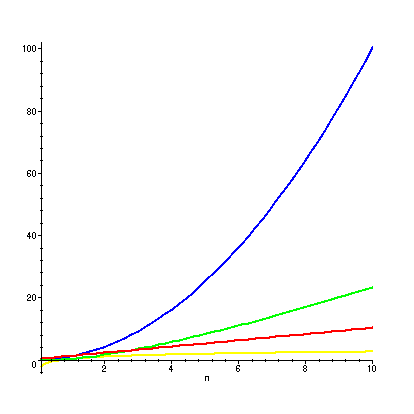
- - -

**Föreläsning 3: Komplexitetsanalys, sökning, rekursion**

* Komplexitetsanalys
* Linjärsökning
* Binärsökning
* Testning
* Rekursion  
  Denna föreläsning handlar om algoritmer. Vi börjar med att titta tillbaka på de två algoritmerna från [föreläsning 1](https://kth.instructure.com/courses/7489/pages/forelasning-1-introduktion-till-kursen).

Komplexitetsanalys

Det är intressant att se *tidsåtgången* för en algoritm. Denna anges ofta som funktion av indatas storlek, som av tradition kallas n. Exempel: För sortering är n antalet tal som ska sorteras.   
Hur växer tidsåtgången  T(n) för växande  n?



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n** | **log(n)** | **n\*log(n)** | **n2** | **2n** |
| 10 s | 1 s | 10 s | 100 s | 17 min |
| 100 s | 2 s | 3 min | 2 tim | 1022 år |
| 10000 s (ca 3 tim) | 4 s | 11 tim | 3 år |  |

Vi analyserar oftast *värsta fallet* (det är i regel enklare att räkna på) men man kan också räkna på *medelfallet*.

Istället för att ange den exakta tidsfunktionen  T(n) nöjer vi oss med att ange ordoklassen  O(n).

**Definition**

T(n) är O(F(n)) om det finns positiva konstanter c och n0 sådana att 0≤T(n)≤c⋅F(n)  för  n≥n0.

Vad innebär detta?

* c⋅F(n) anger en övre gräns för T(n) då n är stort.
* Vi behöver bara ta bara med den term som växer snabbast.
* Vi kan bortse från multiplikation med konstant.

Exempel:

T(n)=10n2 säger vi är  O(n2)

* För små indata är analys onödig - använd den enklaste algoritmen!
* Inläsning från fil tar längre tid än åtkomst av redan inlästa värden.
* Minnesåtgång kan också vara relevant.

Uppgift: Matrismultiplikation

Vad är tidskomplexiteten för matrismultiplikation?

**Sökning**

Förutsättningar:

* Vi söker efter något element i en lista med *n* element.
* Det element vi söker efter karakteriseras av en söknyckel (eng *key*) men kan också ha andra data.

Frågor:

* Vad ska hända om det sökta elementet inte finns med?
* Kan det finnas flera element med samma nyckel? Vill man i så fall hitta alla?

Linjärsökning (Sequential search)

Den enklaste sökningen: bryter så fort den hittar det sökta elementet.

Algoritm:

1. Gå igenom varje element i listan:
   * Jämför elementet med nyckeln, och
   * ...returnera True om dom är lika
2. Returnera False om hela listan gåtts igenom utan att elementet hittats.

Linjärsökning är O(n) (i värsta fallet måste vi titta på alla n elementen).

Här följer en funktion för linjärsökning i en lista.

|  |
| --- |
| def exists(the\_list, key):  for x in the\_list:  if x == key:  return True  return False |

Uppgift: Förbättra funktionen

Den här funktionen talar bara om huruvida elementet finns med i listan.  Hur ska den modifieras för att returnera den plats i listan elementet finns?

Binärsökning

Om listan är sorterad är den snabbaste sökningsalgoritmen binärsökning. Algoritm:

1. Beräkna intervallets mittpunkt.
2. Är det nyckeln? **Avbryt**, det sökta elementet är funnet.
3. Annars: Avgör om det nyckeln finns i första eller andra halvan och fortsätt söka där (upprepa från punkt 1)
4. Om halvan du söker i krympt till ingenting: **Avbryt**. Det sökta elementet fanns inte med.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 13 | 13 | 20 | 22 | 25 | 28 | 30 | 31 | 32 | 32 | 48 | 62 |
| **↓** | | | | | | | | | | | | |
| 11 | 13 | 13 | 20 | 22 | 25 |  |  |  |  |  |  |  |
| **↓** | | | | | | | | | | | | |
|  |  |  | 20 | 22 | 25 |  |  |  |  |  |  |  |
| **↓** | | | | | | | | | | | | |
|  |  |  |  |  | 25 |  |  |  |  |  |  |  |

Binärsökning är O(log2⁡n) i värsta fallet. Varför det?

Vi söker bland n element och undrar hur många varv sökningen kan ta?

Antal element att söka bland halveras i varje varv, så första varvet får vi n2, andra varvet n4, tredje varvet n8. Vi är klara när det bara finns ett element kvar att titta på och då är n2x=1 där x är antal varv. Vi två-logaritmerar bägge led och får att x=log2⁡n.   
Här följer en funktion för binärsökning:

|  |
| --- |
| def binary\_search(the\_list, key):  low = 0  high = len(the\_list)-1  found = False   while low <= high and not found:  middle = (low + high)//2  if the\_list[middle] == key:  found = True  else:  if key < the\_list[middle]:  high = middle - 1  else:  low = middle + 1  return found |

Binärsökning är en knepig algoritm att implementera korrekt. Hur testar vi att koden ovan fungerar?

Testning

Här är några förslag! Testa:

1. normalfallet, t ex att söka efter 14 i listan [9, 14, 23, 54, 92, 104]
2. att söka efter ett tal som *inte* finns med, t ex 15
3. att söka i en tom lista []
4. att söka efter vänstraste elementet 9 ...
5. ... och högraste elementet 104.
6. att söka efter ett element bortom vänstra gränsen, t ex 8 ...
7. ... och bortom högra gränsen, t ex 105.

Rekursion

Rekursiv kommer från latinet och betyder *återlöpande*. Om man i definitionen av ett begrepp använder begreppet självt så är definitionen rekursiv. Rekursiva tankar kan också användas för problemlösning.

* *Rekursiv tanke*:   reducerar problemet till ett enklare problem med samma struktur
* *Basfall*:  ett fall som inte leder till rekursivt anrop

Sifferexempel

Triangeltalet S(N) är summan av de N första heltalen. S(4)=1+2+3+4   
**Fråga:** Vad är värdet på S(N)?  
**Rekursivt svar:** S(N) = S(N-1) + N ... men S(1)=1.   
Här följer en rekursiv funktion för beräkning av triangeltalet:

def S(n):

if n == 1:

return 1

else:

return S(n-1) + n

**Fråga:** Vilken siffersumma har heltalet n?

**Rekursivt svar:** Sista siffran plus siffersumman om man stryker sista siffran i n, ...men noll har siffersumman noll.

def siffersumma(n):

if n == 0:

return   
 else:

return n%10 + siffersumma(n//10)

**Fråga:** Hur många siffror har heltalet n?   
**Rekursivt svar:** En siffra mer än om man stryker sista siffran i n, ...men tal mindre än tio är ensiffriga.

def antalsiffror(n)

if n...

Hur fungerar det?

När man skriver egna rekursiva funktioner bör man lita på att det rekursiva anropet fungerar - man behöver inte analysera anropsgången för varje fall. Men för att förstå varför rekursion kan vara extra minneskrävande är det vara bra att känna till hur programspråken hanterar rekursiva anrop.

* För varje anrop skapas en *aktiveringspost* som innehåller data för anropet, t ex parametrar, lokala variabler och anropspunkt.
* Aktiveringsposten pushas på en stack.
* När det rekursiva anropet är klart poppas aktiveringsposten från stacken, varefter föregående anrop ligger överst på stacken.

|  |
| --- |
| s(1) |
| s(2) |
| s(3) |
| s(4) |
| huvudprogram |

Rekursiv binärsökning

Binärsökning är lätt att göra rekursivt! Basfallet är att listan är tom, dvs har noll element.

Rekursiv binärsökningsfunktion:

def binsok(listan, nyckel):

if len(listan) == 0:

return False

else:

mitten = len(listan)//2

if listan[mitten] == nyckel:

return True

else:

if nyckel < listan[mitten]:

return binsok(listan[:mitten], nyckel)

else:

return binsok(listan[mitten+1:], nyckel)

Några saker att minnas

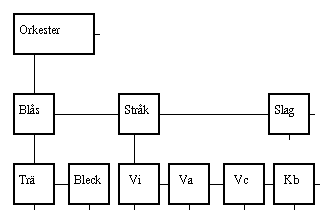
* + En rekursiv funktion kan alltid omformuleras utan rekursion, men om det finns flera rekursiva anrop i funktionen kan det vara besvärligt.
* För vissa problem är en rekursiv funktion mycket enklare att formulera och ger kortare kod än utan rekursion. Ofta måste man gå via den rekursiva lösningen i tanken även om man gör en icke-rekursiv lösning.

**Föreläsning 4: Binära träd**

* Allmänna träd
* Binära sökträd
* Rekursiva tankar för binärträd

Allmänna träd

*Stack* och *kö* är två viktiga datastrukturer man kan bygga av objekt, där varje objekt refererar till **ett** annat objekt.



Med **två** referenser i varje objekt kan man bygga träd, till exempel ett som beskriver en symfoniorkesters sammansättning.   
Här har objekten följande utseende.

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.down = None

self.right = None

All systematisk uppdelning kan beskrivas med liknande träd, till exempel ett bokverks uppdelning i delar, kapitel, sektioner osv. Man kan också tänka sej det som ett släktträd och då kallas ofta *down*-referensen för *firstChild* och *right*-referensen för *nextSibling*. Det räcker med två referenser i varje nod, oavsett hur stora barnaskarorna är.

Användningsområden

Trädstrukturer är hierarkiska och sådana datastrukturer är mycket vanliga. Några exempel:

* Filsystemet använder träd (man kan ha underkataloger i underkataloger).
* Databaser använder träd för att få snabb sökning.
* Schackprogram använder träd för att gå igenom resultaten av möjliga drag.
* Vid datakomprimering kan man använda träd för att få fram en optimal kod (Huffmanträd, kommer på komprimeringsföreläsningen).

Definitioner

* **Noder** är de objekt som trädet är uppbyggt av. De innehåller data och pekare.
* **Rot** är den översta noden i trädet. Den pekas inte ut av någon annan nod.
* **Barn** till en nod är de som pekas ut av noden.
* **Förälder** är noden ovanför i trädet.
* **Syskon** har samma förälder.
* **Löv** är en nod vars bägge pekare är None.
* **Delträd** definieras så här: En godtycklig nod i trädet kan ses som en rot, och den , tillsammans med alla noder under den (barn, barnbarn osv.) bildar ett delträd.
* **Nivå** är det antal steg från roten noden befinner sig. Roten är på nivå noll.
* **Höjd** är den maximala nivån som nån av trädets noder befinner sig på.
* **Balanserat** är binärträdet om skillnaden i höjd mellan höger och vänster delträd till varje nod är noll eller ett.
* **Fullt** är binärträdet om alla noder utom löven har exakt två barn, och alla löv är på samma nivå.

**Binärträd**

När man programmerar binärträd brukar man använda noder, som i en länkad lista, men med två pekare: en åt vänster och en åt höger:

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

Man når trädet genom variabeln *root* som pekar på den översta noden (datalogiska träd har roten uppåt). Rotnodens vänsterpekare pekar på ett mindre binärträd och högerpekaren på ett annat mindre binärträd.

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

**Binära sökträd**

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

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

Algoritmer som går igenom varje nod i trädet (t ex utskrift) har tidskomplexitet O(n). Men sökningen tar bara O(logn) om trädet är balanserat, därför att vi som mest går igenom trädets höjd.

def finns(p,value):

letar = True

while letar:

if p == None:

return False

if value == p.value:

return True

if value < p.value:

p = p.left

if value > p.value:

p = p.right

Rekursivt listexempel

Vi tänker oss en länkad lista av noder, där varje nod innehåller ett värde och en *next*-pekare.  
**Fråga:** Hur många noder finns i listan?   
**Rekursivt svar:** En nod mer än i listan under översta noden.   
**Basfall:** En tom lista har noll noder.

def antal(p):

if p == None:

return 0

else:

return 1 + antal(p.next)

Anropet *antal(p)* ger nu rätt svar!

Rekursiva tankar för binärträd

**Fråga:** Hur många noder finns i binärträdet?   
**Rekursivt svar:** En nod mer än i vänsterträdet och högerträdet tillsammans  
**Basfall:** Ett tomt träd har noll noder.

def antal(p):

if p == None:

return 0

else:

return 1 + antal(p.left) + antal(p.right)

Anropet *antal(root)* ger nu rätt svar!

Sökning i ett binärt sökträd kan implementeras rekursivt om man låter anropet *finns(p,value)* returnera *True*ifall ordet finns i det delträd där *p* är rot.

def finns(p,value):

if p == None:

return False

if value == p.value:

return True

if value < p.value:

return finns(p.left,value)

if value > p.value:

return finns(p.right,value)

Den här funktionen kan du använda i labb 3!   
Där ska du göra en klass som fungerar som ett *abstrakt* binärt sökträd med operationer för att stoppa in ett element, söka efter ett värde och skriva ut alla värden.

Utskrift av binärträd: inorder, preorder, postorder

Om man ska skriva ut alla talen i trädet vill man oftast göra det i så kallad *inordning* (eng. inorder), dvs från vänster till höger.   
**Fråga:** Hur skriver man ut trädet i inordning?   
**Rekursivt svar:** Först skriver man ut vänsterträdet, sedan rottalet, sist högerträdet.  
**Basfall:** Ett tomt träd skriver man inte ut.

Följande funktion gör att *write(root)* skriver ut 1 17 666 4711 9999 för vårt träd.

#Inordning

def inorder(p):

if p != None:

inorder(p.left)

print(p.value)

inorder(p.right)

Om man kastar om dom tre sista satserna får man ändå ut alla talen på skärmen men i andra ordningar. *Preordning* (eng. preorder) innebär att rottalet skrivs först, sedan vänsterträdet och sist högerträdet. I vårt exempel blir ordningen 4711 17 1 666 9999.

Om vi återgår till orkesterträdet kan vi se att preordning faktiskt ger vettigare utskrift. Så här blir koden i det fallet.

#Preordning

def preorder(p):

if p != None:

print(p.value)

preorder(p.down)

preorder(p.right)

Utskriften blir då den naturliga. Om vi för tydlighets skull använder indragning av orden på lägre nivå blir utskriften

Orkester

Blås

Trä

Bleck

Stråk

Vi

Va

Vc

Kb

Slag

(Hur gör man för att få dessa indragningar?)

Slutligen kan man skriva ut i *postordning* (eng. postorder) och det innebär att vänsterträdet skrivs först, sedan högerträdet och sist roten. Det ger 1 666 17 9999 4711 i vårt exempel.

Föreläsning 5: Hashning, bloomfilter

* Pythons dictionary
* Idén med hashning
* Komplexiteten för sökning
* Dimensionering av hashtabellen
* Hashfunktionen
* Krockhantering
* Klassen Hashtable
* Användningsaspekter
* Hashning i Språkteknologi

Pythons dictionary

Med Pythons inbyggda *dictionary* har man möjlighet att skapa en uppslagslista. Man bygger upp den genom att lägga in nycklar och tillhörande värden:

telefonnummer={}

telefonnummer["Linda Kann"] = "08-7909276"

telefonnummer["CSC-service"] = "08-790 7146"

---

Sedan kan man slå upp i listan:

namn = input("Vem vill du ringa till? ")

try:

print("Telefonnumret är ", telefonnummer[namn])

except KeyError:

print(namn,"finns inte med i telefonlistan")

Här är det varken linjärsökning eller binärsökning som används för att hitta nyckeln, utan en ännu snabbare sökmetod: *hashning*.

Idén med hashning

Binärsökning i en sorterad lista går visserligen snabbt, men sökning i en hashtabell är oöverträffat snabbt. Och ändå är tabellen helt oordnad (hash betyder ju hackmat, röra). Låt oss säga att vi söker efter Lyckan i en hashtabell av längd 10000. Då räknar vi först fram *hashfunktionen* för ordet Lyckan och det ger detta resultat.

hash("Lyckan") -> 1076540772

Hashvärdets rest vid division med 10000 beräknas nu

1076540772 % 10000 -> 772

och när vi kollar hashtabellens index 772 hittar vi Lyckan just där!

Hur kan detta vara möjligt? Ja, det är inte så konstigt egentligen. När Lyckan skulle läggas in i hashtabellen gjordes samma beräkning och det är därför hon lagts in just på 772. Hur hashfunktionen räknar fram sitt tal spelar just ingen roll. Huvudsaken är att det går fort, så att inte den tid man vinner på inbesparade jämförelser äts upp av beräkningstiden för hashfunktionen.

Komplexiteten för sökning

Linjär sökning i en oordnad lista av längd N tar i genomsnitt N/2 jämförelser, binär sökning i en sorterad lista log N men hashning går direkt på målet och kräver bara drygt en jämförelse. Varför drygt? Det beror på att det är svårt att undvika *krockar*, där två olika namn hamnar på samma index.

Dimensionering av hashtabellen

Ju större hashtabell man har, desto mindre blir risken för krockar. En tumregel är att man bör ha minst femtio procents luft i tabellen, dvs att

λ=antalhashtabellens

 Då kommer krockarna att bli få.

Hashfunktioner

Oftast gäller det först att räkna om en sträng till ett stort tal. Funktionen *ord(tkn)* i Python konverterar ett tecken till motsvarande ordningsnummer.

T ex är   
ord("A") = 65   
ord("B") = 66   
ord("C") = 67   
Då kan vi räkna om strängen "ABC" till talet 656667, genom att multiplicera den första bokstaven med 10000, den andra med 100, den tredje med 1 och slutligen addera talen. På liknande sätt gör metoden hash(key) i Python men den använder 32 i stället för 100. För en binär dator är det nämligen mycket enklare att multiplicera med 32 än med 100. Här är en förenklad variant:

def hash2(s): # Beräknar hashkoden för en sträng enligt

result = 0 # s[0]\*32^[n-1] + s[1]\*32^[n-2] + ... + s[n-1]

for c in s:

result = result\*32 + ord(c)

return result

Om nyckeln är ett datum eller personnummer behöver vi inte konvertera till tal.

* Man kan "vika" talet genom att dela upp det i lika stora delar som sedan summeras, t ex **20+17+09+11 = 57**,
* Eller kvadrera det:**20170911² = 406865**650**569921** och plocka ut de mittersta siffrorna: **650**.

En hashfunktion bör ha god spridning - vi vill inte att många nycklar ska ge samma hashvärde. Med programmet [barchart.pyFörhandsvisa dokumentet](https://kth.instructure.com/courses/7489/files/1570135/download) med tillhörande datafil [slumpnamn30.txtFörhandsvisa dokumentet](https://kth.instructure.com/courses/7489/files/1570150/download) kan du experimentera med fördelningen för olika hashfunktioner.

Modulo

I kursen använder vi modulo för att få hashfunktionens tal att passa in i vektorn.

Exempel: Datum kan hashas in i hashtabellen med **20170911 % size**

Krockhantering

En idé är att lägga alla namn som hashar till ett visst index som en länkad *krocklista*. Om man har femtio procents luft i sin hashtabell blir krocklistorna i regel mycket korta. Krocklistorna behandlas enklast som stackar, och hashtabellen innehåller då bara topp-pekarna till stackarna.

Linjär probning

En annan metod är att vid krock lägga noden på *första lediga plats*. Är det tomt där, tittar man på nästa, osv. Detta kallas *linjär probning*. En fördel med denna metod är att man slipper alla pekare. En stor nackdel är att om det börjat klumpa ihop sej någonstans har klumpen en benägenhet att växa. Detta kallas för *klustring*.

Kvadratisk probning

I stället för att leta lediga platser som ligger tätt ihop kan man därför göra större hopp. Hopplängden bör då variera. Ett sätt är att "hoppa fram" i jämna kvadrater, så kallad *kvadratisk probning*. Om hashfunktionen gav värdet *h* tittar man i ordning på platserna: h+1, h+4, h+9, ... . Överstiger värdena hashtabellens storlek använder man resten vid heltalsdivision precis som vid beräkningen av h. Om tabellstorleken är ett *primtal*. och tabellen är som mest halvfull, så riskerar man inte att fastna i en evig hopprunda.

Dubbelhashning

Ytterligare ett sätt att lösa krockproblemet är *dubbelhashning*. I denna variant räknas nästa värde fram med en annan hashfunktion som tar som indata den första hashfunktionens värde. För att hitta efterföljande platser låter man den andra hashfunktionen få sitt förra värde som indata.

Både kvadratisk probning och dubbelhashning ger goda prestanda om hashtabellen har femtio procent luft. En nackdel med båda metoderna är att man inte enkelt kan ta bort noder utan att förstöra hela systemet.

Klassen Hashtabell

I en senare laboration ska du implementera den abstrakta datastrukturen hashtabell genom att skriva en klass Hashtabell med operationerna **put** och **get**. Första parametern till put är söknyckeln, till exempel personens namn. Andra parametern är ett objekt med alla tänkbara data om personen. Metoden get har söknyckeln som indata och returnerar dataobjektet om nyckeln finns i hashtabellen, annars skickar vi ett särfall.

from hashtabell import Hashtabell, FannsInte

table = Hashtabell(7)

table.put("one",1)

table.put("two",2)

table.put("three",3)

kontrollord = "xxx"

while kontrollord != "":

kontrollord = input("Ett engelskt räkneord:")

try:

print(table.get(kontrollord))

except FannsInte:

print(kontrollord, "fanns inte i hashtabellen")

print("Försök igen!")

Hashtabellen ska åtminstone ha följande operationer:

**put(key, data)** Lägg in data med nyckeln key i hashtabellen.   
**data = get(key)** Hämta data som hör till key.   
**f = hashfunction(key)** Beräkna hashfunktionen för key.

Men man kan lägga till fler operationer, t ex **\_\_str\_\_()** för att skriva ut hashtabellen, **\_\_contains\_\_(key)** för att se om något finns lagrat med nyckeln key, **\_\_len\_\_()**för att få ut hashtabellens storlek mm.

Användningsaspekter

I nästan alla sammanhang där snabb sökning krävs är det hashtabeller som används. Krockar hanteras bäst med länkade listor, men i vissa programspråk är det svårt att spara länkade strukturer på fil, så därför är dubbelhashning fortfarande mycket använt i stora databaser.

I Ubuntu och andra UNIX-system skriver användaren namn på kommandon, program och filer och räknar med att datorn snabbt ska hitta dom. Vid inloggning byggs därför en hashtabell upp med alla sådana ord. Men under sessionens förlopp kan många nya ord tillkomma och dom läggs bara i en lista som söks linjärt. Så småningom kan det bli ganska långsamt, och då är det värt att ge kommandot *rehash*. Då tillverkas en ny större hashtabell där alla gamla och nya ord hashas in. Hur stor tabellen är för tillfället ger kommandot *hashstat* besked om.

Om man vill kunna söka dels på *namn*, dels på *personnummer* kan man ha en hashtabell för varje sökbegrepp, men det går också att ha en enda tabell. En viss person hashas då in med flera nycklar, men själva informationsnoden finns alltid bara i ett exemplar. Många noder i hashtabellen kan ju peka ut samma nod.

Hashning i Språkteknologi

Hashning används i många olika sammanhang. Här betraktar vi ett exempel från ämnesområdet *Språkteknologi*, dvs behandling av naturliga språk (mänskliga språk, till skillnad från tex programmeringsspråk) med datorer.

Stavningskontroll

Ett stavningskontrollprogram ska läsa en text och markera alla ord som är felstavade. Om man har tillgång till en ordlista som innehåller alla riktiga svenska ord kan man använda följande enkla algoritm för att stavningskontrollera en text.

* Läs in ordlistan i en lämplig datastruktur.
* Öppna textfilen.
* Så länge filslut inte nåtts:
  + Läs in nästa ord från filen.
  + Slå upp ordet i ordlistan och skriv ut det på skärmen om det inte finns med.

Enda problemet är hur man ska välja datastruktur för lagring av ordlistan. Svenska akademiens ordlista innehåller ungefär 200000 ord. Förutom dessa ord finns en hel del böjningsformer och oändligt många tänkbara sammansättningar. Låt oss bortse från detta och anta att vi har köpt en ordlista med dom 200000 vanligaste orden i svenskan. Om vi snabbt ska kunna stavningskontrollera en stor text med en normal persondator måste följande krav på datastrukturen vara uppfyllda.

* Uppslagning måste gå jättesnabbt.
* Datastrukturen får inte ta så mycket minne (helst inte ens så mycket minne som orden i klartext).
* Orden måste vara kodade (eftersom ordlistan är köpt och inte får spridas till andra).
* Vi kan tillåta att uppslagningen gör fel någon gång ibland.

Den sista punkten är inte ett krav utan en egenskap hos vårt problem som vi kan utnyttja. Det är nämligen inte hela världen om programmet missar något enstaka felstavat ord i en jättestor text.

Vanliga datastrukturer (sorterad array, sökträd, hashtabell) faller alla på något av kraven ovan.

Försök med datastruktur: boolesk hashtabell

Låt oss först försöka med hashning där vi inte lagrar själva orden och inte tar hand om eventuella krockar. Vi har en hashfunktion f(ord)=index som för varje ord anger en position i en boolesk hashtabell tab. Den booleska variabeln tab[f(ord)] låter vi vara sann då ord ingår i ordlistan. Detta ger en snabb, minnessnål och kodad datastruktur, men den har en stor brist: Om det råkar bli så att hashfunktionen antar samma värde för ett ord i ordlistan som för ett felstavat ord så kommer det felstavade ordet att godkännas. Om hashtabellen exempelvis är fylld till häften med ettor så är sannolikheten för att ett felstavat ord ska godkännas ungefär 50% vilket är alldeles för mycket.

Bloomfilter

Lösningen är att använda många hashfunktioner som alla ger index i samma hashtabell tab. I Viggos stavningskontrollprogram Stava används till exempel 14 olika hashfunktioner f0(ord),f1(ord), f2(ord),...,f13(ord). Ett ord godkänns bara om alla dessa 14 hashfunktioner samtidigt ger index till platser itab som innehåller sant.

Uppslagning av ett ord kan då ske på följande sätt:

for i in range(14):

if not tab[f(i, ord)]: return False

return True

Om hashtabellen är till hälften fylld med ettor blir sannolikheten för att ett felstavat ord godkänns så liten som (1/2)14=0.006%.

Denna datastruktur kallas *bloomfilter* efter en datalogiforskare vid namn Bloom. Ett abstrakt bloomfilter har bara två operationer: insert(x) som stoppar in x i datastrukturen och isIn(x) som kollar ifall x finns med i datastrukturen.

**Föreläsning 6: Grafer**

* Grafer
* Problemträd
* Breddenförstsökning
* Djupetförstsökning

Grafer

* En allmän graf består av en samling hörn (*vertices*) med kanter (*edges*) emellan.
* Grafen är riktad om kanterna är riktade (pilar).
* Antalet kanter från ett hörn är hörnets grad (*degree*).
* En stig (*path*) är en väg från ett hörn till ett annat, dvs en följd av intilliggande kanter.
* En cykel är en stig där man startar och slutar i ett och samma hörn.
* Om grafen inte har några cykler är den acyklisk.
* I en viktad graf har kanterna vikter.

Abstrakt datatyp för grafer

Följande metoder skulle en graf kunna ha:

**addVertex(node)** Lägger till hörnet *node* till grafen.   
**addEdge(node1, node2)**Drar en kant mellan hörnen node1 och node2.   
**v = getVertex(key)** Hittar det hörn som har nyckeln *key*   
**vlist = getVertices()** Returnerar en lista med alla hörn.

En graf kan implementeras med en *grannmatris* (adjacency matrix) eller *grannlista* (adjacency list).

En grannmatris har ett element för varje möjlig kant mellan två hörn. Elementet kan vara True/False (för en oviktad graf) eller ha ett värde som anger kantens vikt (för en viktad graf).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| A |  | 5 |  | 1 |
| B | 5 |  |  |  |
| C |  |  |  | 2 |
| D | 1 |  | 2 |  |

En gles grannmatris ger ineffektiv lagring, så för grafer med relativt få kanter är det bättre att använda en grannlista. Grannlistan har ett element för varje hörn, och i det lagras en lista (eller vektor) med kanterna.

|  |  |
| --- | --- |
| A | [B, 5], [D, 1] |
| B | [A, 5] |
| C | [D, 2] |
| D | [A, 1], [C, 2] |

**Problemträd**

En mycket stor klass av praktiska problem kan beskrivas med *problemträd* och lösas med trädgenomgång: bredden först eller djupet först. Ett problemträd är en riktad, acyklisk graf.

Problemträd uppkommer ständigt i praktiken. Man brukar kalla startnoden för *urmoder* eller *stamfar* och noderna under den för *barn*.

**Breddenförstsökning**

Nästa laboration går ut på att finna kortaste vägen från **söt** till **sur** genom att byta en bokstav i taget och bara använda ord i ordlistan, till exempel så här:   
söt -> söm -> döm -> dum -> dur -> sur

Problemträdets stamfar **söt** har barnen göt, löt, nöt, röt osv och barnbarnen get, gök, göl, lat, lön osv. Enligt kedjan ovan är **sur** barnbarnbarnbarnbarn till **söt**, men sur finns säkert redan tidigare i problemträdet. För att finna den första förekomsten gör man en breddenförstsökning enligt följande.

Lägg stamfadern som första och enda objekt i en kö. Gör sedan följande om och om igen: Plocka ut den första ur kön, skapa alla barn till denne och lägg in dom sist i kön. Första förekomsten av det sökta ordet ger kortaste lösningen.

Breddenförstsökningsalgoritmen kan sammanfattas så här.

1. Lägg stamfadern i kön.
2. Ta ut det första objektet ur kön.
3. Skapa alla dess barn och lägg in dom i kön.
4. Om någon av barnen är lösningen så är vi klara. Annars - upprepa från punkt 2 tills kön blir tom.
5. När lösningen hittas, följ förälderpekarna och skriv ut kedjan.

Man kan spara in både tid och utrymme om man undviker att skapa barn som är kopior av tidigare släktingar (t ex **nöt**s barn **söt**), I den här kursen kallar vi dem *dumbarn*. I algoritmsammanfattningen ovan kan vi hamna i en oändlig loop om vi inte tar hänsyn till dumbarnen.

Om man bara lägger själva orden i kön finns det ingen möjlighet att i efterhand tala om vägen från söt till sur. Därför bör man för varje nytt ord skapa en liten nod som innehåller ordet och en referens till föräldern och det man lagrar i kön är denna nod.

Breddenförstsökning ger alltid den *kortaste* lösningen. Ofta är det den man är ute efter. Några andra problemexempel är följande.

**Flygresa från Stockholm till Windhoek**

Stockholm är urmoder, destinationer med [direktflyg från Stockholm (Länkar till en externa sida.)Länkar till en externa sida.](http://worldairlinenews.files.wordpress.com/2013/06/scandinavian-sas-arn-62013-route-map.jpg) blir söner och så vidare. Dumbarn är destinationer man redan "passerat". Breddenförstsökning ger en resa med så få mellanlandningar som möjligt.

Totalsökning, brute force

Att det är minst antal mellanlandningar behöver inte betyda att det är den resrutt som tar kortast tid. För att räkna ut den snabbaste resrutten kan man göra en totalsökning och gå igenom alla kombinationer av resrutter. Att systematiskt pröva alla tänkbara lösningar brukar kallas *brute force.* Detta tar exponentiell tid O(kn), vilket blir orimligt långsamt för stora n.

Dynamisk programmering

Om alla resrutter är enkelriktade mot resmålet d.v.s inga returresor finns med så är grafen riktad. Då kan man använda sig av dynamisk programmering som innebär att man sparar undan och utnyttjar tidigare uträknade värden när man ska beräkna nästa värde.

Lönsam valutaväxling

Finns det någon lönsam växlingskedja av typen **1.00 SEK -> 0.11 EURO -> 0.13 USD -> ... -> 1.02 SEK** ? Vi vill ha en algoritm som kan besvara den frågan.

Vi antar att alla växlingskurser är kända, t ex **1.00 SEK -> 0.14 USD** och **1.00 USD -> 7.05 SEK**. En **valutanod**är ett belopp i en viss valuta. Vi utgår från valutanoden **1.00 SEK** och låter den vara urmoder i ett problemträd. Urmoderns barn är alla valutanoder som kan åstadkommas med en växling, till exempel **0.14 USD** och **16.5 JPY**. Barnet **0.14 USD** har i sin tur barn, däribland **0.987 SEK**. Just den är ett så kallat dumbarn och kan lugnt glömmas bort, eftersom den är sämre än en tidigare valutanod.

Om man går igenom problemträdet nivå för nivå, dvs generation efter generation, kanske man till sist stöter på noden **1.05 SEK**. Därmed har man funnit en lönsam växlingskedja och det är bara att sätta igång och växla så fort som möjligt innan kurserna ändras. För att avbryta trädgenomgången och hals över huvud återvända till huvudprogrammet kan man generera ett *särfall* med raise Klar(meddelande) och se till att huvudprogrammet har

try:

- - - # Om särfallet uppstår här...

except Klar:

- - - # ...teleporteras man hit

Allra enklaste sättet att definiera Klar är:

class Klar(Exception):

pass

Om man har en abstrakt kö med metoderna enqueue, dequeue och isEmpty kan breddenförstsökningen programmeras ungefär så här.

|  |
| --- |
| class Node(object):  def \_\_init\_\_(self, amount=1.00, currency=1, parent=None):  # problemträdsobjekt  self.amount = amount # belopp  self.currency = currency # valuta, SEK, USD,...  self.parent = None # förälderpekare  def makechildren(node):  # Skapar barn och lägger dom i kön  #Inläsning av växlingskurserna  q = LinkedQ()  urmoder = Node()  q.enqueue(urmoder)  try:  while not q.isEmpty():  node = q.dequeue()  makechildren(node)  # I makechildren görs "raise Klar(kedja)"  print("Ingen lönsam växling")  except Klar as k:  print("Växla fort:", k) |

Istället för try...except kan man låta makechildren returnera True när en lösning hitttats:

foundChain = None  
while not q.isEmpty():

node = q.dequeue()

foundChain = makechildren(node)  
 if foundChain != None:  
 break

# Efter slingan måste man kolla om man hittat en växlingskedja  
if foundChain == None:

print("Det finn ingen lönsam växling")  
else:  
 print("Växla fort:", foundChain)

Metoden makechildren skapar alla barn och lägger sist i kön. Om man vill bli av med dumbarnen kan man ha en global lista **best** med hittills högsta belopp av varje valuta.

**Djupetförstsökning**

Djupetförstsökning skiljer sig från breddenförstsökning i två avseenden:

* Den första lösning man hittar är inte nödvändigtvis den kortaste.
* Metoden fungerar inte om problemträdet har oändligt djup.

**Åttadamersproblemet**

Man ska placera åtta damer på ett schackbräde så att ingen dam står på samma vågräta, lodräta eller diagonala linje som någon annan.

Problemträdets urmoder är ett tomt bräde. Dom åtta barnen har en dam placerad på översta raden, barnbarnen ytterligare en dam på näst översta raden etc. Problemträdet har djup åtta (fler damer kan vi inte placera ut).  
Den första idén man får är ju att representera schackbrädet med en matris. Men lösningen blir enklare om man använder en vektor, där varje element är ett heltal som representerar damens position på just den raden. Då betyder queen[0]=5 att damen på rad noll står i position 5.

***Rekursiv tanke:***   
Att lösa problemet färdigt när det redan står damer på rad 0..row-1 är detsamma som...  
...att för varje tillåten damplacering på rad row lösa problemet färdigt...  
*Basfall:* ... men om row=8 har man hittat en lösning.

|  |
| --- |
| def completePartialSolution(row):  # Fullborda partiell lösning som har damer på rad 0..row-1  if row == n:  printSolution()  else:  for col in range(n):  if posOK(row,col):  queen[row] = col  completePartialSolution(row+1)  def posOK(row, col):  # Kolla om damen på rad row kan slås av damerna ovanför  for i in range(row):  if queen[i] == col:  return False #rakt ovanför  if queen[i]-col == row-i:  return False #snett ovanför (NV)  if col-queen[i] == row-i:  return False #snett ovanför (NO)  return True  def printSolution():  # Skriver ut den lösning som just nu är lagrad i "queen"  for r in range(n):  for col in range(n):  if queen[r] == col:  print("D", end = "")  else:  print("\*", end = "")  print()  print("===============")  n = 8  queen=[None]\*n  completePartialSolution(0) |

Djupetförstsökning kan också programmeras som breddenförstsökningen, med den lilla skillnaden att kön byts mot en *stack*.   
Här följer några fler exempel på problem som kan lösas med djupetförstsökning.

Hitta ut ur labyrint

En välkänd praktisk metod att [utforska en labyrint (Länkar till en externa sida.)Länkar till en externa sida.](http://www.xefer.com/maze-generator), uppfunnen av den förhistoriska datalogen Ariadne, är att ha ett garnnystan med ena änden fastknuten i startpunkten. Man går så långt man kan, markerar med krita var man varit, går bara outforskade vägar framåt och backar en bit längs snöret när man kör fast. Snöret kan representeras av en stack med dom positioner som snöret för tillfället ringlar igenom.

Problemträdet har startpositionen som urförälder, alla positioner på ett stegs avstånd som barn och så vidare. En position som man varit på förut är ett dumbarn.

Problemträdet vi får när vi söker vägen från söt till sur är alltså exempel på en oviktad (alla kanter är lika värda) riktad (*från* startordet *till* barnen), acyklisk (dumträdet hjälper oss att slippa repriser) graf. Breddenförstsökning och djupetförstsökning är två olika sätt att gå igenom alla hörn.

I breddenförst går vi först igenom alla hörn som ligger en kant från starthörnet, sen alla hörn som ligger två kanter bort osv. Om det finns flera lösningar till problemet stöter vi på den närmaste först (men om vi inte bryter där går vi igenom alla hörnen).

Med djupetförst följer vi istället en stig så långt det går via första kanten från starthörnet. När det tar stopp backar vi ett steg (flera vid behov) tills det går att fortsätta framåt igen.

Problemträd är ett sätt att tänka på problem för att kunna lösa dom med grafalgoritmer. Blanda inte ihop problemträd med den abstrakta datastrukturen binärträd!

**Föreläsning 7: Sortering**

Sortering är en av dom vanligaste uppgifterna för ett program. Här följer en beskrivning av de viktigaste sorteringsalgoritmerna!

* Urvalssortering (Selection sort)
* Bubbelsortering (Bubble sort)
* Insättningssortering (Insertion sort)
* Damerna först
* Quicksort
* Mergesort (Samsortering)
* Divide and conquer
* Räknesortering (Distribution count)

**Urvalssortering (Selection sort)**

Vi tänker oss att vi ska sortera en lista med n tal.

* Sök igenom listan efter minsta värdet. (n-1 jämförelser)
* Flytta det till första positionen. (Ett byte)
* Sök efter näst minsta värdet. (n-2 jämförelser)
* Flytta det till andra positionen. (Ett byte)

- - -

Totalt krävs n(n-1)/2 jämförelser och N-1 byten, helt oberoende av hur pass osorterad listan är från början. Tiden är alltså i stort sett proportionell mot kvadraten på N. Man säger att komplexiteten är *O(n2)*.

|  |
| --- |
|  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **6** | 13 | 5 | 2 | **1** | 3 | 7 | 8 | 10 | 12 | 4 | 9 | 11 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | **13** | 5 | **2** | 6 | 3 | 7 | 8 | 10 | 12 | 4 | 9 | 11 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | **5** | 13 | 6 | **3** | 7 | 8 | 10 | 12 | 4 | 9 | 11 |

def urvalssortera(data):

n = len(data)

for i in range(n):

minst = i

for j in range(i+1,n):

if data[j] < data[minst]:

minst = j

data[minst],data[i] = data[i], data[minst]

**Bubbelsortering (Bubble sort)**

En något smartare metod än urvalssortering är denna:

* Byt första och andra om dom står i fel ordning. (En jämförelse, ett byte)
* Byt andra och tredje om dom står i fel ordning. (En jämförelse, ett byte)

- - -

* Bubbla igenom listan gång på gång tills inga byten sker.

Totalt krävs i värsta fall n(n-1)/2 jämförelser och lika många byten. Men om listan är nästan sorterad från början räcker det med några få genomgångar och då blir bubbel snabbare än urval.

Insättningssortering (Insertion sort)

Denna metod känns särskilt naturlig om man får värden ett efter ett (t ex om dom läses in från en fil) och man vill sortera in dom i en lista. Här sorterar vi in ett värde i taget på följande vis:

* Jämför med varje tidigare värde i listan.
* Om det nya värdet är mindre gör vi plats genom att flytta det tidigare värdet ett snäpp åt höger.
* Flytta fram så många värden som behövs för att sätta in nya värdet på rätt plats.
* Stoppa in det nya värdet.

Om alla värden redan finns i listan sorterar vi in ett värde i taget, med början från det andra.

Komplexiteten är i allmänhet *O(n2)* men den är lite snabbare än urvalssortering i praktiken eftersom en flytt är snabbare än ett byte. För att sortera in ett nytt värde i en redan sorterad lista är insättning bäst.

|  |
| --- |
|  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | **6** | 5 | 2 | 1 | 3 | 7 | 8 | 10 | 12 | 4 | 9 | 11 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 13 | **5** | 2 | 1 | 3 | 7 | 8 | 10 | 12 | 4 | 9 | 11 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 6 | 13 | **2** | 1 | 3 | 7 | 8 | 10 | 12 | 4 | 9 | 11 |

def insattningssortera(data):

n = len(data)

for i in range(1, n):

varde = data[i]

plats = i

while plats > 0 and data[plats-1] > varde:

data[plats] = data[plats-1]

plats = plats - 1

data[plats] = varde

Damerna först

Den enklaste sorteringsuppgiften är att sortera om en personlista med damerna först och den smarta algoritmen kallas **damerna-först-sortering**.

1. Sätt ett pekfinger i var ände av listan!
2. Rör fingrarna mot varandra tills vänstra fingret fastnat på en herre och högra fingret på en dam!
3. Låt damen och herren byta plats!
4. Upprepa från 2 tills fingrarna korsats!

Idén kan utvecklas till Quicksort, som är den snabbaste av alla sorteringsalgoritmer.

Quicksort

Damerna-först-algoritmen med två pekfingrar används i Quicksort. Först bestämmer man vilka (små) nyckelvärden som ska kallas damer. Resten kallas herrar och så utför man algoritmen. Listan delas alltså i två segment, det första med små värden, det andra med stora värden. Nu behöver man bara sortera segmenten var för sej. Det här är en rekursiv tanke!

* Bestäm vilka värden som ska kallas damer.
* Partitionera listan så att damerna kommer först.
* Sortera varje segment för sej.

def quicksort(data):

sista = len(data) - 1

qsort(data, 0, sista)

def qsort(data, low, high):

pivotindex = (low+high)//2

# flytta pivot till kanten

data[pivotindex], data[high] = data[high], data[pivotindex]

# damerna först med avseende på pivotdata

pivotmid = partitionera(data, low-1, high, data[high])

# flytta tillbaka pivot

data[pivotmid], data[high] = data[high], data[pivotmid]

if pivotmid-low > 1:

qsort(data, low, pivotmid-1)

if high-pivotmid > 1:

qsort(data, pivotmid+1, high)

def partitionera(data, v, h, pivot):

while True:

v = v + 1

while data[v] < pivot:

v = v + 1

h = h - 1

while h != 0 and data[h] > pivot:

h = h - 1

data[v], data[h] = data[h], data[v]

if v >= h:   
 break

data[v], data[h] = data[h], data[v]

return v

Komplexiteten blir i allmänhet *O(n log N)*. Det beror på att man kan dela listan på mitten log n gånger. Exakt hur snabb den är beror på hur man avgör vilka värden som ska vara damer. Om man tar det första värdet i listan och utnämner det och alla mindre värden till damer, så blir Quicksort mycket långsam för redan nästan sorterade listor! Det bästa är att ta ut tre värden - det första, det sista och något i mitten - och låta det mellersta värdet bestämma vad som är damer. Det kallas *median-of-three*. Man kan visa att komplexiteten då blir *1.4 n log n* i genomsnitt.

Merge Sort

Om man har flera sorterade småfiler är det lätt att *samsortera* dom till en fil. Det här kan man också göra med en lista om man har extrautrymme för att kopiera den till två andra hälften så långa listor. Det här ger en rekursiv tanke!

* Dela listan i två hälften så långa listor.
* Sortera varje halva för sej.
* Samsortera till ursprungliga listan.

Komplexiteten blir *O(n log N)*, lika snabb som quicksort men kräver extra minnesutrymme. Om första halvan av listan *a* ska samsorteras med andra halvan kopierar man först över allting till hjälplistan *b* och sorterar sedan tillbaka från *b* till *a*. Detta förfarande kallas merge och programmeras lämpligen som en egen metod.

def mergesort(data):

if len(data) > 1:

mitten = len(data)//2

vensterHalva = data[:mitten]

hogerHalva = data[mitten:]

mergesort(vensterHalva)

mergesort(hogerHalva)

i, j, k = 0, 0, 0

while i < len(vensterHalva) and j < len(hogerHalva):

if vensterHalva[i] < hogerHalva[j]:

data[k] = vensterHalva[i]

i = i + 1

else:

data[k] = hogerHalva[j]

j = j + 1

k = k + 1

while i < len(vensterHalva):

data[k] = vensterHalva[i]

i = i + 1

k = k + 1

while j < len(hogerHalva):

data[k] = hogerHalva[j]

j = j + 1

k = k + 1

Divide and conquer

En kanske inte så sympatisk härskarteknik som går ut på att så split mellan sina undersåtar för att rikta deras misstro mot varandra istället för mot härskaren kallas på engelska *divide and conquer*. (På svenska säger vi *söndra och härska*, men det passar inte lika bra här.)

Inom datalogi används detta uttryck för att beskriva en lösningsmetod där man delar upp ett problem i två eller flera delproblem och löser dessa var för sig. Delproblemen löses enklast med rekursion!

Quicksort och mergesort är två exempel på divide-and-conquer-principen.

Räknesortering (Distribution count)

Om man vet att det bara finns ett litet antal nyckelvärden, (t ex om man ska sortera Sveriges befolkning efter födelseår), så är distributionsräkning oslagbart snabbt. Det kräver att talen som sorteras in i listan hämtas från en annan lista eller fil.

* Läs igenom filen och räkna hur många det finns av varje nyckelvärde.
* Dela in listan i lagom stora segment för denna distribution.
* Läs filen igen och lägg in varje värde i sitt segment.

Komplexiteten blir O(n), men fungerar alltså bara om man har få nyckelvärden i förhållande till antalet data som ska sorteras.

Om man upprepar förfarandet för varje position (siffra eller bokstav) i de data som ska sorteras får man *radixsortering*!

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 4 | 3 | 3 | 5 | 3 | 5 | 3 | 5 | 4 | 5 | 3 | 4 |
| **3**:6 | | | | | | **4**:3 | | | **5**:4 | | | |
| 3 | 3 | 3 |  |  |  | 4 |  |  |  |  |  |  |

Sortering av större mängder data

Alla metoderna ovan förutsätter att de data som ska sorteras kan lagras i primärminnet. Om så inte är fallet får man ta till *extern sortering* men det ingår inte i den här kursen.

Hur beskriver man en algoritm?

Ur kursens betygskriterier:  
*För****betyg A****ska kraven för betyg C vara uppfyllda, och man ska dessutom kunna modifiera/kombinera algoritmer och datastrukturer för att lösa nya problem. Här ställs också höga krav på tydlighet i algoritmbeskrivningar.*

Skilj på att *förklara hur en algoritm fungerar* och att *konstruera en algoritm*.

* Vad är indata till algoritmen?
* Vad är utdata?
* Punktvis (inte löptext). Ordningen är viktig!
* När avslutas algoritmen?

**Föreläsning 8: Prioritetskö, heap, bästaförstsökning, heapsort**

* Prioritetskö
* Trappa (heap)
* Trappsortering (Heapsort)
* Bästaförstsökning

Prioritetskö

När man poppar en stack får man ut det senast inpushade. När man tar ut något ur en vanlig kö får man tvärtom ut det som legat längst tid i kön. Man skulle kunna se det som att det som stoppas in tidsstämplas och att det påstämplade talet ger prioriteten för uthämtning.

I en prioritetskö stämplas en prioritet på varje objekt som stoppas in och vid uthämtning får man objektet med högst prioritet.

En abstrakt prioritetskö kan ha föjande anrop:

**insert(p,x)** Stoppa in x med påstämplad prioritet p (oftast ett heltal).

**x = delMax()** Hämta det som har högst prioritet.

**isEmpty()** Undersök om prioritetskön är tom.

Om det man vill stoppa in i prioritetskön är ett tal kan man använda talet självt som prioritet och bara skriva insert(x). Hur den då skiljer sej från en stack och från en vanlig kö ser man av följande exempel.

pq.insert(1)

pq.insert(3)

pq.insert(2)

x = pq.delMax() # x blir 3

En kö hade skickat tillbaka det först instoppade talet 1; en stack hade skickat tillbaka det senast instoppade talet, 2; prioritetskön skickar tillbaka det *bästa* talet, 3. I denna prioritetskö betraktar vi största talet som bäst - vi har en så kallad maxprioritetskö. Det finns förstås också minprioritetsköer, där det minsta talet betraktas som bäst.

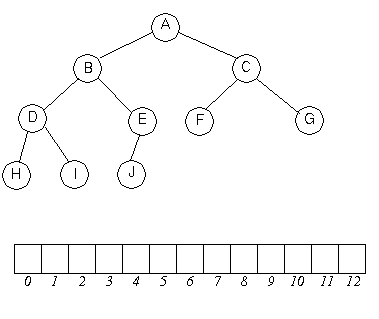
Prioritetsköer har många användningar. Man kan tänka sej en auktion där budgivarna stoppar in sina bud i en maxprioritetskö och auktionsförrättaren efter "första, andra, tredje" gör *pq.delMax()* för att få reda på det vinnande budet. För att hen ska veta vem som lagt detta bud behövs förstås båda parametrarna ipq.insert(p,x).

pq.insert(bud,person) #person är ett objekt med budgivarens namn och bud

winner = pq.delMax() #budgivaren med högst bud

Trappa

Den bästa implementationen av en prioritetskö är en *trappa*, (eng heap), som är en lista tolkad som binärträd.



Roten är tab[1], dess båda barn är tab[2] och tab[3] osv. Vi använder inte tab[0]. Allmänt gäller att tab[i] har barnen tab[2\*i] och tab[2\*i+1]. Trappvillkoret är att *föräldern är bäst*, dvs varje tal ligger på två sämre tal.

Ett nytt tal läggs alltid in sist i trappan. Om trappvillkoret inte blir uppfyllt, dvs om det är större än sin förälder, byter förälder och barn plats och så fortgår det tills villkoret uppfyllts. Det här kallas *upptrappning*och kan i värsta fall föra det nya talet hela vägen upp till toppen, alltså tab[1].

Man plockar alltid ut det översta talet ur trappan och fyller igen tomrummet med det sista talet i trappan. Då är inte trappvillkoret uppfyllt, så man får byta talet och dess största barn. Denna *nedtrappning* upprepas tills villkoret åter gäller.

Både insert och delMax har komplexitet *log N* om trappan har *N* element.

Här följer en rudimentär implementation av en max-heap:

class HeapNode:

def \_\_init\_\_(self, data, priority):

"""data är det objekt vi vill lagra

priority är nyckelvärdet som används för att jämföra objekten"""

self.data = data

self.priority = priority

def \_\_str\_\_(self):  
 return "{0}:{1}".format(self.data, self.priority)

class Heap:

# En max-heap

def \_\_init\_\_(self):

"""Skapar en lista där vi använder element 1..maxsize"""

self.maxsize = 32

self.tab = (self.maxsize+1)\*[None]

self.size = 0

def isEmpty(self):

"""Returnerar True om heapen är tom, False annars"""

return self.size == 0

def isFull(self):

"""Returnerar True om heapen är full, False annars"""

return self.size == self.maxsize

def insert(self, data, priority):

"""Lägger in nya data med nyckeln priority i heapen"""

if not self.isFull():

self.size += 1

self.tab[self.size] = HeapNode(data, priority)

i = self.size

while i > 1 and self.tab[i//2].priority < self.tab[i].priority:

self.tab[i//2], self.tab[i] = self.tab[i], self.tab[i//2]

i = i//2

def delMax(self):

"""Hämtar det största (översta) objektet ur heapen"""

if not self.isEmpty():

data = self.tab[1]

self.tab[1] = self.tab[self.size]

self.size -= 1

i = 1

while i <= self.size//2:

maxi = self.maxindex(i)

if self.tab[i].priority < self.tab[maxi].priority:

self.tab[i],self.tab[maxi] = self.tab[maxi], self.tab[i]

i = maxi

return data.data

else:

return None

def maxindex(self, i):

"""Returnerar index för det största barnet"""

if 2\*i+1 > self.size:

return 2\*i

if self.tab[2\*i].priority > self.tab[2\*i+1].priority:

return 2\*i

else:

return 2\*i+1

Heapsort

Om man stoppar in N tal i en trappa och sedan hämtar ut dom ett efter ett får man dom sorterade. Komplexiteten för denna heapsort blir *O(N log N)*, alltså av lika god storleksordning som quicksort. Visserligen är quicksort lite snabbare, men heapsort har inte quicksorts dåliga värstafallsbeteende. och så kan ju en heap användas till andra saker än sortering också.

heap = Heap()

for word in open("folksagor.txt").read().split():

heap.insert(word, word)

while not heap.isempty():

print heap.delMax()

Bästaförstsökning

Labb 4 behandlar problemet att finna kortaste vägen från FAN till GUD. Man har då ett problemträd med FAN som stamfar/urmoder, på nivån därunder barnen MAN, FIN, FAT osv, på nästa nivå fans barnbarn osv. Om man lägger barnen i en kö kommer man att gå igenom problemträdet nivå för nivå, alltså breddenförst. Om man byter kön mot en stack blir sökningen djupetförst. Med en prioritetskö får man *bästaförstsökning*, dvs det mest lovande barnet prioriteras och får föda barn.

Det här är exempel på en girig algoritm, där man i varje steg väljer den väg som verkar bäst för stunden, utan att reflektera över vad konsekvenserna blir i längden. Ofta ger giriga algoritmer inte den bästa lösningen, men i just det här fallet fungerar det faktiskt! Algoritmen kallas Dijkstra's algoritm (efter den holländske datalogen Edsger W. Dijkstra) och att den fungerar bevisas i fortsättningskursen [DD1352 Algoritmer, datastrukturer och komplexitet (Länkar till en externa sida.)Länkar till en externa sida.](http://www.csc.kth.se/utbildning/kth/kurser/DD1352/).

Det finns flera varianter på Dijkstra's algoritm. Nedan en variant för att räkna ut billigaste vägen från en startnod till en slutnod i en graf med kostnader. För att förklara algoritmen börjar jag med de första stegen.

1. Sätt alla noders prioritet/kostnad till ∞ (i praktiken något maximalt värde)
2. Sätt startnodens prioritet till 0.
3. Stoppa in startnoden i en min-prioritetskö.

Startnoden ligger nu först i kön. I fortsättningen kan noder mitt i prioritetskön behöva omprioriteras. Ett sätt att omprioritera är att införa en remove-funktion som tar bort en nod ur en prioritetskö. Det får inte bli ett "hål" utan i princip sker en upp- och nertrappning på ungefär samma sätt som vid isättning och urtagning från en prioritetskö.

Den fortsatta algoritmen för att hitta billigaste vägen blir ungefär:

1. Plocka ut en nod *p* från prioritetskön.
2. Undersök noden *p*:s barn ett och ett
   1. Räkna ut kostnaden att gå från *p* till barnet.
   2. Om kostnaden är billigare än barnets nuvarande prioritet (från början var kostnaden ∞)
      1. sätt om barnets prioritet till den nya kostnaden
      2. sätt föräldrapekaren att peka på *p*
      3. Omprioritera barnet genom att
         1. plocka bort barnet ur prioritetskön
         2. stoppa in barnet i prioritetskön
3. Upprepa från 1. tills prioritetskön är tom (det går att avbryta tidigare när bättre vägar inte finns)
4. Skriv ut billigaste vägen genom att följa föräldrapekarna från slutnoden till startnoden.

**Exempel 1:** Sök billigaste transport från Teknis till Honolulu. All världens resprislistor finns tillgängliga.

Problemträdets objekt innehåller en plats, ett pris och en förälderpekare. Överst i trädet står Teknis med priset noll. Barnen är alla platser man kan komma till med en transport och priset, till exempelT-centralen, 20.00. Man söker en Honoluluobjekt i problemträdet. Med breddenförstsökning får man den resa som har så få transportsteg som möjligt.

Med bästaförstsökning får man den billigaste resan.

**Exempel 2:** Sök effektivaste processen för att framställa en önskad substans från en given substans. All världens kemiska reaktioner finns tillgängliga med uppgift om utbytet i procent.

Problemträdets objekt innehåller substansnamn och procenttal. Överst i trädet står utgångssubstansen med procenttalet 100. Barnen är alla substanser man kan framställa med en reaktion och utbytet, till exempel C2H5OH, 96%.

Med en max-prioritetskö får man fram den effektivaste process som leder till målet.

**Föreläsning 9: Automater, textsökning**

* Automater
* Textsökning
* KMP-automat
* Boyer-Moore
* Rabin-Karp
* Sökning på webben
* Reguljära uttryck
* Modellering av grafiska gränssnitt

Automater

En portkodsautomat med nio knappar kan se ut så här:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | A | B | C |  |  |  |  |
|  |  |  |  | D | E | F |  |  |  |  |
|  |  |  |  | G | H | I |  |  |  |  |

Anta att den rätta knappföljden är **DEG**. Då har automaten fyra olika *tillstånd*:

1. Starttillstånd.
2. Knapptryckning **D** har just gjorts.
3. Knapptryckningarna **DE** har just gjorts.
4. Knapptryckningarna **DEG** har just gjorts. Låset öppnas.

När automaten är i ett visst tillstånd och en viss knapp trycks ner övergår den i ett nytt tillstånd, och det kan beskrivas med en *övergångsmatris*:

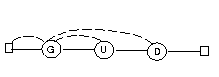
A B C D E F G H I   
 1 1 1 1 2 1 1 1 1 1 Exempel: Om automaten är i tillstånd **3**  
 2 1 1 1 2 3 1 1 1 1 och knapp **D** trycks ner övergår den till  
 3 1 1 1 2 1 1 4 1 1 tillstånd **2**

Man kan också rita en graf med fyra noder (som representerar tillstånden) och en massa bokstavsmärkta pilar (som visar vilka övergångar som finns)

Här är några fler exempel på vad automater kan användas till:

* Söka efter ett ord i en text (se *KMP-automat* nedan).
* Tolka *reguljära uttryck*.
* Beskriva grafiska gränssnitt.
* Kompilatorns/interpretatorns analys av ditt program (se föreläsning om *syntax*).
* Komprimering.
* [Morfologisk (Länkar till en externa sida.)Länkar till en externa sida.](https://sv.wikipedia.org/wiki/Morfologi_(lingvistik)) analys av ord (t ex *o-ut-trött-lig-a*).

Textsökning

Samma automat kan användas för *textsökning*, till exempel för att söka efter **GUD** i bibeln. Bokstav efter bokstav läses och automaten övergår i olika tillstånd. När fjärde tillståndet uppnås har man funnit GUD. Datalogins fader, Donald Knuth, uppfann tillsammans med Vaughan Pratt en enkel metod att konstruera och beskriva automaten. James H. Morris kom på samma sak oberoende av dom två! Därför kallar vi automaten [KMP-automat (Länkar till en externa sida.)Länkar till en externa sida.](http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/kpm-example.html). En KMP-automat har bara en framåtpil och en bakåtpil från varje tillstånd. Så här blir den:   
  
Ett nolltillstånd har skjutits in längst till vänster. Automaten startar emellertid i tillstånd 1, som har ett G i noden. Den tjuvtittar på första bokstaven i bibeln, och om det är ett G läser den G-et och går till höger. Annars följer den bakåtpilen utan att glufsa bokstaven. I nolltillståndet glufsar den alltid en bokstav och går till höger. Koden blir i princip så här, om vi först antar att bokstäverna lästs in i en kö, som har en extra metod *peek()*, med vilken man kan tjuvtitta på första bokstaven.

i = 1 # starttillståndet

while i < 4:

if i == 0 or q.peek() == sokord[i]:

i = i+1

q.dequeue()

else:

i = next[i];

Här är sokord[i] i-te bokstaven i det sökta ordet och next[i] det tillstånd man backar till från tillstånd i. Nextvektorn (bakåtpilarna) i vårt exempel blir

|  |  |
| --- | --- |
| **i** | **next[i]** |
| 1 | 0 |
| 2 | 1 |
| 3 | 1 |

Om vi i stället söker efter ADAM i bibeln blir KMP-automaten så här:   
Nextvektorn för ADAM blir alltså den här:

|  |  |
| --- | --- |
| **i** | **next[i]** |
| 1 | 0 |
| 2 | 1 |
| 3 | 0 |
| 4 | 2 |

För GUD gick bakåtpilen från tillstånd 3 till tillstånd 1, men här vore meningslöst att två gånger i rad kolla om bokstaven är A. Bakåtpilen från tillstånd 4 till tillstånd 2 kräver också en förklaring. Om vi har sett ADA och nästa bokstav inte är ett M kan vi i alla fall hoppas att det A vi just sett ska vara början på ADAM. Därför backar vi till tillstånd 2 och undersöker om det möjligen kommer ett D. Reglerna för hur next-vektorn bildas kan sammanfattas så här:

* next[1]=0.
* Annars är next[i]=1 om ordet inte upprepar sej.
* ...men om de j senaste bokstäverna vi sett bildar början på sökordet sätts next[i]=j+1.
* ...men om bokstav j+1 är samma som bokstav i sätts i stället next[i]=next[j+1].

Kom ihåg att next-vektorn bara behövs när det kommer ett dåligt tecken.

Om den sträng vi söker efter är m tecken lång och texten vi söker i är n tecken lång kräver KMP-sökning aldrig mer än *n+m* teckenjämförelser och är alltså O(n+m). Metoden går igenom texten tecken för tecken - man kan alltså läsa ett tecken i taget t ex från en fil vilket är praktiskt om texten är stor.

Boyer-Moore

Då hela texten vi söker i finns i en lista kan man istället använda Boyer-Moores metod. Den börjar med att försöka matcha sista tecknet i söksträngen, som är *m* tecken lång. Om motsvarande tecken i texten inte alls förekommer i söksträngen hoppar den fram m steg, annars flyttar den fram så att tecknet i texten passar ihop med sista förekomsten i söksträngen.   
Exempel: Vi söker efter **TILDA** i texten **MEN MILDA MATILDA**.

MEN MILDA MATILDA

TILDA

TILDA

TILDA

TILDA

MEN MILDA MATILDA

Boyer-Moore är *O(n+m)* i värsta fallet, men ca *n/m* steg om texten vi söker i består av många fler tecken än dom som ingår i söksträngen, så att vi oftast kan hoppa fram *m* steg.

När du skriver Ctrl-S för att söka efter en sträng i Emacs är det Boyer-Moore som används.

J Strother Moores egen html-visualisering[av Boyer-Moore (Länkar till en externa sida.)Länkar till en externa sida.](http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/index.html).

Visualisering av [naiv sökning, KMP och Boyer-Moore (Länkar till en externa sida.)Länkar till en externa sida.](http://whocouldthat.be/visualizing-string-matching/)

Rabin-Karp

Beräknar en hashfunktion för söksträngen och jämför med hashfunktionen beräknad för alla avsnitt av längden *m* i texten. Låt oss söka efter "TILDA" i texten "MEN MILDA MATILDA". Med Pythons hashfunktion hash() % 17 får vi hash(TILDA)=16

hash(MEN M) = 6

hash(EN MI) = 15

hash(N MIL) = 9

hash( MILD) = 1

hash(MILDA) = 7

hash(ILDA ) = 13

hash(LDA M) = 15

hash(DA MA) = 12

hash(A MAT) = 0

hash( MATI) = 5

hash(MATIL) = 12

hash(ATILD) = 3

hash(TILDA) = 16

För att snabba upp beräkningarna använder man hela tiden förra hashvärdet. Komplexiteten blir   
*O(nm)* i värsta fallet, men i praktiken bara *O(n+m)*.

Sökning på webben

När man använder en *sökmotor*, t ex Google, för att hitta webbsidor som innehåller ett visst ord skulle alla ovanstående metoder bli för tidsödande. Där slår man istället upp ordet i ett index som skapats i förväg. Hur det fungerar kan man läsa mer om i kursen [DD2418, Språkteknologi (Länkar till en externa sida.)Länkar till en externa sida.](https://www.kth.se/social/course/DD2418). En sökalgoritm Google använder kallas map reduce. Algoritmen delar upp sökningen på flera parallella processorer, om det kan man läsa om i kursen parallell programming ID1217.

Reguljära uttryck

Om man t ex skulle vilja söka efter lab1, Lab2, eller labb3 så kan man använda ett *reguljärt uttryck* för att beskriva söksträngen. Ett reguljärt uttryck består av tecken och metatecken som tillsammans utgör ett sökmönster. Metatecken (t ex \* och +) har särskild innebörd. Här följer några regler:

* a\* matchar noll eller flera a:n
* a+ matchar ett eller flera a:n
* a+? matchar ett eller flera a:n men försöker med så få a:n som möjligt
* {n} matchar n gånger
* a? matchar ett eller inget a
* . matchar alla tecken utom radslut
* [a-zA-Z] matchar alla engelska bokstäver
* [abc] matchar a, b eller c
* [^abc] matchar vilket tecken som helst utom a, b eller c
* X|Y matchar uttrycket X eller uttrycket Y
* \. matchar en punkt. Tecknet "\" används för att rädda det efterföljande tecknet från att tolkas som ett metatecken.
* (ab) skapar en grupp. T ex matchar (ab)+ ett eller flera ab:n

Det reguljära uttrycket *[Ll]abb?[1-7]* kan användas för att hitta alla labbvarianter vi eftersökte ovan.

Det finns UNIX-kommandon som använder reguljära uttryck: *egrep* som letar efter ett reguljärt uttryck i en textfil, *sed* som kan byta ut valda delar av en fil (där delarna väljs ut med hjälp att ett reguljärt uttryck) *awk*som är ett programmeringsspråk som bygger på att vissa instruktioner ska utföras vid varje matchning av ett uttryck.

Pythonmodulen r*e* har funktionalitet för reguljära uttryck. Exempel:

import re

pattern="[Ll]abb?[1-7]"

sometext="Lab5, labb 6, labb7 och lab8"

print(re.findall(pattern,sometext))

# Utskriften blir ['Lab5', 'labb7']

Bakom kulisserna i re-modulen skapas automater som kan kontrollera om indata matchar det givna uttrycket.

Ibland kan det bli väldigt mycket jobb att söka om man skriver ett onödigt komplicerat reguljärt uttryck. Antag att du vill leta efter efter *(aa+)b* dvs minst två a följt av b men istället angett reguljära uttrycket *(a+a+)b* villket borde ge samma resultat.

Om man letar i strängen *aaaaaaaaab* (9 a och ett b) så kommer reguljära uttrycket *(a+a+)b*  att först matcha 9a med första a+ därefter misslyckas med att matcha nästa a+ (eftersom nästa tecken är ett b). Då backas första a+ matchningen till att matcha 8a, det andra a+ matchar ett a och därefter matchas b och det hela är klart.

Men om texten man matchade mot inte innehöll ett b på slutet, t.ex. *aaaaaaaaak* då skulle *a+a+* göras om flera gånger. Eftersom det inte kommer ett b kommer sökningen att misslyckas men inte förrän alla sökmöjligheter har uttömts. Det reguljära uttrycket *aa+b* hade istället brutit redan på första försöket.

**Tabell: sökning med reguljära uttrycket *(a+a+)b* på strängen *aaaaaaaaak***

|  |  |  |
| --- | --- | --- |
| a+ | a+ | b |
| aaaaaaaaa |  |  |
| aaaaaaaa | a |  |
| aaaaaaa | aa |  |
| aaaaaa | aaa |  |
| aaaaa | aaaa |  |
| aaaa | aaaaa |  |
| aaa | aaaaaa |  |
| aa | aaaaaaa |  |
| a | aaaaaaaa |  |

**Modellering av grafiska gränssnitt**

Många problem kan modelleras med automater. Objektorienterad programmering lämpar sig väl för att implementera tillstånden.

Exempel:   
I många grafiska gränssnitt kan man markera text genom att trycka ner musknappen och hålla den nedtryckt medan man drar musen över texten. Det här kan vi se som en automat med ett normaltillstånd och ett markeringstillstånd, där musknappstryck/släpp ger övergång mellan tillstånden.

Objektmodellering och tillståndsprogrammering kan man läsa mer om i kursen [DD2385, Programutvecklingsteknik (Länkar till en externa sida.)Länkar till en externa sida.](https://www.kth.se/social/course/DD2385).   
  
Är man särskilt intresserad av grafiska gränssnitt finns kursen [DH2323, Datorgrafik med interaktion (Länkar till en externa sida.)Länkar till en externa sida.](https://www.kth.se/social/course/DH2323)   
  
Den som vill veta hur automater egentligen fungerar kan läsa Dilians kurs [DD1372, Automata and languages (Länkar till en externa sida.)Länkar till en externa sida.](https://www.kth.se/social/course/DD2372/) (men den kräver en del andra förkunskaper).

**Föreläsning 10: Syntax, rekursiv medåkning, testning**

* Syntax för formella språk
* Rekursiv medåkning
* Syntaxkontroll med stack

Syntax för formella språk

Ett *formellt språk* är en väldefinierad uppsättning textsträngar som kan vara oändligt stor, till exempel alla Python-program, eller ändligt stor, till exempel alla månadsnamn. Det bästa sättet att definiera ett språk är med en *syntax* (observera betoning på sista stavelsen!), det vill säga en grammatik.

Exempel: Språket som består av satserna JAG VET, JAG TROR, DU VET och DU TROR definieras av syntaxen  
<Sats> ::= <Subj> <Pred>   
<Subj> ::= JAG | DU   
<Pred> ::= VET | TROR

I syntaxen ovan har vi tre *omskrivningsregler*. Varje regel består av ett vänsterled med en *icke-slutsymbol* (t ex <Sats> ovan) och ett högerled som talar om vad man kan ersätta vänsterledet med. I högerledet får det förekomma både icke-slutsymboler och *slutsymboler* (t ex JAG i exemplet ovan). Tecknet | betyder *eller*.

Meningar av typen   
JAG VET ATT DU TROR ATT JAG VET OCH JAG TROR ATT DU VET ATT JAG TROR definieras nu så här:

<Mening> ::= <Sats> | <Sats><Konj><Mening>

<Konj> ::= ATT | OCH

Med hjälp av den rekursiva definitionen av Mening har vi plötsligt fått en syntax som beskriver en oändlig massa meningar!

Syntaxen för programspråk beskrivs ofta i BNF.

Man kan förstås beskriva syntaxen för BNF i BNF, och det ser ut så här:

<syntax> ::= <regel> | <regel> <syntax>

<regel> ::= "<" <regelnamn> ">" "::=" <uttryck> <radslut>

<uttryck> ::= <lista> | <lista> "|" <uttryck>

<radslut> ::= <RETURTECKEN> | <radslut> <radslut>

<lista> ::= <term> | <term> <lista>

<term> ::= <slutsymbol> | "<" <regelnamn> ">"

<slutsymbol> ::= '"' <text> '"' | "'" <text> "'"

En kompilator fungerar ungefär så här:

källkod --> lexikal analys --> syntaxanalys --> semantisk analys -->

--> kodgenerering --> målkod

* Under en *lexikal analys* sållas oväsentligheter såsom blanktecken och kommentarer bort samtidigt som symboler vaskas fram.
* *Syntaxanalysen* (parsningen) kontrollerar att programmet följer syntaxen och skapar ett *syntaxträd*.
* Sen följer *semantisk analys* där kompilatorn ser efter vad programmet betyder.
* Sist sker *kodgenerering* där programmet översätts till målkod (t ex lab6.pyc).

Den första delen av syntaxanalysen, att kontrollera om ett program följer syntaxen, kan göras med *rekursiv medåkning* eller med en stack.

Rekursiv medåkning (recursive descent)

Denna metod för syntaxanalys ska du använda dig av i labb 8: Formelkoll   
För varje symbol i grammatiken skriver man en inläsningsmetod. Om vi vill analysera grammatiken vi började med behöver vi alltså metoderna:

* readMening()
* readSats()
* readSubj()
* readPred()
* readKonj()

Flergrenade definitioner kräver tjuvtitt medq.peek() som vi har lagt till i klassen WordQueue. När något strider mot syntaxen låter vi ett särfall skickas iväg. Här följer ett program som undersöker om en mening följer vår syntax.

# Syntaxkontroll

from wordqueue import WordQueue

class Grammatikfel(Exception):

pass

def readMening(q):

readSats(q)

if q.peek() == ".":

q.dequeue()

else:

readKonj(q)

readMening(q)

def readSats(q):

readSubj(q)

readPred(q)

def readSubj(q):

word = q.dequeue()

if word == "JAG":

return

if word == "DU":

return

raise Grammatikfel("Fel subjekt: " + word)

def readPred(q):

word = q.dequeue()

if word == "TROR":

return

if word == "VET":

return

raise Grammatikfel("Fel predikat: " + word)

def readKonj(q):

word = q.dequeue()

if word == "ATT":

return

if word == "OCH":

return

raise Grammatikfel("Fel konjunktion: " + word)

def printQueue(q):

while not q.isEmpty():

word = q.dequeue()

print(word, end = " ")

print()

def storeSentence(mening):

q = WordQueue()

mening = mening.split()

for ordet in mening:

q.enqueue(ordet)

q.enqueue(".")

return q

def kollaGrammatiken(mening):

q = storeSentence(mening)

try:

readMening(q)

return "Följer syntaxen!"

except Grammatikfel as fel:

return str(fel) + " före " + str(q)

def main():

q = WordQueue()

mening = input("Skriv en mening: ")

resultat = kollaGrammatiken(mening)

print(resultat)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Testa med unittest

Här är ett exempel på hur man kan testa programmet med unittest (mer om testning på föreläsningen efter tentaveckan).

import unittest

from syntax import \*

class SyntaxTest(unittest.TestCase):

def testSubjPred(self):

""" Testar Subj och Pred """

self.assertEqual(kollaGrammatiken("JAG VET"), "Följer syntaxen!")

def testFelKonj(self):

self.assertEqual(kollaGrammatiken("JAG VET MEN"), "Fel konjunktion: MEN före . ")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

Man kan översätta assert med *"make certain that".*Det finns flera assert-anrop man kan använda t.ex.

Tester består av hårdkodade startvärden och ett förväntat utfall som man jämför med. Om testutfallen stämmer med de förväntade utfallen så har testet lyckats.

Om man skrivit ett program som ber användaren mata in värden så kan man testa det med hårdkodade värden genom att skicka fördefinierade invärden i en indatafil med |

Skapa en indata fil som heter *indata.txt* och en fil med förväntade svarsvärden *facit.txt*

|  |  |
| --- | --- |
| *indata.txt* | *facit.txt* |
| *JAG VET ATT DU TROR* | *Följer syntaxen!* |
|  |  |

Mata in indata till programmet med |

> more indata.txt | python3 mittsyntaxprogram.py

Resultatet skrivs ut på skärmen. För att jämföra om resultatet gick bra behöver du spara ner utfallet på disk och jämföra filerna med t.ex. *diff*

> more indata.txt | python3 mittsyntaxprogram.py > utfall.txt  
> diff utfall.txt facit.txt

En fördel med att använda unit-test är att man kan samköra och administrera många programmerares tester på samma sätt.

Syntaxkontroll med stack

Ett alternativt sätt att kontrollera om inmatningen följer en syntax är att använda en stack. Som exempel tar vi upp en vanlig källa till fel: omatchade parenteser. Så här kan man använda en stack för att hålla reda på parenteserna:

1. Skapa en tom stack
2. Slinga som läser symboler (här:tecken) tills inmatningen tar slut
   * Om symbolen är en startsymbol (t ex[), lägg den på stacken.
   * Om symbolen är en slutsymbol (t ex ]), titta på stacken. Om stacken är tom eller om den symbol som poppar ut inte matchar slutsymbolen har vi ett syntaxfel.
3. När inmatningen tar slut - kolla om stacken är tom. Om den inte är tom har vi fått ett syntaxfel.

Men vad händer om man lagt in parenteser inuti en kommentar? Vi vill att alla tecken inuti kommentaren ska läsas bort. Lösningen är att låta programmet bete sig som en automat med flera tillstånd.

1. Letar parenteser att lägga på stacken. Övergår till tillstånd 2 om den upptäcker en kommentar, till tillstånd 3 om inmatningen tar slut.
2. Inuti kommentaren - läser bort tecken. Återgår till tillstånd 1 om kommentaren tar slut.
3. När inmatningen tar slut - kollar om stacken är tom. Om den inte är tom har vi fått ett syntaxfel.

Den som vill skriva sitt eget programmeringsspråk måste först skriva en syntax för språket, och sedan ett program som kan tolka språket.

Föreläsning 11: Komprimering

* Tal i bas 2, 8 och 16
* Filformat (.txt .doc .jpg etc)
* Storleksordningar data
* Komprimering
* Följdlängdskodning (run-length encoding)
* Huffmankodning
* Lempel-Ziv-kodning
* Entropi
* Komprimering av bilder
* Komprimering av rörliga bilder
* Komprimering av ljud
* Felkorrektion
* Hammingavstånd

Binära tal (och andra baser)

Till vardags använder vi decimala talsystemet med tio siffror: 0-9. Andra talsystem finns, t ex binära tal (två siffror: 0-1), oktala tal (åtta siffror: 0-7) och hexadecimala tal (sexton siffror:0-F).

|  |  |  |  |
| --- | --- | --- | --- |
| **Decimalt** | **Binärt** | **Oktalt** | **Hexadecimalt** |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |
| 16 | 10000 | 20 | 10 |
| 17 | 10001 | 21 | 11 |
| 18 | 10010 | 22 | 12 |
| 19 | 10011 | 23 | 13 |
| 20 | 10100 | 24 | 14 |

Det är extra enkelt att konvertera binära tal till oktala resp hexadecimala: tre binära siffror ger en oktal siffra, och fyra binära siffror ger en hexadecimal siffra.

I python kan man skriva binära, oktala och hexadecimala tal direkt, med prefixet **0b**,**0o** respektive **0x**.  
Minnesadresser skrivs ut som hexadecimala tal, till exempel:

<\_\_main\_\_.MittEgetObjekt instance at 0x7f7cf8cb22d8>

Komprimering

Komprimering innebär att man använder någon metod för att minska storleken på en fil. Vi skiljer mellan *förlustfri komprimering* (non-lossy compression) där det går att dekomprimera för att få tillbaka filen i ursprungligt skick och *förstörande komprimering* (lossy compression) där man tar bort data. Att det går att komprimera utan att förstöra en fil beror på att filer oftast har redundans, dvs innehåller mer än nödvändigt.

Följdlängdskodning - RLE

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

*ÅÅÅÅH! JAAAAAAA! AAAAAAAAAAAAH.*

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

4ÅH! J7A! 12AH.

Men om grundtexten innehåller siffror blir det svårtolkat. Därför väljer vi ett bryttecken, t ex **§**, som vi är säkra på inte kommer att förekomma i texten.

§4ÅH! J§7A! §12AH.

Algoritmen blir enkel, men tyvärr inte så användbar för textkomprimering eftersom de flesta texter inte innehåller längre följder av samma tecken.

Huffmankodning

Om vi vet hur vanliga olika tecken är i texten kan vi ställa upp en tabell där vi för varje tecken kan ange sannolikheten för att ett visst tecken ska dyka upp. I David A. Huffmans metod kodar man varje tecken med ett binärt tal, där vanligare tecken får kortare koder. Algoritmen som beräknar vilket tecken som ska få vilken binär kod går ut på att man ritar upp ett binärt träd, där varje tecken ses som ett löv. Sedan numrerar man trädets grenar med 0 och 1 och följer trädet från roten ut till varje löv för att se koderna.

1. Sortera tecknen som ska kodas i stigande sannolikhetsordning.
2. Rita grenar från de två tecken som har lägst sannolikhet och låtsas att vi har ett nytt tecken med sannolikhet som är summan av deras sannolikheter. Numrera ena grenen med 0 och andra med 1.
3. Sortera in det nya "låtsastalet".
4. Upprepa punkt 2 tills alla tecken (även de nytillkomna) kommit med. Roten bör få sannolikhet 1.
5. Börja från roten och följ grenarna ut till ett löv. Samla nollor och ettor på vägen - dessa ger koden för lövets tecken.

Vi illustrerar algoritmen med ett exempel. En genomgång av skräcklitteraturen ger följande frekvenstabell:

|  |  |  |
| --- | --- | --- |
| Huffmankod | Tecken | Sannolikhet |
|  | G | 0.05 |
|  | R | 0.05 |
|  | ! | 0.1 |
|  | . | 0.15 |
|  | A | 0.15 |
|  | H | 0.2 |
|  | I | 0.3 |

Texten *HAHA!****IIIIIIH!****AHRG...* skulle alltså kodas som

01 101 01 101 001 11 11 11 11 11 11 01 001 101 01 0001 0000 100 100 100

Huffmankodning är en *statistisk metod*.

Lempel-Ziv

Alla texter följer inte statistiken. Här följer ett utdrag ur romanen *Gadsby* av Ernest Vincent Wright (1872-1939).

*IF YOUTH, THROUGHOUT all history, had had a champion to stand up for it; to show a doubting world that a child can think; and, possibly, do it practically; you wouldn't constantly run across folks today who claim that ''a child don't know anything.'' A child's brain starts functioning at birth; and has, amongst its many infant convolutions, thousands of dormant atoms, into which God has put a mystic possibility for noticing an adult's act, and figuring out its purport.*

Jacob Ziv och Abraham Lempel uppfann en förutsättningslös metod som anpassar sig till indata. Principen är att man går igenom filen och bygger en ordlista som används för kodningen. Lempel-Ziv finns i ett otal olika varianter: LZ77, LZSS, LZFG, LZW, LZMW, LZAP, LZY, LZP, osv. Så här fungerar LZW (en variant gjord av T. Welch):

* Läs in tecken för tecken och slå ihop till en sträng s.
* Fortsätt på det viset så länge som strängen redan finns med i ordlistan.
* Så småningom får vi en sträng som inte finns i ordlistan (s finns med men inte s+c).
* Skriv då ut koden för strängen s, skriv ut tecknet c, och lägg in s+c i ordlistan.

def lzw(text):  
 table = Table()  
 q = Queue()  
 for c in text: # spara texten tecken för tecken i en kö  
 q.enqueue(c)  
  
 s = ""  
 kodtext = "" # här sparas det kodade meddelandet  
 while not q.isEmpty():  
 print(table)  
 c = q.dequeue()  
 if table.exists(s+c):  
 s = s + c  
 else:  
 kodtext += str(table.code(s)) + c  
 table.add(s + c)  
 s = ""  
 if not s == "":  
 kodtext += str(table.code(s))  
  
 return kodtext

Klassen Table (som används för ordlistan) är tänkt att vara en datastruktur där man kan stoppa in strängar med add(), kolla om en sträng finns med exists(), och få ut en kod för en given sträng med code().   
LZ-komprimering används i många komprimeringsprogram, t ex compress, zip, WinZip och GZip (här i kombination med Huffmankodning).

Exempel: Använd LZW-algoritmen ovan för att komprimera NÄSSNUVSNORSNOK

Om vi använder en vektor som tabell och för enkelhets skull kodar strängarna med vektorindex får vi tabellen:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **code** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **sträng** | "N" | "Ä" | "S" | "SN" | "U" | "V" | "SNO" | "R" | "SNOK" |

och det komprimerade ordet blir: NÄS2NUV3OR6K

Entropi

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

Anta att vi har en teckenmängd m1, m2,...,mn (t ex alfabetet) och att sannolikheten för att tecknet mi ska förekomma är P(mi).   
Då är L(mi)=-log(P(mi)) minimilängden för ett kodord för tecknet mi och   
Lmedel = P(m1)\*L(m1) + ... + P(mn)\*L(mn)   
medellängden för koderna (entropin).

Komprimering av bilder

Det är vanligt att varje bildpunkt (pixel) i en färgbild representeras med ett 24-bitars binärt tal (vilket ger oss åtta bitar för vardera rött, grönt resp blått). Då tar en färgbild 100x100 pixlar 24000 bitar, dvs 24 kB och en bild som täcker en 600x800-skärm tar 11.5 MB.

|  |  |  |
| --- | --- | --- |
|  |  |  |

Vilken redundans kan finnas i en bild? Vissa färger kanske är vanligare, så att vi kan använda Huffmankodning för att få kortare koder för dessa. I foton är närliggande pixlar ofta lika (blå himmel t ex), likaså i streckteckningar och grafer (bara svarta och vita pixlar). Där kan man använda RLE genom att räkna antal vita resp svarta pixlar i följd. Även varianter av LZW kan användas - då innehåller tabellen pixelinfo istället för strängar! Vi kan också använda förstörande komprimering för att ta bort information som ögat ändå inte ser.

GIF (Graphics Interchange Format) är ett filformat för bilder där färgkodningen görs med 8 bitar, dvs man får 28=256 färger. Sen används en variant av LZW för att komprimera. Den komprimeringen är förlustfri och storleken minskas med ungefär faktorn 4.

JPEG (Joint Photographic Experts Group) är bättre för foton och andra bilder där närliggande pixlar har liknande färger. Färgbilder delas upp i en belysningsdel och en färgdel, där färgdelen komprimeras med förstörande komprimering eftersom ögat är mindre känsligt för färgförändringar. Sen används en kombination av RLE och Huffmankodning för att koda grupper av pixlar. Komprimeringsgraden är parameter till algoritmen, så man kan bestämma själv hur hårt man vill komprimera. Färgkodningen görs med 24 bitar, dvs 224 (nästan 17 miljoner) färger.

Komprimering av rörliga bilder

En videofil innehåller massor av bilder och dessutom ljud så det är extra viktigt att kunna komprimera såna. Dekomprimeringen måste gå snabbt om man direkt ska kunna se filmen i realtid. Det mest kända formatet för rörliga bilder är MPEG (Moving Picture Experts Group). MPEG är egentligen en samling standarder för kombinationer av ljud och video.

Komprimeringen av video-delen kan delas upp i *bildkomprimering* av varje enskild bildruta och *tidskomprimering* där man utnyttjar likhet mellan på varandra följande bilder.

För bildkomprimeringen används i regel JPEG. För tidskomprimeringen finns ett antal olika metoder:

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

Tidskomprimeringen kan göra det knepigare att redigera filmen.

Komprimering av ljud

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

När det gäller ljud kan man också använda förstörande metoder Två exempel på sådana är *tystnadskomprimering* där man ersätter mycket svaga ljud med tystnad och *companding* där man minskar ordlängden för varje ljudpunkt (t ex från 16 till 12 bitar).

MP3 (MPEG Audio Layer-3 encoding) använder en kombination av tekniker där man utnyttjar en modell av den mänskliga hörseln samt Huffmankodning.

**Felkorrektion**

Vill man gardera sig mot fel kan man lägga till redundans (motsatsen till komprimering). Det finns många olika sätt att göra det på, här följer några exempel:

* + Kontrollsiffra (t ex sista siffran i ett personnummer).
  + Skicka kopior av hela meddelandet, minst tre behövs om man ska kunna korrigera.
  + Paritetsbitar, att man lägger till en etta eller nolla till ett binärt tal för att göra det udda. Ett jämnt tal innebär att nån bit är fel.
  + Hammingavstånd: Lägg till så många extrabitar till koden så att varje enbitsfel ger ett kodord som skiljer sig i en bit från det korrumperade kodordet, men i flera bitar från alla övriga kodord. Två kodord har Hammingavstånd **d** om dom skiljer sig åt i d bitar.  
    En koduppsättning har Hammingavstånd **d** om *alla* kodord är minst d ifrån varann. Givet koderna nedan - hur ska vi tolka meddelandet **10010 01110 10101** ?

|  |  |  |
| --- | --- | --- |
|  | A | 01011 |
|  | F | 10010 |
|  | I | 01100 |
|  | N | 10101 |

 Hamminavstånd exempel

A och F skiljer sig 3 bitar, A och N skiljer sig fyra bitar. Skillnaderna visas i tabellen nedan. Hammingavståndet för koduppsättningen är minsta avståndet mellan två koder d.v.s 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | F | I | N |
| A |  | 3 | 3 | 4 |
| F |  |  | 4 | 3 |
| I |  |  |  | 3 |
| N |  |  |  |  |

Föreläsning 12: Kryptering, datasäkerhet

* Kryptering:
  + Transpositionschiffer
  + Cesarchiffer
  + Bokchiffer
  + One-time pad
  + Säker nyckelöverföring
  + RSA
  + Datasäkerhet
  + Lösenord
  + Validera data
  + Dekompilera kod

Kryptering

Att *kryptera* ett meddelande innebär att vi kodar det så att det blir oläsligt för alla utom den som vet hur man *dekrypterar* det. Kryptering används i alla sammanhang där man har anledning att hålla något hemligt, t ex vid överföring av kontokortsnummer över internet eller när man vill skicka lägesrapporter från diktaturer.

Olika former av kryptering har använts så länge man har velat ha skriftliga hemligheter. Redan dom gamla grekerna hade flera system för kryptering.

Rollista:

* *Alice* som vill skicka ett meddelande till Bob
* *Bob* som vill få meddelandet från Alice (oläst)
* *Eve* (av engelskans *eavesdrop* - tjuvlyssna) som försöker avkoda meddelandet

Transpositionschiffer

I det antika Grekland användes en [scytale (Länkar till en externa sida.)Länkar till en externa sida.](http://upload.wikimedia.org/wikipedia/commons/thumb/5/51/Skytale.png/320px-Skytale.png) - en dekrypteringspinne för att avkoda hemliga meddelanden skrivna på en remsa skinn. Avsändaren (Alice) och mottagaren (Bob) har varsin pinne av samma diameter. Avsändaren lindar skinnremsan som en spiral runt pinnen och skriver sitt meddelande längs med pinnen. Remsan blir då obegriplig tills den hamnar i mottagarens händer.

Samma effekt kan vi få genom att skriva meddelandet i en mxn-matris och bilda det kodade meddelandet genom att ta varje kolumn i texten. Den som känner till matrisens storlek kan lätt avkoda det. Man brukar skippa mellanslag och skiljetecken för att göra det svårare att knäcka koden.

JAGGÖ

MMERP

ENGAR

NAIDE

NGAML

AEKEN

kodas alltså som:

JMENNAAMNAGEGEGIAKGRADMEÖPRELN

Caesarchiffer

Julius Caesar lär ha använt denna metod:   
A byts mot D  
B byts mot E  
C byts mot F  
osv.

En variant är *rot13* där man förskjuter varje bokstav 13 steg. Hamnar man utanför alfabetet roterar man runt till början igen.

def rot13(meddelande):

alfabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

kod = ""

for bokstav in meddelande:

index = (alfabet.find(bokstav) + 13) % 26

kod = kod + alfabet[index]

return kod

Om vi skippar ÅÄÖ får vi ett alfabet som består av 26 tecken, vilket innebär att vi kan använda samma funktion för dekryptering!

rot13("HEMLIGHETER") blir URZYVTURGRE

rot13("URZYVTURGRE") blir HEMLIGHETER

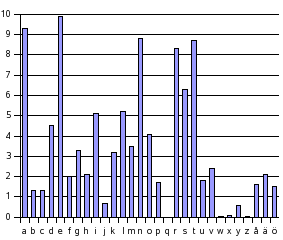
Ännu hemligare blir det om vi slumpar fram en mappning istället för att flytta fram ett visst antal steg.

ABCDEFGHIJKLMNOPQRSTUVWXY

|||||||||||||||||||||||||

FSXNVDQBCULRHKTPAOMWJEIGY

Alla dessa chiffer är dock enkla att knäcka för Eve, som har statistik över hur ofta varje bokstav förekommer i det aktuella språket. Så här ser det ut för svenska:



Bokchiffer

Det är ett enkelt men mycket effektivt chiffer där man använder en bok som nyckel. Man letar rätt på ordet i boken och skriver ner sidnummer samt ordets nummer på sidan, t ex

534 13 127 220 10 109 220 129

Den ena siffran anger sidan och den andra anger vilket ord på sidan det är. (Man kan använda bokstäver istället för ord, men då ska det vara en tjock bok.) Alice och Bob måste i förväg ha kommit överens om vilken bok det gäller utan att Eve hör).

One-time pad

One-time pad är en oknäckbar krypteringsmetod som använder sig av en slumpad nyckel.

Givet ett meddelande på binär form:

* Slumpa fram en nyckel med lika många binära siffror som meddelandet har
* Gör bitvis xor mellan meddelandet och nyckeln.

|  |  |  |
| --- | --- | --- |
|  | | |
| värde 1 | värde 2 | XOR resultat |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

 Det resulterande kodade meddelandet kan bara dekrypteras av den som har den slumpade nyckeln.

Alice och Bob måste ha varsitt exemplar av en "pad" med slumpade siffror, som inte Eve har tillgång till.   
Alice använder första sidan i sitt block för att kryptera ett meddelande och river sedan av första arket och äter upp det.   
Bob dekrypterar med sin första sida, river av och eldar upp arket.

Säker nyckelöverföring

Alice vill skicka ett exemplar av blocket till Bob utan att Eve (som jobbar som brevbärare) kan kopiera det. Hur ska hon göra?

* Alice lägger blocket i en ibrottssäker låda och sätter sitt A-lås på det. Endast hon själv har A-nyckeln.
* Hon skickar lådan till Bob (Eve kan inte låsa upp den).
* Bob kan inte öppna lådan ännu. Han sätter på sitt B-lås. Endast Bob har B-nyckeln.
* Han skickar tillbaka lådan till Alice.
* Alice låser upp A-låset och plockar bort det. Nu är lådan enbart låst med B-lås.
* Hon skickar lådan till Bob, som låser upp den och får sitt block.

Asymmetrisk kryptering

Ett problem med många chiffer är att man måste berätta för mottagaren hur hon ska dechiffrera meddelandet. Hur är man säker på att den viskningen inte blir avlyssnad?

Asymmetrisk kryptering (RSA) förutsätter nyckelpar. En publik och en privat nyckel hos vardera part. Man krypterar med mottagerens publika nyckel och signerar med sin egen privata nyckel. Mottagaren dekrypterar med sin privata nyckel och autentiserar med sändarens publika nyckel.

Bob har en publik och privat nyckel. Han lägger upp den publika på sin webbsida där alla kan se den (även Eve). Alice krypterar ett meddelande med Bobs publika nyckel och skickar meddelandet till Bob. Eve kan inte läsa det krypterade meddelandet, men Bob kan dekryptera med sin privata nyckel.

 Vid symmetrisk kryptering har sändare och mottagare samma krypteringsnyckel. Det går i regel mycket fortare än asymmetrisk kryptering. Standarden idag är AES, tidigare var det DES.

RSA

Krypteringsalgoritmen RSA (döpt efter Ron Rivest, Adi Shamir och Leonard Adleman som först publicerade den) är ett exempel på asymmetrisk kryptering. Du har en *offentlig nyckel* och en *privat nyckel*. Den offentliga nyckeln delar du ut till alla, så att dom kan kryptera meddelanden och skicka till dig, men det är bara du som kan dekryptera med hjälp av den privata nyckeln.

Först måste vi beräkna nycklarna:

* Välj två stora primtal, p och q.
* Beräkna **n** = p\*q.
* Beräkna f = (p-1)\*(q-1).
* Välj **e** så att det inte har några gemensamma faktorer med f (dvs gcd(e,f) ska bli 1).
* Beräkna **d** som den multiplikativa inversen till e modulo f, dvs så att e\*d modulo f = 1.

e och **n** är offentliga, medan **d** är privat. Själva krypteringen går till så att man först konverterar meddelandet till ett tal (t ex genom att konkatenera ASCII-koderna). Man delar upp det i lagom stora delar som vi kallar m1, m2 osv.

Sedan beräknar vi mie modulo n för att få de kodade delarna ci.

Den som vill dekryptera måste känna till d, och kan då få ut det ursprungliga meddelandet genom att beräkna cid modulo n för varje ci.

På University of Illinois har man gjort en [demo av RSA (Länkar till en externa sida.)Länkar till en externa sida.](http://logos.cs.uic.edu/340%20Notes/rsa.html), prova gärna!

För att knäcka RSA måste man faktorisera **n**, dvs lista ut vilka de två stora primtalen p och q är. Det finns ännu ingen algoritm som kan göra detta i *polynomisk* tid, dvs O(Lk) där L är längden av primtalet och k är en godtycklig konstant, vilket innebär att beräkningen tar för lång tid för att vara praktiskt användbar. Därför räknar man med att RSA i nuläget är oknäckbar för tillräckligt stora primtal.

Ovannämnda algoritmer är publika och man kan matematiskt verifiera att de fungerar. Det som är hemligt är nycklarna. Att hemlighålla algoritmen (security by obscurity) bör betraktas med skepsis. Under 2013 läckte [Edward Snowden hemligt material (Länkar till en externa sida.)Länkar till en externa sida.](http://www.lawfareblog.com/catalog-of-the-snowden-revelations/) i huvudsak från amerikanska NSA vilket bekräftat bilden av att algoritmerna i huvudsak håller men att det finns mängder med andra säkerhetsproblem.

**Datasäkerhet**

Lösenord

Lösenord bör sparas saltade och hashade.

Exempelvis skulle en lösenordsfunktion:

def testaHemligtPassword( password ):

if password == "123aabbccdd" :

return True

else:

return False

kunna skrivas så här:

def dunkeltPasswordsTest(password):

hårdkodat\_salt = "123£"!#¤!..."

hårdkodat\_saltat\_och\_hashat\_lösen = "312sd3$£$#!SESD..."

hashat\_password = hashfunc( password + hårdkodat\_salt)

if hashat\_password == hårdkodat\_saltat\_och\_hashat\_lösen :

return True

else:

return False

Givet koden är det svårt att gissa lösenordet. Eftersom en hashfunktion kan ge samma värde för två helt olika indata (krock) så går det inte att, givet hashvärdet, lista ut vilket det ursprungliga värdet var.

I mars 2013 anordnade nättidningen Ars Technica en tävling att gissa hashade lösenord.  Bl.a. används befintliga lösenord som algoritmen hakar på suffix och prefix.

Validera indata

Ett användarvänligt program berättar för användaren om indata är på fel format så att hen har chans att rätta till det. Men det finns ytterligare skäl till att validera inmatning innan den skickas vidare till programmet - det är en möjlig säkerhetsrisk.

Kontrollera t ex:

* Indatas längd
* Datatyp
* Filformat (inte bara ändelsen)

Dekompilera kod

Definitionen på ej hackningsbar skulle vara att den som har tillgång till källkoden inte vet mer än den som har den kompilerade koden. För den senare fungerar koden som en svart låda (black box).

**Föreläsning 13: Repetition inför tentan**

* Repetition
  + Datalogi
  + Abstraktion
  + Datastrukturer
  + Algoritmer
* Instuderingstips
* Tentan

Repetition

Detta är en repetitionsföreläsning. Den täcker inte hela kursen, men är en översikt och snabbgenomgång.

Kursen heter *Tillämpad* datalogi. Mängden tillämpningar är obegränsad. All tillämpning bör motiveras - man måste kunna tala om varför man valt att göra på ett visst sätt.

Datalogi

*Vad var datalogi nu igen?*

Datalogi är läran om *datastrukturer* och *algoritmer*, dvs hur man kan organisera och hålla reda på data samt hur dessa data kan utnyttjas enligt en steg-för-steg-beskrivning för att (effektivt) lösa något problem.

Abstraktion

Ytterligare ett centralt begrepp i kursen är *abstraktion*. I Nationalencyklopedin står det så här:

* *Abstrakt*: "En föreställning är abstrakt om den syftar till att fånga det allmängiltiga hos företeelsen i fråga och bortse från eventuella tillfälligheter. Föreställningen om cirkelns begrepp är t.ex. en abstrakt föreställning till skillnad från föreställningen om en enskild cirkel."
* *Abstrakt tänkande*: "*Abstrakt tänkande* är tankeprocesser som grundar sig på abstrakta begrepp och allmänna principer och inte på enskilda föremål eller konkreta företeelser, och där olika begrepp kan sammanställas till nya helheter, i vilka oväsentliga delar utelämnats."

I datalogi:

* I definitionen av en abstrakt datatyp (ADT) anger man vilka operationer som finns (t ex insert(x), exists(x)), dvs man definerar ett gränssnitt.
* Vid konstruktion av algoritmen behöver man inte tänka på implementationen av datastrukturerna.
* Det här gör det lättare för programmerare att samarbeta i ett projekt:
  + Den som skriver ADT:en kan ändra implementationen, så länge allt fungerar likadant.
  + Den som använder ADT:en behöver inte bry sig om hur den är konstruerad, utan behöver bara förstå gränssnittet.
* Förenklar kodåtervinning (tänk på labbarna i kursen!).

*Ett exempel*: en abstrakt ordlista kan tex defineras med ett gränssnitt bestående av två operationer:insert(x), exists(x).

Ni har själva implementerat en sådan datastruktur på två olika sätt

* I labb 3 - med binärt sökträd
* I labb 5 del 2 - med en hashtabell

När ni sedan använde ordlistan i breddenförstsökningen i labb 4 så använde ni den *abstrakt*, utan att reflektera över hur den var implementerad.

Datastrukturer

Datastrukturer används för att lagra och använda data. I kursen har åtminstonde följande datastrukturer tagits upp:

* Små objekt för egendefinierade saker (t ex Noder av olika slag)
* Länkade listor bestående av noder med en next-pekare
* Vektor/array (inbegriper pythons inbyggda lista)
* Stack (implementerad med en länkad lista)
* Kö (implementerad med en länkad lista)
* Allmänna träd (implementerade med noder med två pekare)
* Binära träd (implementerade med noder med två pekare)
* Hashtabeller (implementerade med en vektor)
* Booleska hashtabeller och bloomfilter
* Trappa/heap (implementerad med en vektor som tolkas som ett binärträd)
* Prioritetskön (implementerad med en trappa)

Vi har definerat dessa datastrukturer abstrakt - vi är överens om hur de bör funka. Dessutom har vi implementerat dem. Sedan har vi använt dem på ett abstrakt vis - utan att behöva bry oss om hur de var implementerade. I kursen ingår både och - att förstå hur de funkar och använda dem på ett abstrakt plan.

Algoritmer

Algoritmer används för att lösa problem. En algoritm utnyttjar en eller flera olika typer av datastrukturer och det är rätt datastruktur i kombination med rätt algoritm som gör algoritmen effektiv. I kursen har åtminstonde följande algoritmer tagits upp:

* Sökning, tex:
  + Linjärsökning
  + Binärsökning
  + Grafgenomgång/sökning i graf, tex:
    - Breddenförstsökning
    - Djupetförstsökning
    - Bästaförstsökning
* Rekursiva algoritmer, tex:
  + Listrekursion
  + Trädrekursion
  + Binärträdssökning, -utskrift
  + Rekursiv medåkning för syntaxkontroll
* Sortering, tex:
  + Urvalssortering
  + Insättningssortering
  + Bubbelsortering
  + Räknesortering (Distribution count)
  + Radixsortering (Hålkortssortering)
  + Damernaförstsortering
  + Kvicksortering (Quicksort)
  + Samsortering (Mergesort)
  + Trappsortering (Heapsort)
* Hashning, med tillämpningar:
  + En miljon sånger
  + Data för atomer
  + Bloomfilter
  + för att spara lösenord
  + hashsummor av installationsfiler (t.ex MD5)
* Textsökning, tex:
  + KMP-automat för textsökning
  + Boyer-Moore
  + Rabin-Karp
  + Reguljära uttryck
* Komprimeringsalgoritmer, tex
  + Följdlängdskodning (RLE)
  + Huffmankodning
  + Lempel-Ziv-kodning (LZ), speciellt LZW
* Redundans, korrigering
  + paritetsbit
  + hammingavstånd
  + sista siffran i personnummer
* Krypteringsalgoritmer, tex
  + Transpositionschiffer
  + Caesarchiffer
  + rot13
  + Bokchiffer
  + One-time pad
  + Key exchange
  + RSA

Det finns också en mängd namnlösa småalgoritmer som ingår i de ovanstående. Givetvis är det viktigt att förstå hur ett binärt träd byggs upp innan man kan söka i det, hur en hashtabell eller en boolesk hashtabell fylls i innan sökning kan ske och så vidare. Hur man sätter in något i en datastruktur kan ju också beskrivas med en algoritm!

Algoritmer jämförs genom antalet operationer som måste utföras givet ett antal element eller mer grovt med komplexitetsberäkningar, där komplexiteten anges med en funktion av viss storleksordning, Ordo, *O*(f(N)). Här är en **på intet sätt uttömmande tabell** över några algoritmer och deras tidskomplexitet. Kom ihåg att det inte räcker att kunna dessa utantill - man måste även kunna resonera om varför det är så, och när det inte gäller.

|  |  |
| --- | --- |
| **Komplexitet** | **Algoritmer** |
| O(n2) | enkla sorteringsalgoritmer, quicksort |
| O(n\*log(n)) | mergesort, heapsort, quicksort |
| O(n) | linjärsökning, räknesortering |
| O(log(n)) | binärsökning, sökning och insättning i binärträd |
| O(1) | insättning och sökning i hashtabell |
| 1 | en addition, en multiplikation, en jämförelse |

När man anger ordoklassen behöver man bara ta med den största termen, och kan strunta i multplikation eller division med konstant, t ex är O(nlogn(n) + 155\*log(n) - 1) = O(nlog(n)).

Man mäter komplexitet i enkla operationer (tex: en addition, en multiplikation eller en jämförelse). Vilken operation man mäter beror på vilken typ av algoritm det är frågan om. Exempelvis innehåller all någon form av jämförelse så där är det naturligt att räkna antal jämförelser snarare än aritmetik.

Man måste naturligtvis definera vad man menar med de i uttrycket ingående variablerna och vilka förutsättningar som gäller.

Samma algoritm har olika tidskomplexitet vid olika förutsättningar. Ofta (men inte alltid) är man intresserad av det värsta fallet och de förutsättningar som gör att algoritmen tar längst tid. Här är en tabell över hur lång tid det tar (hur många jämförelser det går åt) för att hitta ett värde i ett binärt sökträd i några olika fall:

|  |  |
| --- | --- |
| **Sökning i balanserat binärträd** | **Tidsåtgång** |
| Om det sökta finns, i bästa fall | 1 |
| Om det sökta finns, i värsta fall | log(n) |
| Om det sökta finns, i snitt | log(n)-1 |
| Om det sökta ej finns | log(n) |

Om man vet att det sökta finns och bara vill konstatera var i trädet det finns behöver man inte kontrollera den sista noden. Då blir värsta fallet: log(n)-1 och snittet ungefär: log(n)-2.

Men om insättningen i det binära trädet gick dåligt ligger alla värden i en tarm och då blir sökningen som i en enkellänkad lista:

|  |  |
| --- | --- |
| **Sökning i enkellänkad lista** | **Tidsåtgång** |
| Om det sökta finns, i bästa fall | 1 |
| Om det sökta finns, i värsta fall | n |
| Om det sökta finns, i snitt | n/2 |
| Om det sökta ej finns | n |

Om man vet att det sökta finns och bara vill konstatera var i listan det finns behöver man inte kontrollera den nedersta noden. Då blir värsta fallet: n-1 och snittet: (n-1)/2.

OBS. Oftast har man ingen aning om huruvida det sökta finns eller inte...

Instuderingstips

För varje datastruktur och algoritm gäller att åtminstonde kunna:

* Förstå
  + Abstrakt: hur använder man den?
  + Konkret: hur implementerar man den? (Kunna beskriva i ord.)
* Analysera
  + Hur snabb/effektiv är den? Tidsomplexitet/minnesåtgång.
  + Vad har den för fördelar/nackdelar? Begränsningar?
  + När är den lämplig/olämplig, jämfört med andra algoritmer/datastrukturer?
* När du tittar på gamla tentor...
  + Jobba mer med sådant du *inte* kan
  + Titta även på tentor utan lösning

Betygskriterier

För betyg E

ska du kunna avgöra vilken algoritm som löser ett givet problem, kunna beskriva algoritmen och demonstrera den steg för steg med givna data, samt implementera den. Motsvarande gäller för datastrukturer.

För betyg C

ska du ha uppfyllt kraven för betyg E, och dessutom ska du kunna jämföra algoritmer och datastrukturer och bedöma dessas lämplighet för ett givet problem. Här ställs också krav på tidsplanering.

För betyg A

ska du ha uppfyllt kraven för betyg C, och du ska dessutom kunna modifiera/kombinera algoritmer och datastrukturer för att lösa nya problem. Här ställs också höga krav på tydlighet i algoritmbeskrivningar.

**Hur beskriver man en algoritm?**

Skilj på att *förklara hur en algoritm fungerar* och att *konstruera en algoritm*.

* Vad är indata till algoritmen?
* Vad är utdata?
* Punktvis (inte löptext). Ordningen är viktig!
* När avslutas algoritmen?

Tentan

Mest problemfrågor, t ex:

* Demonstrera en algoritm för givna data.
* Visa hur en operation i en datastruktur fungerar för givna data.
* Beskriv en algoritm som löser ett givet problem.
* Uppskatta tidskomplexiteten för din algoritm.
* Berätta vilka algoritmer/datastrukturer från kursen som kan användas.
* Resonera om varför en viss algoritmen/datastrukturen är lämplig i sammanhanget.

Men också teorifrågor, t ex:

* Vilken tidskomplexitet har algoritmen xxx?
* Vad är viktigt att tänka på när en använder datastrukturen yyy?

Kom ihåg:

* Viktigt att motivera svaren!
* Viktigt att motiveringar är tydliga.
* *Inte* viktigt att skriva programkod, strukturerad text eller pseudokod går bra.
* Rita gärna, så blir det lättare för oss rättare.

Tillåtna hjälpmedel på tentan är: Se [kursPM](https://kth.instructure.com/courses/7489/pages/kurspm)

**Tentauppgift betyg C: Rekursiv bränsleåtgång**

Bränsleåtgången per antal maskhålshopp som rymdvarelserna gör räknas ut med följande rekursiva metod. En tildastudent påpekar vänligt för rymdvarelserna att implementationen är onödigt ineffektiv.

def fuel(N): # N is nr of wormhole jumps

if N == 0 :

return 3000

if N == 1 :

return 2990

else:

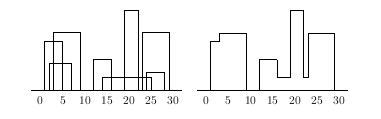
return fuel(N-1) + fuel(N-2) - 3000

Gör följande:

* Ange bränsleåtgången för 6, 5, 4, 3, 2, 1 maskhålshopp
* Förklara vad som är ineffektivt med implementationen.

[Lösning (Länkar till en externa sida.)Länkar till en externa sida.](http://www.csc.kth.se/utbildning/kth/kurser/DD1320/tilda15/forel/fuelLosning.html)

Tentauppgift betyg A: Silhuettproblemet

Om man på håll betraktar en stad i skymningen ser man inte dom enskilda byggnaderna utan bara silhuetten, den yttersta konturen, avteckna sig mot himlen. Hitta på en algoritm som givet varje byggnads kontur beräknar stadssilhuetten.   


Anta att alla byggnader står på x-axeln, är rektangulära och beskrivs av tripplar(left, height, right) där left är vänsterväggens x-koordinat, right är högerväggens x-koordinat och height är höjden (y-koordinat).

Inmatningen består av n stycken tripplar, ordnade i stigande värden på vänsterväggar, och utmatningen ska vara en rad med x,y-koordinater som från vänster till höger beskriver silhuetten.

x-värdena anger var på x-axeln silhuetten går vertikalt och y anger på vilken höjd silhuetten fortsätter efter x-värdet. Den sista y-koordinaten är alltid noll eftersom alla byggnader står på x-axeln.

Exempel: Om n=8 och inmatningen är   
(1,11,5) (2,6,7) (3,13,9) (12,7,16) (14,3,25) (19,18,22) (23,13,29) (24,4,28)   
så blir utmatningen

x= 1 y= 11

x= 3 y= 13

x= 9 y= 0

x= 12 y= 7

x= 16 y= 3

x= 19 y= 18

x= 22 y= 3

x= 23 y= 13

x= 29 y= 0

**Föreläsning 14: Kattis, syntaxträd, tentagenomgång**

* Syntaxträd
* Om testning på Kattis
* Labb 8
* Labb 9
* Labb 10

Tentakomplettering

E-komplettering på fredag 3 november kl 10-12. Bokningstider läggs upp [på vanliga bokningssidan](https://kth.instructure.com/courses/7489/pages/bokning).

Komplettering av C-uppgifter (kan ge D på uppgiften) nästa vecka fredag 10 november kl 10-12.

Du får frågor på det område som du fick Fxpå.

Tentan finns i [tentabanken (Länkar till en externa sida.)Länkar till en externa sida.](https://www.kth.se/social/course/DD1320/page/tentabank-5/).

Laborationerna

I labb 8 och 9 ska du skriva ett program som kollar att molekyler som användaren matar in via tangentbordet följer en given syntax.

I labb 10 ska du bygga vidare på programmet från labb 9 så att det även skapar ett *syntaxträd* (och ritar upp det med hjälp av en färdig grafikmodul, molgrafik.py).

Syntaxträd

När man använder en syntax för att tolka text (indata, programkod etc) brukar man skapa ett *syntaxträd* som datastruktur för den parsade texten.

Ett syntaxträd är ett allmänt träd, där de inre noderna är icke-slutsymboler och löven slutsymboler.

Välkänt exempel:  
*Språket som består av meningar av typen*  
*JAG VET ATT DU TROR ATT JAG VET OCH JAG TROR ATT DU VET ATT JAG TROR har följande syntax:*  
*<Mening> ::= <Sats> | <Sats><Konj><Mening>*  
*<Konj> ::= ATT | OCH*  
*<Sats> ::= <Subj> <Pred>*  
*<Subj> ::= JAG | DU*  
*<Pred> ::= VET | TROR*

Vi kan skapa ett syntaxträd för en given mening genom att parsa meningen och identifiera delarna. Exempel:   
JAG VET  
är en *<Mening>*, som i sin tur är en *<Sats>*, som är *<Subj>* följt av *<Pred>*, där *<Subj>* är JAG och *<Pred>* är VET.

**Uppgift:** Givet följande BNF-grammatik för vanliga taluttryck med operationerna + - \* /.

*<Uttryck> ::= <Term> | <Term> + <Uttryck> | <Term> - <Uttryck>*

*<Term> ::= <Faktor> | <Faktor> \* <Term> | <Faktor> / <Term>*

*<Faktor> ::= TAL | -<Faktor> | (<Uttryck>)*

Rita upp ett syntaxträd för uttrycket 2\*(3+4\*5).

**Lösning:**

|

\*

/ \

2 ( )

|

+

/ \

3 \*

/ \

4 5

Syntaxträdet i labb10 ska representeras av ett allmänt träd där varje nod (motsvarar en ruta i bilden nedan) kan representeras med ett objekt av klassen Ruta nedan:

class Ruta:

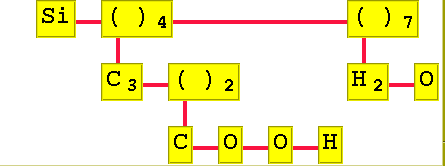
def \_\_init\_\_(self):

self.atom = "( )"

self.num = 1

self.next = None

self.down = None

Molekylen Si(C3(COOH)2)4(H2O)7 får följande utseende:   
  


Sist i laboration 10 ska du också beräkna molekylernas vikt. För att göra det ska du använda hashtabellen du skrev i laboration 7!

Kattis

Ett automatiskt system för rättning av labbar

Kattis började i form av Marvin, som användes för att rätta inlämningarna i programmeringstävlingskursen DD2458 (programmering och problemlösning under press), men används numera även i DD1352 (Algoritmer, Datastrukturer och Komplexitet) och DD2387 (Programsystemkonstruktion med C++) med flera.

Kattis är skrivet i en kombination av Python, PHP och SQL och körs på en Ubuntu-server. En specialbyggd säkerhetslösning används för att hindra fusk. All inskickad kod lagras också för att senare kunna kontrollera koden ifall det skulle behövas.

Den pedagogiska tanken med Kattis är att eleverna skall kunna sända in sin kod och snabbt få svar på huruvida koden fungerar. Kattis skall även ge eleverna övning i att söka efter buggar i koden. Kattis har utvecklats av:

* Fredrik Niemelä
* Gunnar Kreitz
* Per Austrin

Övriga som hjälpt till är:

* Johanna Eriksson
* Mattias de Zalenski
* Pehr Söderman

Hur ska jag göra?

1. Följ länken: [Kattis (Länkar till en externa sida.)Länkar till en externa sida.](https://kth.kattis.scrool.se/)
2. Logga in (längst upp till höger) med ditt KTH-id
3. Välj "COURSES" i övre menyn
4. Välj "tildah17"
5. Klicka på "I am a student taking this course and I want to register for it on Kattis."
6. Välj "Problem list".  
   Här finns flera problem, två testproblem (**hello** och **different**) och labb 9 (**formelkoll2**)
7. Klicka på ett av problemen
8. Välj "SUBMIT" för att komma till inskickningssidan.
9. Fyll i fälten "Problem ID" och "Language"
10. Välj en fil (ditt python-program)
11. Tryck på knappen "Submit"
12. Uppdatera webbsidan för att få se Domarstatus.

Indata

Programmen du skickar till Kattis ska läsa indata från [*stdin* (Länkar till en externa sida.)Länkar till en externa sida.](http://en.wikibooks.org/wiki/Python_Programming/Input_and_output).   
Det fungerar som att läsa från fil! Enkelt exempel:

from sys import stdin

rad = stdin.readline()  
print(rad)

Så här ska du göra i labben:

from sys import stdin

def main():

#stdin = open("indata1.txt") #Lätt att ändra för att testa indata från fil

rad = stdin.readline()

while rad[0] != "#":

q = LinkedQ()

for tkn in rad:

q.put(tkn)

try:

readformel(q)

print("Formeln är syntaktiskt korrekt")

except Syntaxfel as felet:

rest = str(q).strip()

print(felet, "vid radslutet", rest)

rad = stdin.readline()

main()

Utdata

Utdata ska du skriva ut med print som vanligt.

print(x)

Syntaxfel

Definiera ditt eget särfall som en subklass till Exception. Du behöver inte göra egna metoder - allt som finns i Exception ärvs av din klass.   
Exempel (Python 3):

class Syntaxfel(Exception):

pass

def readformel(q):

...

raise Syntaxfel("Felaktig gruppstart")