Datatyper

**Datatyper**

Abstrakt datatyp = Vet vad man kan göra med typen och vad dess

metoder gör utan att se implementationen.

Kö: Först in först ut

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.

Med noder, lägga in:

1. Skapa ny nod med värde x, 2.Ny nod nu efter sist. 3.Ny nod är sist.

Plocka ut ur kön:

1. Returnera first. 2. Den som är efter first är nu first. 3. Returnera

lista.

Bra vid bredden först lösning då man vill ha den snabbaste lösningen.

Först alla barn, sen alla barn barn osv.

Prioritetskö:

Samma som vanlig kö men med enqueue(x,p) med prioritet.

Bästa implikationen är att skapa en heap/trappa som ett träd där

förälder<barnet(min-heap), om inte byt med minsta barnet. Insättning/

Uttagning O(logn)

Stack: Sist in först ut.

push(x) Lägg x överst på stacken.

x = pop() Plocka ut och returnera det som ligger överst.

isEmpty() Undersök om stacken är tom.

En länkad lista är långsammare att läsa för måste gå igenom alla, men

insättning och borttagning går fort. En vektor har index sök, går fort.

Men måste flytta på varje nod när man tar ut eller sätter in något vilket

tar längre tid än länkad lista.

Bra vid djupetförstsökning. Den första lösningen är inte alltid den

kortaste. Börjar med barn rakt ner först innan den kollar på syskon.

**Tidskomplexitet:**

Tidskomplexitet:

O(n2) enkla sorteringsalgoritmer, quicksort

O(n\*log(n)) mergesort, heapsort, quicksort

O(n) l injärsökning, räknesortering

O(log(n)) binärsökning, sökning och insättning i binärträd

O(1) insättning och sökning i hashtabell

1 en addition, en multiplikation, en jämförelse

T=k\*O(n)

**Sökning**

Sökning

Linjärsökning (Sequential search), jämför med alla element I tur och

ordning, O(n), bryt när den hittar det sökta eller listan tar slut.

Binärsökning. I en sorterad lista, tittar I mitten, jämför, m!=key,

beroende på om key >/< mitten, nu mitten I höger eller vänster halva.

O(logN)

**Träd**

Träd

Allmäna träd har en rot, och noder med value, down, right.

**Binärt Träd**

Binärt träd

Binärt träd med noder value, right, left. Varje förälder kan endast ha 2

barn.

A-B/C-DE/FG Vänster◊Höger

Preorder: Roten djupet först: A, B, D, E, C, F, G

Postorder: Bredden, Tillplattad: D, B, E, A, F, C, G

Inorder: Barnet först: D, E, B, F, G, C, A

Grafer

**Grafer**

Kan representeras som en graf, grannmatris, eller grannlista. Kan vara

viktad eller oviktad och riktad eller oriktad.

**Hashning**

Hahning

Att lägga in en nyckel i en lista med länkat värde.

Nykeln görs om till ett tal, tex summa(ord[i]]%mod(1000) =772, om

vi sätter in våra värden på plats 772 för alla nycklar i en lista, även då

listan är osorterad kan vi sedan söka med O(1) , 1 st jämförelser för

sökta nyckeln blir samma hashvärde och behöver bara göra en

jämförelse, förutsatt att det inte skett några krockar, i värsta fall om

alla är dubbletter blir sökning O(n)

Boolesk hashtabell: Endast True/False lagras. För att t.ex. ange

förekomst av ord i ordlista. Kan inte hantera krockar.. Sannolikhet för

fel vid 50 % ettor är 50 %.

Bloomfilter: 14 stycken booleska hashtabeller med olika

hashfunktioner. Godkänn om det är True på alla 14 ställen.

Sannolikhet för fel blir (1/2)14 = 0.006 %. Har två operationer i

datatypen: insert(x) – stoppa in värde. Och isIn(x) – kolla om det

förekommer. O(1)-O(n)

**Sortering**

Sortering

Sökning tar O(N) I en osorterad lista, men efter sortering kan man

istället använda binärsökning.

Urvalssortering: (Selection sort)Sök igenom efter minsta tal. Flytta till

första

position. Sök efter nästförsta. Flytta till andra position. n(n-1)/2

jämförelser => O(n2)

Bubble sort: Byt första och andra om de är fel ordning. Byt andra och

tredje om de står i fel ordning. Osv. Gå igenom gång på gång tills inga

byten sker. Fördel att bubbelsortering faktiskt kan sluta i förtid →

lämpad för sortering av halv-sorterad

data. Värsta fallet n(n-1)/2 jämförelser => O(n2) Om listan nästan är

sorterad brukar denna vara snabbare än urvalsort.

Insättningssortering: Jämför med tidigare värden i lista, om det är

mindre så gör vi plats genom att flytta tidigare ett snäpp höger. Flytta

så mycket som behövs. Stoppa

in nya värdet. Börja om med nästa värde. Lämpad för då

man får värde en efter en inläsning från fil. Och för att stoppa in i

redan sorterad lista. Flytt snabbare än byte så

bättre än urvalssortering. O(n2)

Damerna först: 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!

Quicksort: Som damerna först, men man väljer nyckelvärden

(dam/herre). 1) Bestäm vilka värden som ska kallas damer(små). 2)

Partitionera listan så att damerna kommer först. 3) Sortera varje

segment för sej. O(nlogn) → listan kan delas på mitten logn gånger.

Blir väldigt långsam om listan redan är sorterad så man kan välja

median of 3 och välja första, andra och en i mitten som dam för att få många splittar.

Merge Sort: Snabb men minneskrävande metod. O(nlogn) 1) Dela

listan i två hälften så långa listor. 2) Sortera varje halva för sej. 3)

Samsortera till ursprungliga listan. (QS och MS är divide and conquer)

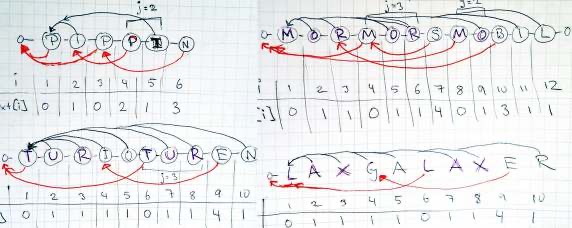
Räknesortering: (Distribution count)Om man skall sortera på ett mindre antal nyckelvärden, krävs då att man innan vet att det finns ett begränsat antal nycklar. 1) Läs igenom filen och räkna hur många det finns av varje nyckelvärde. 2) Dela in listan i lagom stora segment för denna distribution. 3) Läs filen i gen och lägg in varje värde i  
sitt segment. || Krävs att man hämtar in data från en annan  
lista eller fil. O(n)  
Radixsortering: Om man upprepar förfarandet (RS) för varje position (siffra eller bokstav) i de data som ska sorteras får man *radixsortering*! Upprepad räknesortering som går igenom varje position så att det blir fullständigt sorterat. O(wn) där w är längden och n är antal datavärden)  
**Heapsort:** Man stoppar in n tal i en heap och plockar ut dem en för en→ sorterade.O(nlogn) (vi måste stoppa in och ta bort n gånger). Quicksort är lite snabbare, men heapen har inte samma värstafall. Heap kan användas till annat också.  
Min-Heap Binärt träd, elementet på första nivån ska vara det minsta värdet(roten). Varje förälder<barnet. Skapa en lista där plats 0 är tom, lägg sedan i noderna i A B C D E F G ordning(se Binärt träd) . Insättning: Sätt i det nya elementet på nästa lediga plats på nivån. Jämför med föräldern, om förälder är större, byt plats. Fortsätt tills det nya elementet är större än sin förälder eller tills den kommer till plats 1. Uttagning: Ta ut roten, sätt in det sista elementet som ny rot. Så länge den nya roten är större än sina barn, byt med det minsta barnet. Både insättning och utsättning är O(logn). Max-Heap är då förälder>barnet.

**Automater och textsökning**

Vissa inmatningar leder till vissa tillstånd, t.ex portkod. Om koden är DEG, ger D tillstånd 2, E, tillstånd 3, och G tillstånd 4, och endast då låses porten upp i den rätta inmatade ordning, annars tillbaka till tidigare tillstånd t.ex DD, DED.

**KMP-Automat**

Då man vill söka efter ett enskilt ord i en bok. 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.



def finns(p, value): #Kollar om det finns i trädet

if p == None: #Tomt träd

return False

if value == p.value: #Den finns

return True

if value < p.value: #Går neråt till vänster

return finns(p.left, value)

if value > p.value: #Går neråt till höger

return finns(p.right, value)

inorder skriv(b.left), print(b.value),, skriv(b.right)

def makechildren(parent, q, svenska, dumdum):

for i in range(len(parent.word)):

for letter in letterList:

temp = parent.word.replace(parent.word[i], letter)

if temp in dumdum:

pass

elif temp in svenska:

dumdum.put(temp)

child = ParentNode(temp, parent)

q.enqueue(child)

def findWay(svenska):

q = LinkedQ()

dumdum = Bintree()

parent = ParentNode(start)

q.enqueue(parent)

dumdum.put(start)

while not q.isEmpty():

current = q.dequeue()

if current.word == end:

current.writechain()

break

else:

makechildren(current, q, svenska, dumdum)

if q.isEmpty():

print("Det finns ingen väg från %s till %s"

% (start, end))

Kryptering

**Kryptering**

Transpositionsschiffer

Sätt in hemlis i matris rad för rad med hemlig n\*m, skriv ut kolumner

som en sträng.

Ceasar chiffer Byt mot annan bokstav, lätt knäckt med statistik

Bokchiffer Samma bok, hemlis=sida i bok,ord på sidan ◊ (314 761)

One-time-pad

Varsin nyckel med binära siffor lika långt som meddelandet, gör xor

för varje bokstav, kasta en sida efter and. Oknäckbar. Lika=0

Hamming pad: xor med alla nycklar, lägsta summa vinner.

Säker nyckelöverföring

A:A-lås ->B: A+B-lås ->A: B-lås ->B:Öppen

Asymmetriskt kryptering

Var sitt nyckelpar. Symmteriskt samma nyckelpar

**RSA**

RSA

En privat nyckel, en publik nyckel.

Nyckelgenerering:

1. Välj 2 st stora primtal, ex p=2, q=7

2. N= p\*q=14 (Public key)

3. Φ(N)=(p-1)(q-1)= 6 (co-prime)

4. Nyckelpar A= mellan 1-6, co-prime (6,14)=5 -> Nyckel A=(5,14)

5. Nyckel B= 5\*B mod(6)=1 (flera), ex: 11 -> Nyckel B= (11,14)

Encrypt ”B”=2->25mod(14)=4-> ”D”

Decrypt: ”D”=4->411mod(14)-> 2=”B”

*Polynomisk* tid, dvs O(Lk) där L är längden av primtalet och k är en

polynomisk tid, dvs O) där L är längden av primtalet och k är en

godtycklig constant oknäckbar för stora primtal

Linked Q-LAB

def enqueue(self, inData):

x = Node(inData)

#Om kön är tom

if self.isEmpty():

self.first = x

self.last = x

x.next = None

#Standardfallet

else:

self.last.next = x

self.last = x

def dequeue(self):

utData = self.first

if self.first == self.first.next:

self.first = None

else:

self.first = self.first.next

return utData.value

Binärträd-LAB

def putta(root, newvalue):

if root == None: #Vid tomt träd

return Node(newvalue)

if newvalue < root.value: #Om nya är före root => lägger till höger

if root.left == None: #Om inget till vänster

root.left = Node(newvalue)

else

putta(root.left, newvalue)

if newvalue > root.value: #Om nya är efter root => lägger till höger

if root.right == None: #Om inget till höger

root.right = Node(newvalue)

else:

putta(root.right, newvalue)

return root

**Boyer-Moore**

Då hela texten vi söker efter är m tecken lång finns i en lista  
som är *n* lång *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.  
Lägger först den sökta strängen i början på listan, jämför med sista bokstaven, om den bokstaven inte alls förekommer i strängen hoppar den m steg till höger. Om den matchar men inte alla tecken bak, hoppa till nästa förekomst av den kombinationen.

**Rabin-Karp**

Kollar på de m första orden i den sökta listan, räknar ut hashvärdet, om hashvärdet inte är lika med hashvärdet för det sökta ordet, gå ett steg till höger. Komplexiteten blir  
*O(nm)* i värsta fallet, men i praktiken bara *O(n+m)*.

**Reguljära uttryck**

söka efter lab1, Lab2, eller labb3, [Ll]abb?[1-7] \*0 eller fler, +1 eller fler, ? 1 eller 0,[a-b,A-b]alla mellan, [ab] a eller b, [^ a] alla utom a, (grupp).

**Syntax**

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

<Konj> ::= ATT | OCH  
<Sats> ::= <Subj> <Pred>  
<Subj> ::= JAG | DU

<Pred> ::= VET | TROR  
Jag vet att du tror att du och jag etc..  
<rapport> ::= <väder> **.** <temperatur> **.**<väder> ::= <vind> <moln> <nederbörd>  
<vind> ::= | <riktning> VIND,  
<riktning> ::= NORDLIG | OSTLIG | SYDLIG | VÄSTLIG  
<moln> ::= SOLIGT | MULET  
<nederbörd> ::=|OCH REGN|OCH SNÖFALL|REGN OCH SNÖFALL

<temperatur>::= NOLLGRADIGT|grader>PLUSGRADERgrader>MINUSGRADER <grader> ::= <tal>-<tal>  
<tal> ::= 1 | 2 | ...  
NORDLIG VIND, MULET OCH SNÖFALL. 5-10  
MINUSGRADER. SYDLIG VIND, MULET, REGN OCH SNÖFALL. 1-6 MINUSGRADER. SOLIGT. 20-23 PLUSGRADER. **Komprimering**Följdlängdskodning – RLE  
ÅÅÅH! JAAAA!◊3ÅH! J4A! alt §3ÅH! J§4A! (om num i text) Huffmankodning  
Gör binärt träd med frekvens, störst till lägst, 0 på lägst sidan.  
LZW  
NÄSSNUVSNORSNOK.  
1)Läs in N. Finns ej med. Skriv ut N. Skapa kod 0.  
2)Läs in Ä. Finns ej med. Skriv ut Ä. Skapa kod 1.  
3)Läs in S. Finns ej med. Skriv ut S. Skapa kod 2.  
4)Läs in S. Finns med. Läs in N, SN finns ej med. Skriv ut kod  
en för S och skapa kod 3 för SN.

CODE 0 1 2 3 4 5 6 7 8  
Sträng N Ä S SN U V SNO R OK -> NÄS2NUV3OR6K

Föreslå en datastruktur som håller reda på förkunskapskrav för alla kurser.

a) Vilken datastruktur väljer du? Motivera ditt svar. b) Beskriv hur din datastruktur ska implementeras.

Många olika lösningar möjliga, t ex

 Graf implementerad med grannistor

 Allmänt träd för förkunskapskrav per kurs (noder med pekare) + lista med

  kursnoder.

  Hashtabell där noderna har referens till förkunskapskursernas noder

Man ska enkelt kunna hitta en kurs i datastrukturen, och därifrån gå vidare till förkunskapskravskurserna, och sedan vidare till deras förkunskapskrav osv.

För var och en av datastrukturerna (vektor/heap) ska du:  
• Beskriva i detalj hur datastrukturen ska användas i algoritmen för att bygga  
Huffmanträdet.  
• Uppskatta tidskomplexiteten för att bygga Huffmanträdet med hjälp av den datastrukturen.

Vektor:

1. Sortera noderna i avtagande frekvensordning

med Quicksort O(nlogn)

eller Insättningssortering O(n²)

2. Så länge det finns noder kvar i vektorn: plocka ut de två lägsta och ersätt med ny nod O(n) per varv = > O(n²) totalt

Heap:

1. Stoppa in n noder i heapen O(nlogn)

2. Så länge det finns noder kvar i heapen:

plocka ut de två lägsta och ersätt med ny nod 2\*logn + logn per varv => O(nlogn) totalt

Se​ ​ovan​ ​till​ ​vänster.​ ​Hepvillkoret​ ​måste​ ​vara​ ​uppfyllt,​ ​dvs​ ​föräldern​ ​större​ ​än barnen​ ​(eller​ ​tvärtom​ ​för​ ​min-heap),​ ​för​ ​varje​ ​nod.

Se​ ​ovan​ ​till​ ​höger.​ ​Mindre​ ​sorteras​ ​till​ ​vänster,​ ​större​ ​till​ ​höger,​ ​gäller​ ​för​ ​varje nod. Struktur​ ​och​ ​ordning,​ ​t​ ​ex​ ​något​ ​av​ ​följande:

En​ ​heap​ ​måste​ ​vara​ ​komplett,​ ​det​ ​måste​ ​inte​ ​ett​ ​binärtäd

I​ ​en​ ​heap​ ​måste​ ​heapvillkoret​ ​vara​ ​uppfyllt​ ​för​ ​varje​ ​nod,​ ​binärträdet har​ ​mindre​ ​värden​ ​till​ ​vänster​ ​och​ ​större​ ​till​ ​höger.

En​ ​heap​ ​lagras​ ​i​ ​en​ ​array,​ ​ett​ ​binärträd​ ​har​ ​noder​ ​med​ ​left-​ ​och right-pekare

Även​ ​praktiska​ ​experiment​ ​duger​ ​som​ ​svar,​ ​t​ ​ex​ ​“prova​ ​att​ ​stoppa​ ​in ett​ ​nytt​ ​element”,​ ​“skriv​ ​ut​ ​trädet”

Snabb​ ​sökning. (binärträd)

Sortering​ ​(heapsort)​ ​eller​ ​prioritetskö​ ​eller​ ​bästaförstsökning (heap)

Vid linjär probning sparas både key och value efter varandra. Om nyckel man söker efter inte finns på hashindex så söker man linjärt efterföljande element tills man träffar på en tom plats. Då antas elementet inte finnas i hashvektorn.

a)  När man tar bort ett element skapas ett hål. Sökning kommer att stanna här och direkt efterföljande element inte genomsökas (de försvinner)

b)  Man måste lägga dit ett element som markerar att det varit något här

c)  Det är samma problem

d)  Krocklistor har inte samma problem. Borttagning är enkelt att göra. Det blir inget hål i listan

Det finns fler än tre aspekter. Här är några förslag:

Att använda xor är snabbt. Mycket snabbare än RSA.

xor-nyckeln är en engångsnycker och behöver genereras om för varje meddelande. RSA nycklarna behöver inte genereras om.

Engångsnycklarna måste distribueras på ett säkert sätt. Den publika RSA nyckeln är lättare att distribuera eftersom den innehållet inte är hemligt.

xor-nyckeln garanterar inte autentisering, om någon kommit över nyckeln kan man skicka ett falskt meddelande (men endast ett ty engångsnyckel). RSA medger både autentisering och kryptering.

**Kryptering med bokchiffer**

3 jämförelsekriterier efterfrågas, olika utfall av jämförelserna kan godkännas beroende motivering. a) Tre jämförelser mellan bokchiffer med digital respektive tryckt bok.

1. i)  **Kryptering**​ - bokchiffer är starkt krypterat. **L**​ **ika starkt**​ digitalt och i tryckt format
2. ii)  **Nyckelutdelning**​ - Kan tänkas vara både lika svårt att lösa digitalt och tryckt bok.

Det går att argumentera för att det är ​**både**​ enklare ​**eller**​ svårare att distribuera en exakt likadan digital bok/pdf än en tryckt. T.ex. kan vattenmärkning av en pdf tillföra tecken eller epub/mobi-format vara olika på olika plattformar.

1. iii)  **Prestanda**​ - Det går ​**snabbare**​ att dekryptera/krypterar en digital bok med ett datorprogram.

b) Bokchiffer vs RSA:

1. i)  **Kryptering**​ - i princip **l**​ **ika stark**​ som RSA. Dåligt indata kan göra det knäckbart.
2. ii)  **Nyckeludelning/autentisering**​ - RSA löser detta ​**bättre**​ än bokchiffer. Den publika nyckeln i RSA behöver inte vara hemlig och kan enklare delas ut.
3. iii)  **Prestanda**​ - Digitalt bokchiffer är potentiellt snabbare än RSA men det är OK det går att argumentera för båda. Man behöver inte motivera skillnad i digital snabbhet. Bokchiffer med tryckt bok är ​**väsentligen långsammare**​.

Det finns andra jämförelsekriterier och jämförelseresonemang som godkänts, t.ex.

1. iv)  Implementation - enklare att implementera bokchiffer, i synnerhet digitalt
2. v)  Autentisering - kan lösas med en tillfredsställande nyckelutdelning. Bara den som

har nyckeln/boken kan skicka meddelanden och man kan föra liknande

jämförelseargumentation som för nyckelutdelningen

1. vi)  Säker förvaring - större risk för intrång i datorn med digital bok. Det kan finnas spår, digitalt eller fysiskt.

Bloomfilter Ett antal viruspatienter har satts i karantän. Karantänen för den enskilde varar som mest i 14 dagar och man är beredd på en hög omsättning patienter. Det finns ett maxtak på hur många patienter man kan husera samtidigt. Man vet inte hur länge epidemin varar. Viruspatienterna behöver tillfälliga personliga lösenord. För att inte lag- ra lösenorden i klartext så tänker man använda bloomfilter. Valet står mellan att lagra lösenorden i ett bloomfilter eller hashade och saltade. Jämför dessa två alternativ. Var noga med att motivera dina slutsatser med hänvisning till de givna förutsättningarna.

Man kan inte ta bort ett lösenord från bloomfiltret. Nollställer man ett ords 14 ettor så påverkar man andra ord som använder samma ettor. Det går inte heller att bygga om bloomfiltret eftersom lösenorden inte finns i klartext någon annanstans. Den höga omsättningen kommer leder till vektorn fylls utav ettor och mängden false positives kommer att öka. Det går inte att utnyttja att det finns ett maxtak på antal samtidiga patienter eftersom man inte kan ta bort de patienter som skrivs ut. Om man bara lagrar lösenorden i bloomfiltret finns dessutom ingen koppling mellan användare och lösenord så man kan logga in med andra användares lösenord. Visserligen lagras inte lösenorden i klartext men sammantaget är bloomfilter ingen bra lösning givet förutsättningarna.

Om man lagrar patient och lösenord hashade och saltade på fil så lagras de inte i klartext. Det går det att ta bort patienter varefter de skrivs ut oavsett omsättning och man kan utnyttja att det bara är ett visst antal samtidiga patienter inlagda.

Redovisa fördelar och nackdelar med att använda hashtabell respektive binärt sökträd när man ska lagra ett okänt antal nycklar/värden. Var tydlig med dina jämförelsekriterier.

För betyg D så ska man ta hänsyn till att det är ​**ett okänt antal nycklar**​ och nämna minst en av

* ●  Binärträd kan växa obehindrat
* ●  Om hashtabellen inte omdimensioneras -> löjligt långa krocklistor

○ eller skriva att dimensionera hashtabellen kräver omhashning av alla element.

**Dessutom**​ ska man för betyg D jämföra prestanda i det allmänna fallet ● Binärträd O(logN) vs Hashtabell O(1)

För C så ska man uppfylla D och nämna alla tre ovan eller 2 + en annan bra jämförelse t.ex. risk för träd att bli obalanserad med sorterat indata.

En bild som visar bord

Automatiskt genererad beskrivningEn bild som visar bord

Automatiskt genererad beskrivning