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:

O(n2)	enkla sorteringsalgoritmer, quicksort
O(n*log(n))	mergesort, heapsort, quicksort
O(n) 1	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

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

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

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

Kan representeras som en graf, grannmatris, eller grannlista. Kan vara viktad eller oviktad och riktad eller oriktad.

Hashning

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

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 => $O(n^2)$

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 => $O(n^2)$ 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. $O(n^2)$

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.

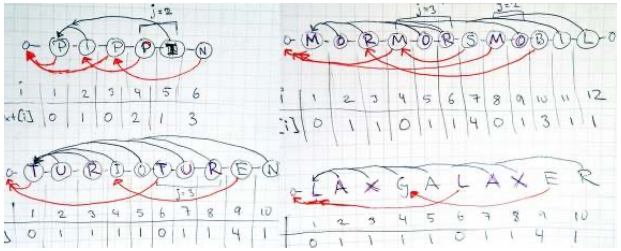
Min-Heap Binärt träd, elementet på första nivån ska vara det minsta värdet(roten). Varje förälder<barnet. Skapa en lista där plats 0 är tom, lägg sedan i noderna i A B C D E F G ordning(se Binärt träd) .
Insättning: Sätt i det nya elementet på nästa lediga plats på nivån. Jämför med föräldern, om förälder 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.



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

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
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->2^5mod(14)=4-> ”D”
Decrypt: ”D”=4->4^{11}mod(14)-> 2=”B”
Polynomisk tid, dvs $O(L^k)$ 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
```

```
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))
```

indata: x (mutbudget)
n mutkolvar (en vektor med de n tjänstemännen som ska mutas)
utdata: Ja om det gick att muta alla n med totalt x kronor, Nej annars

1. Upprepa punkt 2 - 4 nedan för varje person p i listan **mutkolvar**:
2. Hitta ett intervall [**minmuta**, **maxmuta**] mellan vilka mutgränsen ligger.

a) Börja med **muta** = **x**.
b) Om muta accepteras (dvs **p.acceptera(muta)**):

Halvera **muta** tills den inte längre accepteras
c) Intervallet blir då mellan **minmuta = muta** och **maxmuta = muta*2**

3. Nu gör vi binärsökning mellan minmuta och maxmuta för att hitta den exakta mutgränsen.

a) Sätt **muta = mittpunkten** i intervallet b) Om muta accepteras:

sätt **maxmuta = muta** annars:

sätt **minmuta = muta + 1**
c) Är intervallets längd är större än 1? Upprepa a) och b) d)
...annars vet vi att maxmuta är p:s mutgräns

4. Minska budgeterade värdet **x = x - mutgräns** Om x < 0:

så har vi överskridit budget, svaret är **Nej**. Annars

fortsätt från punkt 1 med nästa person
5. Har vi kommit igenom alla personer så är svaret **Ja**.

Binärsökning bland x tal => log(x) Totalt n varv => en faktor n
Svar: O(n*log(x))

Ingående variabler: n (antal tjänstemän), x(mutornas storlek)

För betyg A krävs att din algoritmbeskrivning är tydlig och välstrukturerad.

O(N) är krav B löst en uppgift rätt A två uppgifter rätt samt avbryter när det är fel eller reflekterar att det borde avbryta innan O(N) när det är fel

b) För att övertyga sig om att trädet är korrekt och balanserat måste man gå igenom alla noder d.v.s. O(N). För betyg A behöver man inse det. O(log N) är fel. O(N^2) är för oeffektivt.

a) Det finns flera olika lösningar. Man kan dela upp problemet ett i två delar. För att lösa om det är ett korrekt binärt träd kan man t.ex. tänka sig en rekursiv lösning där man skickar med ett intervall som noden måste hålla sig inom. Den rekursiva tanken är att jag kollar om min egen nod är inom intervallet.

I ett specifikt anrop så kollar jag om min nods värde är inom intervallet. Om inte kastar jag ett undantag. I annat fall så anropar jag rekursivt till vänster med minvärdet jag fått och min egen nods värde som maxvärde. På samma sätt anropar jag rekursivt till höger med min nod som minsta värde och skickar med maxvärdet jag fått. Är noden None gör jag ingenting.

Det räcker inte att kolla om min nod är korrekt i förhållande till mina barns värden vilket man kan se i det givna exemplet på tentatalet.

```
      5
   4
3   7
```

Ett annat alternativ till lösning är att gå igenom trädet inorder och ser om de påträffade värdena är sorterade. Det räcker med att spara och jämföra med det senast besökta värdet, man behöver inte spara alla värden i en lista som man sedan går igenom. Koden blir lite enklare och skulle kunna se ut ungefär så här

```
class Remember:
     def check(value):
         if value < self.previous_value:
             raise "WRONG TREE"
         self.previous_value = value
```

```
def isCorrect(p, remember)
    if p != None
       isCorrect(p.left)
       remember.check(p.value)
       isCorrect(p.right)
```

Att kolla om trädet är balanserat kan man också göra rekursivt men det är svårt att få det rätt. En naiv tanke är att att jämföra maxhöjden till vänster och till höger.

```
def checkBalance1(p):
   r = height(p.right, 1)
   l = height(p.left, 1)
   if abs ( l - r ) > 1 :
      raise "NOT BALANCED"

def height(p, h):
   if p == None:
      return h
   else
      return max(height(p.right, h+1), height(p.left, h+1))
```

Det blir dock fel svar. Ovanstående kommer att godkänna ett träd som ser ut som ett stort lambda.

```
     5
   3   6
 2       8
1          9
```

Man måste i varje deträd avgöra om det är balanserat. En kollosalt oeffektiv lösning är

```
def isBalanced2(p)
    if p != None:
    return isBalanced2(p.left) and
             isBalanced2(p.right) and
             abs(height(p.left) - height(p.right)) <= 1)
```

Där man traverserar trädet flera gånger.
Ett lite bättre alternativ som fortfarande har problem

```
def almostGoodEnough(p, height):
   if p == None:
       return height
   else:
        height += 1
        l_height = almostGoodEnough(p.left, height)
        r_height = almostGoodEnough(p.right, height)
        if abs ( l_height - r_height ) > 1 :
           raise "NOT BALANCED"
```

```
       return max( l_height, r_height )
```

Vi ska konstruera en algoritm för att lägga till så få bitar som möjligt till kodorden, så att Hammingavståndet för kodorden blir d.

Vilken typ av problem är det här?
Vi kan modifiera kodorden på olika sätt och får då en massa olika förslag på koduppsättningar. Grafproblem - startnoden är den ursprungliga koduppsättningen, ett steg bort i grafen har vi noder där koduppsättningen modifierats med en bit.
Vi söker en koduppsättning som uppfyller ett visst villkor, men inte vilken som helst. *Så få bitar som möjligt* ska läggas till. Det här går att göra med breddenförstsökning!

Vi låter en nod innehålla en hel uppsättning med n stycken m-bitars kodord. Ett exempel på en koduppsättning för n=3 och m=4 är:

```
1001
1101
1111
```

Barnen skapas genom att vi gör alla möjliga förlängningar med en bit. Här är barnen på första nivån för exemplet ovan:

10010	10010	10010	10010	10011	10011	10010	10011
11010	11010	11011	11011	11010	11010	11011	11011
11110	11111	11110	11111	11110	11111	11111	11111

För att se till att vi får med alla kombinationer på ett strukturerat sätt har vi lagt till de binära talen 000, 001, 010, 011 osv.

Algoritm:

1. Skapa en lista `bin` med de binära talen från 0 upp till 2^n-1.
2. Skapa en tom kö.
3. Skapa ursprungsnoden. Den ska innehålla en lista `koder` med de n ursprungliga kodorden.
4. Lägg ursprungsnoden i kön.
5. Upprepa följande tills Hamming(koder) == d, eller kön är tom:
   - Ta ut första noden ur kön.
   - Gå igenom listan `bin`. För varje binärt tal:
     - Skapa en ny nod genom att lägga till de binära siffrorna till `koder`
     - Lägg in den nya noden i kön.

Datastrukturer:

- Noder med en lista `koder` där aktuella uppsättningen kodord lagras.
- *Inga* föräldra-pekare. Om vi hittar en nod som uppfyller kravet Hamming(koder) == d så är `koder` lösningen.
- *Ingen* dumbarnstruktur. Algoritmen ger inga identiska noder som behöver sållas bort.

Det behövs tre nästlade for-loopar för att gå igenom alla stjärnor och jämföra deras inbördes avstånd.

```
for a in stars:
  for b in stars:
    for c in stars:
      if distance(a,b) == distance(a,c) == distance(b,c):
        add_liksidig(a,b,c)
```

Algoritmen är O(N^3). För 100000 = 10^5 stjärnor så blir det 10^{15}

En annan lösning är att spara undan alla distanser. Det tar O(N^2) att gå igenom alla stjärnor och ta reda på inbördes distanser. Det finns som mest N^2 olika distanser. Vi kan spara undan i en distanstabell med stjärnpar och om det blir mer än två stjärnpar spara undan distansen för en senare analys. Beroende på fördelningen av stjärnavstånd skulle en sådan lösning kunna vara bättre än O(N^3).

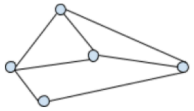
Det finns flera sätt att läsa in data på. Man behöver en datastruktur för grafen och man behöver veta hur många attraktioner det finns. En datastruktur som fungerar är t.ex. en dictionary med attraktionerna som nycklar och en lista med andra attraktioner som value.

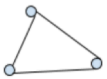
Datastrukturen ska beskrivas med exempeldata.

Ingång	Hoppet	Åksjukan	Godishjulet
Hoppet	Ingång	Åksjukan	Mardrömstunneln
Åksjukan	Ingång	Hoppet	Mardrömstunneln
Mardrömstunneln	Hoppet	Åksjukan	Godishjulet
Godishjulet	Ingång	Mardrömstunneln	
Musfällan	Bergbanan	Vattenfallet	
Bergbanan	Musfällan	Vattenfallet	
Vattenfallet	Musfällan	Bergbanan	

Algoritmen blir en fullständig grafgenomgång antingen med bredden först eller djupet först. Förutom de vanliga stegen där redan-tidigare-besökta noder beaktas så måste man räkna eller markera de attraktioner man behandlat. Efter grafgenomgången jämför man med förväntat antal attraktioner för att avgöra om man når alla.

Uppgiften ger höjd för att läsa in alla attraktioner i en grafstruktur. Men, man måste på något sätt kunna jämföra med förväntat antal attraktioner. Attraktionerna kan ligga i två disjunkta grafer varav bara den ena gås igenom av algoritmen.





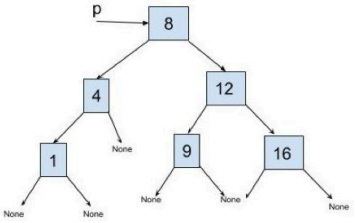
a) Konstruera en effektiv algoritm som givet (1) en pekare till ett binärt sökträd och (2) en sökt summa, returnerar två nodpekare. Summan av värdena i de två noderna ska bli den sökta summan. Om det inte går att hitta två sådana noder så ska *None, None* returneras.

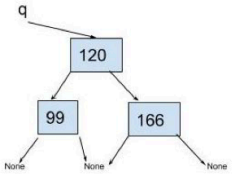
Algoritmen och datastrukturer ska vara tydligt beskrivna.

b) Ange komplexitet för din algoritm.

c) Visa hur din algoritm fungerar om man söker efter summan 10 i trädet p nedan.

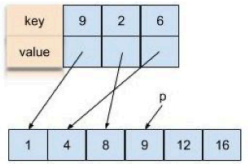
d) Visa hur din algoritm fungerar om man söker efter summan 10 i trädet q nedan.



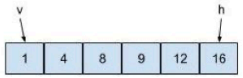


Det går att lösa problemet i O(N) för betyg A. Om man går igenom trädet inorder kan man bygga upp en sorterad lista/vektor istället vilket kan vara enklare för att förklara algoritmen.

Låt en pekare p gå igenom listan, för varje nod kolla om dess värde finns i en dictionary/hashtabell. Om inte, lägg in (summan - värde) som key och pekaren till noden som value. Första elementet 1 finns inte i hashtabellen så lägg in 9 som pekar på noden med 1.



Efter tre varv ligger 9, 6 och 2 i hashtabellen och då är man framme på noden med värdet nio. Eftersom 9 finns i hashtabellen så är man klar och returnerar p och pekaren till 1. Man kan göra samma algoritm genom att gå igenom trädet direkt i in-, pre- eller postorder (se suboptimeringar) och lägga in de matchande talen i dictionary allt eftersom.



En annan lösning i O(N) är att använda två pekare h, v som börjar i varsin ände. Låt högerpekaren stega sig inåt så länge som summan är större än den sökta. När summan är mindre stegar man vänsterpekaren. Genom att stega varsamt ömsom högerpekare och vänsterpekaren i små steg fram och tillbaka kan man ringa in den sökta summan i O(N). Om man inte är varsam kan det resultera i en kvadratiskt lösning där man för varje högerpekare stegar igenom alla vänsterpekare. Det är mycket lättare att beskriva traverseringen för den här algoritmlösningen på en lista/vektor än på ett träd.

Det går också att lösa problemet i O(NlogN) för betyg B. Man traverserar trädet i in- pre- eller postorder och binärsöker i trädet efter den sökta summan minus nodens värde.

Det går att göra en del suboptimeringar som att t.ex. inte undersöka tal som är betydligt större än den sökta summan. Det är lätt att göra fel och kapa trädet utan att undersöka vänsternoder som skulle kunna vara inom intervallet. Om det finns negativa tal i trädet så behöver man veta trädets minsta tal för att avgöra intervallet man letar inom. Då kan det vara bra att gå igenom trädet inorder och börja med minsta noden.

9. Lukes pappas problem

a) Algoritm

Indata: Darth:s startpunkt p_{start}

Rebellernas gömställe p_{slut}

Matrisen *connected* som anger vilka punkter som är förbundna med maskhål.

Utdata: Utskrift av en lista med punkter p_{start}...p_{slut} som ger kortaste vägen

1. Skpa en datastruktur *used* för använda punkter (se b)
2. Skapa en kö
3. Lägg in en nod p_{start} med förälder None i kön
4. Markera p_{start} som True i *used*
5. Plocka ut en punkt **p** ur kön (med dequeue)
6. Gå igenom varje matriselement på p:s rad i matrisen *connected*:
   a. Om p_i inte redan är använd och om *connected*[**p**,p_i] är True:
      i. Lägg in en nod p_i med **p** som förälder kön
      ii. Lägg till p_i till datastrukturen med använda punkter
      iii. Avbryt om $p_i = p_{slut}$, skriv ut lösningen (enligt E-uppgift 2)
7. Upprepa från punkt 4 så länge kön inte är tom
8. Om kön blev tom, berätta för Darth att ingen väg finns (om du vågar)

b) Datastrukturer

A. Boolesk lista *used* med n platser, använd[i] sätts till True i 5.a.ii
B. Kö med noder
C. Noder med en punkt och en pappapekare
D. (och den givna matrisen connected)

c) Demonstrera hur algoritmen fungerar

d) Tidskomplexitet

Problemträdet blir aldrig större än de n punkterna

För varje punkt går vi igenom n-1 matriselement

Att kontrollera dumbarn beror på vald datastruktur, men max n

Totalt $n*(n-1 + n)$ dvs $O(n^2)$