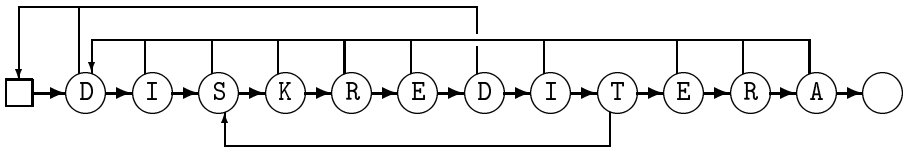
KTH, NADA, Linda Kann

2D1320, Tilda, Lösningsförslag 22 augusti 2002

(5p) **1.** *Diskreditering*



i	next[i]
1	0
2	1
3	1
4	1
5	1
6	1
7	0
8	1
9	3
10	1
11	1
12	1

2. *Diskussionsgrupper*

(5p) Det räcker med två referenser, down och next, i varje objekt,
även om det finns flera kommentarer. Första kommentaren till ett inlägg får man om man följer down och från den kommentaren kommer man åt övriga via next. Se bilden på föreläsningen om träd.

(1p) Basfall:
Om ingen kommentar finns (down är null) ska inget skrivas ut.

(4p) Allmänt fall/rekursionsfall:
Skriv ut kommentaren. Skriv sedan ut underkommentarerna (down) rekursivt (med en parameter som ökas med ett i anropet för att hålla reda på antalet stjärnor) och skriv sist ut kommentarerna på samma nivå (next) rekursivt.

3. *Diskotek*

(6p) Plocka ut första låten ur kön, spela upp den och stoppa in den i binärträdet. Plocka sedan ut nästa låt, kolla att den inte finns i trädet (redan spelad), spela upp den och stoppa in i binärträdet. Fortsätt på samma sätt tills kön är tom.

4. *Diskuskast*

(1p) Distributionsräkning (räknesortering) fungerar här. Om vi antar att kastlängden mäts i centimeter har vi cirka 2000 olika kastlängder i intervallet 50-70 m. Tidsåtgång: ca 2N dvs 2 miljoner läsningar från filen (inte jämförelser).

(1p) Mergesort har komplexitet $O(NlogN)$. Beräknar vi $Nlog_2N$ får vi ungefär 20 miljoner jämförelser. Men här krävs också extra minnesutrymme under sorteringen.

(1p) Insättningssortering har komplexitet $O(N^2)$ om man sorterar om alla posterna. Men här ska vi bara sortera in M resultat från senaste tävlingen, där M är mycket mindre än N, ca 50. Då får vi $M * N$=50 miljoner.

(1p) Urvalssortering har komplexitet $O(N^2)$ ty man sorterar om alla posterna. Tusen miljarder jämförelser, vilket är onödigt mycket.

Allra snabbast är att sortera dom nya resultaten (det tar ingen tid för det är så få) och sedan samsortera dessa med resten (en enda merge). Det kräver N jämförelser.

5. *Diskål*

Algoritm:

(8p) Sortera först ordlistan i en vektor efter ordlängd (längsta orden först) med räknesortering (distributionsräkning), så att vi kan gå igenom dom i längdordning (här spelar bokstavsordning ingen roll).

Lägg i samma veva in orden i ett binärt sökträd (eller Bloomfilter), som ska användas för att kolla om delorden finns i ordlistan.

1. Ta ut ett ord i taget från vektorn.
2. Gå igenom en kontrollslinga som delar ordet i två delord med längd 2 och ordlängd-2, 3 och ordlängd-3 osv.
3. Undersök för varje delning om bägge delorden finns i ordlistan. Avbryt slingan om något av orden inte finns med.

Det första ord som klarar sig igenom hela kontrollslingan är det längsta så då avbruter vi sökningen.

Datastrukturer: Orden läses in från en fil och lagras i en *vektor* (sorterade i längdordning) och i ett *binärträd* (sorterat i bokstavsordning) eller i ett *Bloomfilter* (hashtabell med ett antal booleska hashfunktioner).

Klassuppdelning: En klass för hantering av binärträdet/Bloomfiltret med metoder för att stoppa in ett ord och för att kolla om ett ord existerar. En klass för resten, med metoder för distributionsräkning och för att gå igenom kontrollslingan. Metoder för orddelning finns i klassen String.

(6p) **6.** *Diskmaskinssyntax*

<diskprogram> ::= <fördisk><diskning><sköljning><torkning>
<fördisk> ::= F | ε
<diskning> ::= <disk> | <disk><diskning>
<disk> ::= D(<temperatur>)
<temperatur> ::= 50...85
<sköljning> ::= S | S<sköljning>
<torkning> ::= T | ε

7. *Diskad sökning*

(1p) Linjärsökning $O(N)$ är onödigt långsam när man vill söka efter något i ett stort *sorterat* register, t ex hela sveriges telefonkatalog.

(1p) Binärsökning i vektor kräver att vektorn är sorterad. För engångssökning i en

stor datamängd (t ex för att hitta en lotterivinnare) lönar det sig inte att sortera ($O(NlogN)$) för att sedan kunna söka snabbt med binärsökning ($O(logN)$).

(1p) Binärsökning i binärt sökträd har också komplexiteten $O(logN)$ om trädet är balanserat, men i det urartade fallet att trädet blir som en lång lista får vi ($O(N)$). Det här kan uppstå t ex för ett personregister där man lägger in nyfödda vartefter, och sorterar efter personnummer.

(1p) Hashning har komplexiteten $O(1)$. En hashtabell är olämplig när data och datamängd varierar mycket över tiden.

8. *Diskofildata*

(3p) Abstraktion är bra eftersom:

Den som programmerar CD-spelaren inte behöver bry sig om hur data har lagrats (det syns ju inte i en abstrakt datatyp) och det konkreta lagringssättet på nya CD-skivor kan ändras utan att CD-spelaren behöver programmeras om (eftersom det finns metoder för åtkomst).

(2p) Man bör se till att ett abstrakt `Stycke` låter sig anpassas till olika typer av musik. `Stycke` kan vara en klass med bland annat följande innehåll.

```
Stycke(); // Konstruktor
int getNumber();
double getLength();
String getName();
String getInfo();
```