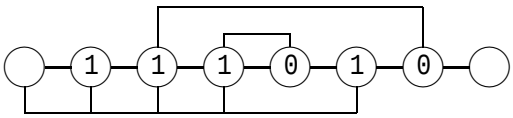
KTH, NADA, Henrik Eriksson

2D1320, Tilda, Tentamenslösning 7 mars 1998

1. *Knuthautomat*



i	next[i]
1	0
2	0
3	0
4	3
5	0
6	2

Boyer–Moore tittar i stället på vart sjätte tecken. Så länge det inte är en etta eller en nolla hoppar man vidare. Om det är en etta eller nolla undersöker man närmaste omgivningen noga och hoppar sedan vidare.

2. *Ordning i kön*

- Lägg en dumpost med namnet Pnyxtr sist i kön.
- Ta en post ur kön, se på namnet och pusha den på stacken. Upprepa sedan:
  1. Ta en post ur kön.
  2. Om det är olika namn poppar vi hela stacken till kön.
  3. Om det är dumposten bryts slingan, annars pushar vi posten på stacken.

3. *Rekursiv enbenthet*

```
IF p=NIL THEN RETURN 0
ELSIF p.left=NIL AND p.right=NIL THEN RETURN 0
ELSIF p.left=NIL THEN RETURN 1+Onelegs(p.right)
ELSIF p.right=NIL THEN RETURN 1+Onelegs(p.left)
ELSE RETURN Onelegs(p.left)+Onelegs(p.right)
END;
```

4. *Hashade molekyler* Låt p vara ett primtal i närheten av 15000 och låt a_i vara antalet atomer med atomnummer i, där $1 \leq i \leq 105$. En lämplig hashfunktion ges av

```
h:=13827;
FOR i:=1 TO 105 DO h:=(128*h+a[i]) MOD p; END
```

Hashvektorn tar 15000*4 byte, hashposterna 10000*68 byte (namn och datapekare), dataposterna 10000*200 byte. Totalt alltså 2.74 megabyte.

5. *Hinkproblemet*

Breddenförstsökning med kö är bäst. En position är en vektor med hinkinnehåll och faderspekare. En son skapas genom hällning från hink i till hink j.

För att slippa dumsöner kan man koda en lösningsposition som ett heltal, till exempel med $262144h_1 + 4096h_2 + 64h_3 + h_4$, där h_i är innehållet i hink i. För varje position man skapar sparar man koden i ett dumsonsbinärträd eller en dumsonshashvektor.

När en lösning dyker upp skriver man ut kedjan rekursivt, för att få den i rätt ordning.

Modulerna Queue och Bintree behövs och i huvudmodulen finns procedurerna MakeSons och WriteChain och en posttyp med hinkvektor och faderspekare.

6. *Syntax för misstänksamma*

```
<mening> ::= <subjekt><verb><attsats> | <subjekt><verb> INTE <attsats>
<subjekt>::= JAG | DU
<verb>   ::= ANAR | LÅTSAS
<attsats>::= <nothing> | ATT <subjekt><verb><attsats>
<attsats>::= ATT <subjekt> INTE <verb><attsats>
```

Proceduren `ReadMening` anropar `ReadSubjekt`, `ReadVerb`, tjuvtittar och glufsar eventuellt INTE och anropar slutligen `ReadAttsats` osv enligt syntaxen.

7. *Sökarna*

Sökning i en hashvektor med femtio procent luft kräver i genomsnitt drygt en jämförelse. I ett binärträd med tjugo poster kräver en sökning cirka fem jämförelser. Därför tog viggosökning en fjärdedel så lång tid som felipesökning. Med trehundra deltagare blir krocklistlängden drygt tio och en misslyckad sökning tar i genomsnitt drygt tio jämförelser. I ett binärträd med trehundra poster tar misslyckad sökning bara åtta jämförelser.

8. *Abstrakta hinkar*

En abstrakt `Hink.T` kan vara en objekttyp med följande gränssnitt.

```
INTERFACE Hink                   (* Huvudprogrammet kan ha          *)
PROCEDURE Make(size:INTEGER):T; (* VAR h1:=Hink.Make(17);          *)
TYPE T<:AbstraktHink;
     AbstraktHink = OBJECT       (* Implementeringen av T           *)
     METHODS                     (* har förstås fälten              *)
       contents():INTEGER;       (* size:INTEGER och                *)
       spaceLeft():INTEGER;      (* content:INTEGER:=0              *)
       empty():BOOLEAN;
       full():BOOLEAN;           (* Exempel på användning:          *)
       add(amount:INTEGER);      (* IF h1.spaceLeft()<h2.contents() *)
       fill();                   (* THEN h2.add(-h1.spaceLeft());   *)
     END;                        (*      h1.fill();                 *)
END Hink.                        (* END;                            *)
```