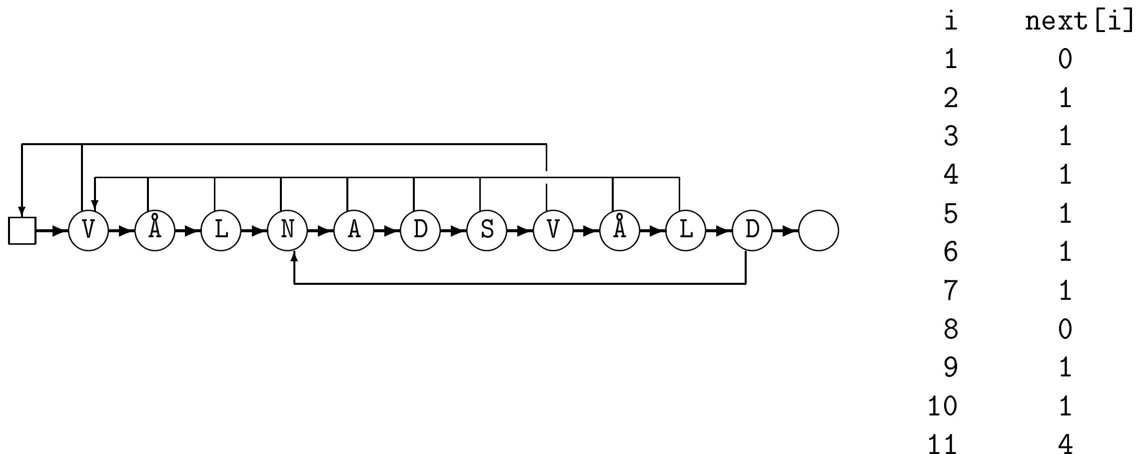
2D1320, Tilda, lösningsförslag 19 oktober 2002

(5p) **1.** *Spökspö*



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

2. *Binärt spökträd*

(2p) Eftersom orden kommer i bokstavsordning blir trädet bara en lång tarm. Därför
(1p) krävs sju jämförelser, lika många som antalet ord, för att upptäcka att ordet *vampyr* inte finns med.

(2p) Ett balanserat träd:

```
                        gengångare
          fantom                                spöke
    ande          gast                poltergeist        vålnad
```

(1p) Basfall:
Om trädet är ett löv ska det skrivas ut.
(3p) Allmänt fall/rekursionsfall:
Om det finns ett vänsterträd – skriv ut vänsterträdets löv. Om det finns ett högerträd – skriv ut högerträdets löv.
(1p) Vad skrivs ut? För trädet ovan blir det:
`ande gast poltergeist vålnad`

3. *Spöktåg*

(6p) Tåget står på spår A, det har n vagnar och önskad permutation finns i `önsketåg[0..n-1]`.
För varje vagn i önsketåget:
```
for (int i=0; i<n; i++)
```
Kör vagnar från A till B tills önskad vagn dyker upp.
```
while ((vagn=A.pop()) != önsketåg[i]) B.push(vagn)
```
Kör då önskad vagn till spår C
```
C.push(vagn)
```
och kör tillbaka vagnarna från B till A.
```
while (!B.isEmpty()) A.push(B.pop())
```
Tåget står nu på C. Flytta det till A.
```
while (!C.isEmpty()) A.push(C.pop())
```

4. *Spökenas tio i topp*

(4p) Räknesortering (distributionsräkning) används då vi vill sortera en samling data där många värden är lika varandra, men så är inte fallet här. Quicksort sorterar stora datamängder snabbt men är onödigt komplicerad när bara tio värden ska sorteras. Insättningssortering är en enkel metod som duger bra för bara tio värden, urvalssortering likaså.

(2p) För att få fram och rangordna de tio bästa av tusen är urvalssortering bäst – man kan ju bryta när de tio första är klara. Både quicksort och insättningssortering kräver att man sorterar alla tusen innan man kan se vilka som är de tio bästa.

5. *Spökhuset*

(8p) Använd breddenförstsökning med en kö. Söner är alla intilliggande rum. De rum som läggs i kön läggs samtidigt in i ett dumträd (för att förhindra att man går runt i cirklar). När utgången hittas kan sökningen avbrytas. Lämpliga datastrukturer är kö, dumträd och en vektor eller matris för kartan.

(6p) **6.** *Andemeningen*

```
<andemening> ::= <vrål> | <skratt>
<vrål>  ::= B<un>
<un> ::= U | U<un>
<skratt>  ::= H<vokal> | H<vokal><skratt>
<vokal>  ::= A | E | I | O | U | Y | Å | Ä | Ö
```

(4p) **7.** *Spökhistoria* Om man för enkelhets skull bara vill ha en enda hashtabell kan man hasha in varje spökhistorieobjekt två gånger i samma hashtabell, dels på ort och dels på spöktyp. (Alternativt har man två hashtabeller, en för orthashning och en för spöktypshashning, och hashar in varje objekt i båda tabellerna.) Hashtabellerna ska innehålla *referenser* till historieobjekten. Som hashfunktion fungerar till exempel Javas hashCode() bra. Gör en uppskattning av hur många spökhistorier som kommer att stoppas in i databasen och skapa sedan en hashtabell med dubbelt så många platser. Använd en hashtabell med krocklistor för då fungerar hashningen även om antalet objekt blir större än tabellens storlek.

(3p) **8.** *Spökposter*

En abstrakt datatyp förenklar programmeringsarbetet eftersom man inte behöver tänka på spökimplementationen. Om spöket är abstrakt kan man ändra representationen utan att det påverkar programmet i övrigt, vilket gör det enklare att använda spöket i andra program. Dessutom förenklar abstraktion samarbetet med andra programmerare. Ett `Spöke` kan vara ett objekt med bland annat följande anrop.

```
Spöke(int läskighet, String andemening); // Konstruktor
int geLäskighet();
void bliLäskigare(int procent);
void bliSnällare(int procent);
void skräm();
String väsnas();
```