Kapitel 4

Kö

4.1 Kö

I vardagslivet är *köer* vanligare än stackar – kanske alldeles för vanliga. En kö är alltså en ”anordning”, som har *ett första* och ett *sista element*. Element som *tillkommer*, läggs till sist i kön och element som *tas bort* hämtas från köns början. Till skillnad från stacken, som vi minns kallades för LIFO, kallas en kö för FIFO *(first in first out)*.

4.1.1 Operationer

När en ny ADT presenteras så ska man direkt ställa sig frågan: *Vilka operationer är förknippade med datastrukturen*?

Operation	**Förklaring**
Empty	Skapar en tom kö
Enqueue(v)	Sätter in elementet v sist i kön
Isempty	Testar om kön är tom
Front	Ger värdet av första elementet i kön
Dequeque	Tar bort första elementet v från kön

När vi studerar operationerna till denna datastruktur, ser vi hur lik denna datastruktur är datastrukturerna – *stack* och *lista*. Vi inser att vi skulle klara oss med enbart datastrukturen *lista*. Det är bara namnen på operationerna som skiljer.

4.1.2 Akademisk diskussion

Det verkar inte orimligt att man i en tillämpning av kö behöver ha reda på hur lång kön är just nu. Det är förstås praktiskt att ha en operation, *Length* i gränsytan, som returnerar antalet element i kön. Experterna anser att en sådan operation inte är *teoretiskt motiverad*.

”Goddag yxskaft”, säger säkert praktikern och försöker konstruera en egen operation i sitt program som bygger på de givna operationerna. Hur ska detta göras och hur effektiv blir den?

Här nedan följer ett citat som ska ge dig en känsla för hur teoretikern tänker:”Tyvärr är effektiviteten låg hos den ovan efterfrågade operationen. Med operationen inkluderad i gränsytan finns möjlighet att göra effektivare implementationer, genom att man då har tillgång till den speciella implementationen av Kö som används. Hittills har vi dock försökt undvika att låta operationsrepertoaren svälla ut för att täcka en mångfald specifika tillämpningssituationer. Man kan befara att en alltför liberal politik öppnar vägen för allmän förslumning av gränsytan”.

4.1.3 Felhantering

Vi tar åter upp problemet med vad som ska hända om man använder operationer på ett felaktigt sätt på en datastruktur. Vad tycker du ska hända om *Decueue* eller *Front* används på en tom kö?

Vi slår nu fast att det är upp till användaren att utforma sina algoritmer på ett sådan sätt att en felaktig användning av en operation inte kan inträffa. Eftersom operationen *Isempty* finns kan problemet enkelt undanröjas!

Så när vi implementerar en datastruktur så använder vi till exempel *assert* för att underlätta något för programutvecklaren. Det är mindre katastrof om exekveringen avbryts än om den fortsätter med felaktiga värden! I övrigt har den som gör implementationen inget ansvar.

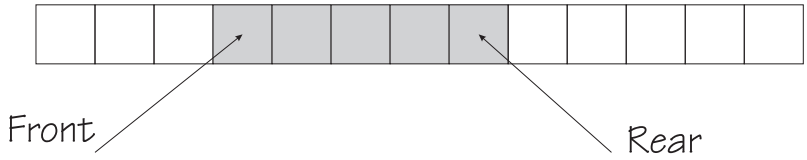
Ett annat fel som är svårare att hantera är då kön blir full. Det är ingen tillfällighet att teoretikerna inte tar med operationen *Isfull* i repertoaren. För att avgöra om en kö är full måste man ta hänsyn till implementationen och det vill man inte göra på den här nivån.

4.1.4 Implementation som array

Tar man direkt modellen från *lista implementerad som array* tappar man onödigt mycket effektivitet. Det är främst operationen *remove* som kostar mycket eftersom alla bakomliggande element då ska flyttas ett steg framåt i arrayen.

Lösningen på effektivitetsproblemet heter *cirkulär array*. I figuren visas en kö i form av en vanlig array. Vi har två variabler, *front* och *rear*, som anger index till första respektive sista elementet i kön.

När ett element hämtas från kön ökas *front* med 1 och när ett element läggs till i kön så ökas *rear* med 1. Att lägga till och ta bort element på kön tar nu konstant tid i anspråk. Dessa operationer har inget att göra med hur många element det finns i kön.



Figur 4.1:

Problemet med denna implementation är dock att kön ”glider iväg” och så småningom når *rear* arrayens övre gräns. Det är nu vi gör den till en ”cirkulär array”. Variabeln *rear* ”hoppar” till början av arrayen. Senare så når också *front* fram till slutet och får då göra samma hopp.

Det enda problem som egentligen återstår är om *rear* försöker passera *front* – vi har fler element i kön än vad arrayen kan hålla.

Vår första implementation blir som vanligt ett enkelt program med fastlåst datatyp och där operationernas funktioner inkluderas i klientens program.

```
1  #define MAXQUEUE 10
2  struct person{
3    char namn[20];
4    char telnr[20];
5  };
6
7  typedef struct person persontyp;
8
9  struct queue{
10   int count;
11   int front;
12   int rear;
13   persontyp list[MAXQUEUE];
14 };
15 typedef struct queue queuetyp;
16 queuetyp q;
```

1 Först definierar vi köns maximala storlek.

2–7 Elementen i vår kö kommer att vara av typ *persontyp*.

9–14 Just i den här implementationen samlar vi alla data om kön i en speciell post. Där finns uppgifter om *antalet element på kön* (count), *index till första elementet* (front), *index till sista elementet* (rear) och till sist själva kön (list).

```
1 void Empty(){
2   q.count=0;
3   q.front=0;
4   q.rear=0;
5 }
```

Denna operation initierar kön och ska anropas innan kön börjar användas.

```
1 void Enqueue(persontyp p){
2   assert(q.count<MAXQUEUE);
3   q.count++;
4   q.list[q.rear]=p;
5   q.rear=(q.rear+1)%MAXQUEUE;
6 }
```

1 Funktionen *Enqueue* har ett element av *persontyp* som inparameter. Funktionen returnerar ingenting.

2 Vi inleder med *assert*, som testar att det finns plats på kön för ännu ett element.

3 Antalet element i kön räknas upp.

4 Elementet *p* placeras så sist i kön.

5 Index till första lediga platsen i kön räknas upp. Här kommer detta programs enda fiffighet. Förvissa dig om att du förstår hur *rear* index hoppar till början av arrayen när det är dags för det.

```
1 persontyp Dequeue(void){
2   persontyp x;
3   assert(q.count>=1);
4   q.count--;
5   x=q.list[q.front];
6   q.front=(q.front+1)%MAXQUEUE;
7   return x;
8 }
```

1 Den här funktionen har inga inparametrar. Funktionen returnerar den post som är först i kön.

2 Vi behöver tillfälligt en lokal post för att lagra elementet som ska returneras.

3 Vår lilla vakt dyker upp här också. Finns det verkligen ett element att returnera?

4 Antalet element i kön justeras.

5-7 Elementet hämtas ut. *Front* räknas upp med samma trick som för *rear* tidigare. Till sist returneras elementet.

```
1 int Length(void){
2   return q.count;
3 }
4 int Isempty(void){
5   return q.count==0;
6 }
7 int Isfull(void){
8   return q.count==MAXQUEUE;
9 }
```

Här har vi samlat tre hjälpoperationer. Vi trotsar experterna och implementerar både *Length* och *Isfull*

1-3 Operationen returnerar hur många element det finns i kön.

5-7 Operationen returnerar **true** om kön är tom och **false** annars.

9-11 Operationen returnerar **true** om kön är full och **false** annars.

4.1.5 Implementation som länkad lista

Som en avdramatisering väljer vi här, när vi ska implementera kö som en länkad lista, att välja våra egna namn på operationerna. De är dock inte svåra att identifiera. Det är lättare att göra det här eftersom namnen *Enqueue* och *Dequeue* inte är lika inarbetade som stackens *Pop* och *Push*.

Vi nöjer oss med att implementera tre operationer. Att lämna *Isfull* är inte så dramatiskt här eftersom vi tänker oss *heapen* som ett ”hav av minne” som säkert räcker för våra behov.

Vi implementerar de tre operationerna *tom, laegg_till* och *ta_ut*. Samma svaghet som i förra exemplet. `Koeelement` kan bara hålla reda på ett heltal.

```
1  struct koeelement{
2    int tal;
3    struct koeelement *naesta;
4  };
5
6  typedef struct koeelement koeelement;
7  koeelement *foerst,*sist;
8
9  int tom(void){
10   return (foerst==NULL);
11 }
```

1–4 Precis som för en stack, behöver vi ett element med en pekare till ett element av samma typ.

7 De två pekarna som ska hålla reda på var kön börjar och slutar, deklarerar vi globalt och kallar för *foerst* och *sist*.

9–11 Funktionen *tom* känner bara efter om *foerst* har värdet NULL. Detta är liktydigt med att kön är tom.

```
1  void laegg_till(int t){
2    koeelement *ny;
3    ny=(koeelement *)malloc(sizeof(koeelement));
4    ny->tal = t;
5    ny->naesta = NULL;
6    if(tom())
7      foerst=ny;
8    else
9      sist->naesta=ny;
10   sist=ny;
11 }
```

3–4 Plats för ett nytt köelement skapas på heapen och värdet på variabeln *tal* kopieras dit från inparametern *t*.

5 Det nya köelementet har ingen efterföljare och därför får *ny->nasta* värdet NULL.

6–8 Om kön är tom så är detta första elementet i kön. *foerst* sättes då till att peka på det nya elementet i kön.

9 Här hamnar vi oftast, då det redan finns andra element i kön. Det element som varit sist fram till nu får en efterföljare.

10 *sist* skall alltid peka ut det sista elementet i kön, och nya element läggs alltid till sist i kön, så *sist* sättes till att peka på det nya elementet.

```
1  int ta_ut(void){
2    koeelement *tmp;
3    int laengst_fram;
4    laengst_fram = foerst->tal;
5    tmp=foerst;
6    foerst=foerst->naesta;
7    if(tom())
8      sist = NULL;
9    free(tmp);
10   return laengst_fram;
11 }
```

5 Denna funktion returnerar värdet på det första elementet i kön. Värdet på variabeln *tal* kopieras till variabeln *laengst_fram*, som senare returneras.

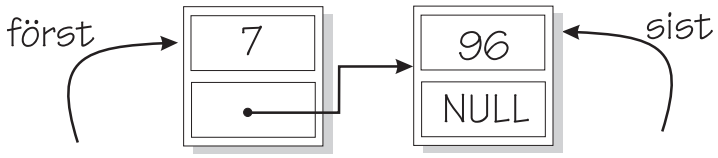
6 *tmp* behöver vi för att kunna ”avboka” platsen på heapen vilket sker i 9.

7 *foerst* får adressen till nästa elementet i kön.

8–9 Om det elementet vi tar bort från kön är det sista elementet sätts *sist* till NULL.

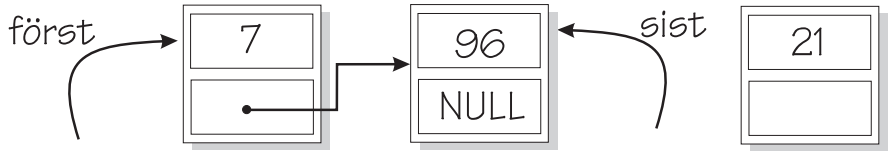
4.1.6 Ett enkelt exempel

Nedan följer några enkla satser med tillhörande figurer.

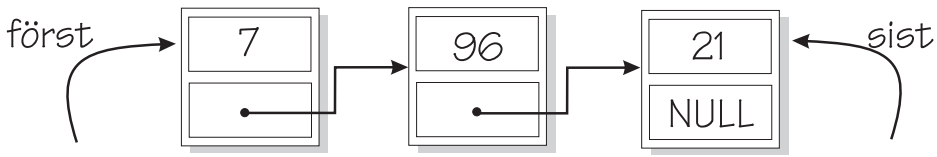


Figur 4.2: *Variabeln* `tal` *tilldelas värdet* 7 *och adderas till kön. På samma sätt används sen* `tal` *till att lägga till talet* 96 *i kön.*

```
1   int tal;
2   tal=7;  laegg_till(tal);
3   tal=96; laegg_till(tal);
```

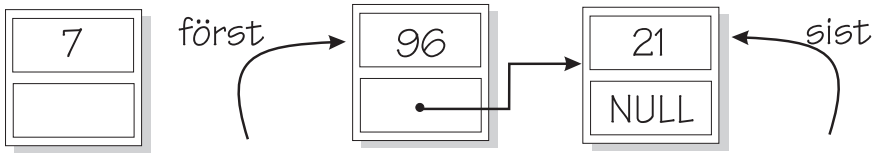


Figur 4.3: *Talet 21 står nu på tur att hamna i kön*



Figur 4.4: *21 är på plats*

```
1   tal=21; laegg_till(tal);
```



Figur 4.5: *Efter första varvet i loopen nedan hämtas talet 7 från kön.*

```
1    while(!tom()){
2       tal=ta_ut();
3       printf("%d",tal);
4    }
```

4.1.7 Mer om rekursion

Svansrekursion (tail recursion) kan mycket enkelt ersättas med **iteration**. Kännetecknet för en svansrekursion är att den rekursiva funktionen avslutas med **ett** anrop på sig själv. Här ett exempel där vi summerar talen i en vektor `a`.

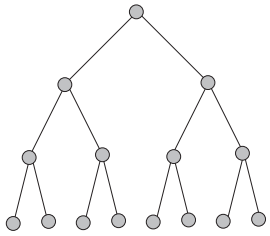
```
1  #include <stdio.h>
2  int sum(int a[],int n){
3    if(n==0)
4      return a[0];
5    else
6      return a[n]+sum(a,n-1);
7  }
8  int main(void){
9    int a[5]={10,13,6,8,19};
10   printf("Summan=%d\n",sum(a,4));
11 }
```

4.1.8 Rekursionsträdet

Ett träd är ett lämpligt sätt att åskådliggöra hur en rekursiv funktion arbetar.

Om funktionen från ett given *nod* alltid har två *barn* och vi betraktar 10 nivåer (generationer) kommer rekursionsträdet att innehålla $2^9 = 512$ *löv*, noder längst ned i trädet. Totalt kommer trädet att innehålla 1023 noder.

Ett exempel på detta tema: *Vi befinner oss i koordinaten* $(36, 27)$ *och vill hem till* $(0, 0)$. *Vår promenad måste bestå av små promenader i rakt nordlig (8 steg), sydlig (3 steg), östlig (5 steg) och västlig (6 steg) riktning. På hur många sätt kan vi ta oss hem?*



Figur 4.6:

Självklart finns det ett oändligt antal möjligheter att nå målet och därför måste vi begränsa oss och säga att antalet små promenader måste vara ≤ 10. Så här skulle funktionen kunna se ut:

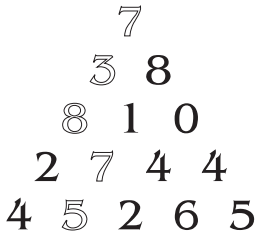
```
1  int antal=0,antalanrop=0;
2
3  void promenad(int x,int y,int n){
4    antalanrop++;
5    if(x==0 && y==0){
6      printf("Vi har hittat fram");
7      antal++;
8    }
9    else
10     if(n<10){
11       promenad(x,y+8,n+1);
12       promenad(x,y-3,n+1);
13       promenad(x+5,y,n+1);
14       promenad(x-6,y,n+1);
15     }
16 }
```

Från utgångsläget $(36, 27)$ har vi fyra val, trädets rot har fyra barn. Från varje nod därefter finns ytterligare fyra barn. Om $n = 10$ kommer trädet att få $4^{10} = 1\,048\,576$ löv och totalt $1\,398\,101$ noder. Man kan dock inte ta sig till målet med 10 små promenader. Hur många löv får vi då vi istället sätter $n = 15$? $4^{15} = 1\,073\,741\,824$ är ganska mycket och $1\,431\,655\,765$ noder totalt är ännu mer. Sensmoralen är: *Håll utkik på hur arbetet växer då trädets djup ökar eller då antalet barn från varje nod ökar.*

Om vi verkligen vill lösa detta problem finns det mycket effektivare metoder. Just i detta exempel ser vi att 9 promenader söder ut ger $9 \cdot (-3) = -27$ och 6 promenader väster ut ger $6 \cdot (-6) = -36$. Hela promenaden kan inte göras med färre än 15 små promenader. Vi kan blanda dessa i godtycklig ordning och det finns 5005 olika sätt att ta sig hem med 15 promenader.

$$\frac{15!}{9! \cdot 6!} = 5005$$

4.1.9 Ett ytterligare exempel på rekursion



Figur 4.7:

Figur 4.11 visar en *taltriangel*. Skriv ett program som bestämmer den högsta summa som kan erhållas på vägen från triangelns topp ned till något av löven i basen.

Ett steg kan tas antingen snett ned åt vänster eller snett ned åt höger. Antalet rader `n` i triangeln är $1 < n \leq 10$. Talen i triangeln är alla heltal i intervallet [`0...99`].

På filen `input.dat` finns triangelns tal. Filen inleds med ett tal som anger `n`. Här ser du filen som motsvarar figuren.

```
5
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

För testexemplet ges utskriften:

```
30
```

```
1  #include <stdio.h>
2  int tab[10][10],n,max=0;
3  void berakna(int rad,int kol,int s){
4    if(rad==n-1){
5      s=s+tab[rad][kol];
6      if(s>max) max=s;
7    }
8    else {
9      berakna(rad+1,kol,  s+tab[rad][kol]);
10     berakna(rad+1,kol+1,s+tab[rad][kol]);
11   }
12 }
```

Kommentarer:

2 Vi tillåter oss att ha `tab`, tabellen som innehåller alla talen, `n`, som talar om hur många rader som används i tabellen och `max`, som håller reda på den största funna summan, globala.

3 Funktionen `berakna` har tre parametrar: `rad` håller reda på, på vilken rad vi befinner oss. På samma sätt håller `kol` reda på var på raden vi befinner oss. `s` håller delsumman så långt. Vill vi undvika globala variabler får vi öka ut parameterlistan med tre yttreligare parametrar.

4-7 Stoppet i den rekursiva processen är när vi når triangelns bas. Radnumret i tabellen är då `n-1`. Vi adderar det sista talet till `s` som håller summan och jämför sedan `s` med det hittills bästa resultatet.

9-10 Har vi inte nått ett löv, basen i triangeln, har vi två fortsättningar. I båda fallen ökar vi `rad` med `1`. Värdet för `kol` bestäms av om vi ska gå åt vänster eller höger. `s` ökas med värdet hos den nod i vilken vi just nu befinner oss.

```
1  int main(void){
2    FILE *infil;
3    int i,j;
4    infil=fopen("input.dat","rt");
5    fscanf(infil,"%d",&n);
6    for(i=0;i<n;i++)
7      for(j=0;j<=i;j++)
8        fscanf(infil,"%d",&tab[i][j]);
9    fclose(infil);
10   berakna(0,0,0);
11   printf("Maximal summa: %d\n",max);
12 }
```

Kommentarer:

H Inläsning av data till `tab`, anrop av `berakna` och utskrift av resultatet.

Observera: Detta är ingen speciellt effektiv algoritm för att bestämma den största summan. Vi kommer i kapitlet om *Dynamisk programmering* att återknyta till detta problem och ange en betydligt snabbare metod.

Att arbeta med rekursiva funktioner, som i detta kapitel, snuddar vid en teknik, som kallas *backtracking*, som vi återkommer till. Vi ska då lösa problem, som till exempel labyrinter och pussel.

4.1.10 Kö och Bredden Först

Här ska vi fokusera, dels på hur en kö kan användas och dels på de problem som kan uppstå vid rekursion.

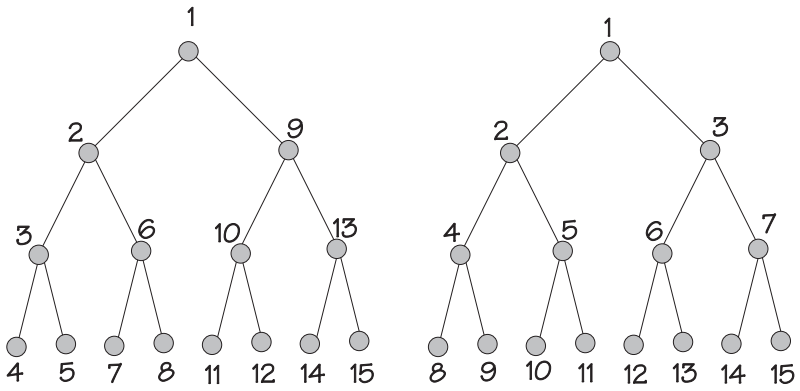
4.1.11 *Djupet först* kontra *Bredden först*

I problemet *Hissen i lustiga huset* räcker det uppenbarligen inte att ha kontrollen, att man inte hamnar utanför huset, som avbrottsvillkor. I varje ”varv” finns högst två möjligheter: *att åka uppåt* eller *att åka nedåt.*

Som vi skriver funktionen kommer den ena möjligheten att väljas tills den inte längre är möjlig. Då kommer det andra valet att ta vid och utföras till det första är möjligt igen. Troligtvis kommer detta, för de flesta indata, att leda till att vi hamnar i en loop och därmed aldrig når målet, trots att det skulle kunna vara möjligt.

Det finns flera metoder, en del riktigt enkla, att lösa detta problem. Vi ska här diskutera *bredden först*, en viktig metod som är overkill för just detta problem, men som samtidigt ger oss en naturlig användning av en kö.

Som motpol till *bredden först* står *djupet först.* Den senare tekniken är just den vi, utan framgång, försökt använda på problemet *Hissen i lustiga huset.* Nedanstående figur ger stöd för vidare diskussion:



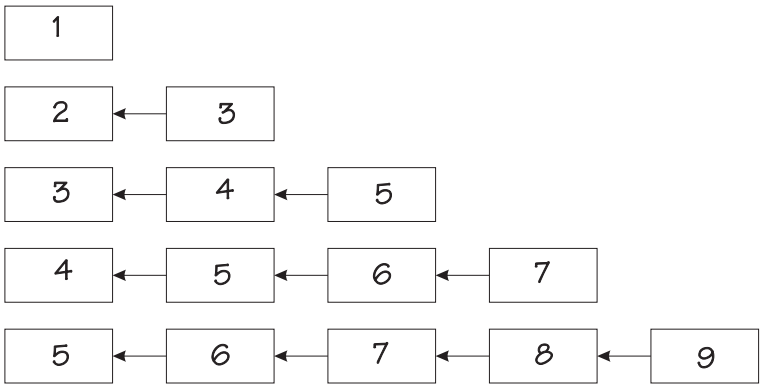
Figur 4.8: *Till vänster ser vi ordningen i vilken noderna i ett rekursionsträd besöks när tekniken kallas djupet först. Till höger gäller nummerordningen tekniken bredden först*

Om vi är på jakt efter en lösning, där vi saknar kunskap om, efter hur ”många drag” eller på vilket djup den kommer att dyka upp i rekursionsträdet och då det heller inte finns ett bestämt maximalt djup i trädet, så kanske vi aldrig når fram till lösningen utan i stället kommer att arbeta långt ned trädet och långt ifrån lösningen.

Undersöker vi istället noderna som de är numrerade i det högra trädet, det vill säga en nivå i taget, kommer vi, då vi stöter på en lösning att kunna säga att på ingen annan plats i trädet finns en nod med eftersökt lösning på grundare nivå än den vi funnit. Det kan möjligtvis finnas en på samma nivå.

Djupet först är starkt förknippad med en stack, en mekanism vi använder vid rekursion. Bredden först är på samma sätt starkt förknippad med en kö. Stacken och dess funktioner har vi gratis i språket C. Kön får vi, som programmerare, skapa och hantera själva.

Från början finns en situation. Den kan beskrivas till exempel i en `struct`. Detta objekt, ställer vi i kö.



Figur 4.9: *Här ser vi hur kön förändras tills vi finner lösningen i nod 5.*

Direkt hämtar vi ned detta objekt (nod) från kön och placerar därefter barnen till objektet (noden) på kön. I nästa steg hämtar vi ned objekt 2, undersöker om detta är en lösning och om inte genererar vi två nya objekt till kön 4 och 5. Om vi finner en lösning vet vi alltså att det inte finns en lösning på lägre nivå.

4.1.12 Mjölkflaskorna

Vi ska nu använda denna teknik på följande klassiska problem:

En mjölkförsäljare har tre flaskor, som rymmer 12, 8 *respektive* 5 *liter.* 12*-litersflaskan är full med mjölk och de andra två är tomma. Försäljaren vill nu dela upp mjölken så att* 12*-litersflaskan och* 8*-litersflaskan var och en innehåller* 6 *liter mjölk. Han häller fram och tillbaka mellan flaskorna utan att någon mjölk går till spillo. Varje hällning måste fortsätta tills antingen avgivande flaska är tom eller mottagande är full. Målet är att göra denna uppdelning med så få hällningar som möjligt.*

Skriv nu ett generellt program som tar emot uppgifter om antalet flaskor $\mathtt{nf}, 2 \leq \mathtt{nf} \leq 5$*), vilken volym varje flaska har, hur många liter mjölk flaskan innehåller från början och hur många liter flaskan ska innehålla när arbetet är klart. Talet* -1 *berättar att just den flaskans innehåll till slut, är ointressant.*

Så här kan en dialog komma att se ut:

```
Antal flaskor: 3
Flaska 1, volym   : 12
Flaska 1, innehåll: 12
Flaska 1, mål     :  6
Flaska 2, volym   :  8
Flaska 2, innehåll:  0
Flaska 2, mål     :  6
Flaska 3, volym   :  5
Flaska 3, innehåll:  0
Flaska 3, mål     :  0
```

Så kan svaret ges:

```
Drag 1: Från 1 till 2
Drag 2: Från 2 till 3
Drag 3: Från 3 till 1
Drag 4: Från 2 till 3
Drag 5: Från 1 till 2
Drag 6: Från 2 till 3
Drag 7: Från 3 till 1
```

Med hjälp av följande program kan vi lösa problemet:

```
1  #include <stdio.h>
2  #include <conio.h>
3  #include <stdlib.h>
4
5  struct drag{
6    int dragnr;
7    int flaskor[2];
8    int vol[5][3];
9    int dragen[20][2];
10   struct drag *next;
11 };
```

Kommentarer:

H Objektet som ska köas är definierad som en `struct`.

6 `dragnr` talar om på vilken nivå i trädet objektet skapades och om objektet är en lösning så talar `dragnr` om hur många drag (hällningar), som behövs för att nå hit. Objekt `1` finns har `dragnr=0`.

7 Det senaste draget som ledde hit lagras här. Från flaska i index `0` och till flaska i index `1`.

8 All information om flaskornas status och mål. Lite onödigt kanske, eftersom två av raderna i `vol` är konstanta och därmed lika i samtliga objekt.

9 Hela ”dragserien”, alla hällningar som leder fram till denna situation.

10 En pekare, som nästa objekt i kön.

```
1  struct drag *start,*stop;
2  int undersokt=0;
3
4  int empty(void){
5    return (start==NULL);
6  }
7
8  void enqueue(struct drag d){
9    struct drag *ny;
10   ny=(struct drag*)malloc(sizeof(struct drag));
11   *ny=d;
12   if(empty()){
13     start=ny;
14     start->next=NULL;
15   }
16   else
17     stop->next=ny;
18   stop=ny;
19 }
20
21 struct drag dequeue(void){
22   struct drag *tmp,d;
23   d=*start;
24   tmp=start;
25   start=start->next;
26   if(empty())
27     stop=NULL;
28   free(tmp);
29   return d;
30 }
```

Kommentarer:

H Detta är inget annat än en kopiering av de funktioner som hanterar en kö, implementerad som en länkad lista och som finns i boken.

1 `start` och `stop` är två globala pekare som håller reda på var första respektive sista objektet i kön finns.

2 `undersokt` är en variabel, som håller reda på hur många objekt som har genererats under exekveringen. Endast för statistikens skull.

```
1  void init(int *nf){
2    int i;
3    struct drag nu;
4    start=NULL;
5    printf("Antal flaskor: ");
6    scanf("%d",nf);
7    for(i=0;i<*nf;i++){
8      printf("Flaska %d, volym   : ",i+1);
9      scanf("%d",&nu.vol[i][0]);
10     printf("Flaska %d, innehåll: ",i+1);
11     scanf("%d",&nu.vol[i][1]);
12     printf("Flaska %d, mål    : ",i+1);
13     scanf("%d",&nu.vol[i][2]);
14   }
15   nu.dragnr=0;
16   nu.flaskor[0]=-1;
17   nu.flaskor[1]=-1;
18   enqueue(nu);
19 }
```

Kommentarer:

H En funktion för initiering och inläsning av data.

4 `start` håller reda på första elementet i kön.

5-6 `nf` håller reda på hur många flaskor som ingår i problemet.

7-17 Det första objektet byggs upp i `nu`.

18 och läggs här på kön.

```
1  int koll(int volym[5][3],int nf){
2    int i;
3    for(i=0;i<nf;i++)
4      if(volym[i][1]!=volym[i][2] && volym[i][2]!=-1)
5        return 0;
6    return 1;
7  }
```

Kommentarer:

H Denna funktion tar reda på om målet har uppnåtts. Raderna 1 och 2 i matrisen `vol` jämförs. Om värdena inte överensstämmer kan jämförelsen endast godkännas om motsvarande element i rad 2 innehåller -1, ”vad som helst är tillåtet”. Om funktionen returnerar 1 har programmet funnit en lösning.

```
1 int main(void){
2   int nf,from,to,i,klar;
3   struct drag nu,t;
4   init(&nf);
```

Kommentarer:

3 Vi behöver två variabler av typen `struct drag`, här kallade `nu` och `t`.

4 Efter anrop av `init` vet vi allt vi behöver om problemet och första objektet finns i kön.

```
1   do{
2     klar=0;
3     if(!empty()){
4       nu=dequeue();
5       if (!koll(nu.vol,nf)){
6         for(from=0;from<nf;from++)
7           for(to=0;to<nf;to++)
8             if(from!=to){
9               t=nu;
10              if(t.vol[from][1]>0 && t.vol[to][1]<t.vol[to][0] &&
11                 !(from==t.flaskor[1] && to==t.flaskor[0])){
12                if(t.vol[to][1]+t.vol[from][1]<=t.vol[to][0]){
13                  t.vol[to][1]=t.vol[to][1]+t.vol[from][1];
14                  t.vol[from][1]=0;
15                }
16                else {
17                  t.vol[from][1]=t.vol[from][1]−
18                                 (t.vol[to][0]−t.vol[to][1]);
19                  t.vol[to][1]=t.vol[to][0];
20                }
21                t.dragnr++;
22                t.flaskor[0]=from;
23                t.flaskor[1]=to;
24                t.dragen[t.dragnr][0]=from+1;
25                t.dragen[t.dragnr][1]=to+1;
26                enqueue(t);
27                undersokt++;
28              }
29            }
30      }
31      else
32        klar=1;
```

```
33      }
34      else
35        klar=1;
36    } while(!klar);
```

Kommentarer:

H Programmets hjärta! Kanske borde det ha skrivits som en egen funktion.

1-33 Det hela fungera som en `do`-loop. Den avbryts inte förrän `klar` blir `1`, (true).

2 Därför sätter vi `klar` till `0` (falsk).

3 Bara om det finns något objekt i kön kan vi fortsätta. Om kön skulle vara tom, så beror det på ett fel i problemet. Möjligheten att hälla fram och tillbaka upphör liksom aldrig.

4 Vi tar ner första objektet från kön och lagrar i `nu`.

5 Endast om det inte är en lösning, kommer vi att expandera detta objekt. Är det däremot en lösning hamnar programmet direkt i 32 där `klar` får värdet `1`.

6-7 Varje flaska ska nu få chansen att bli en ”från”- och ”till”-flaska. Om kombinationen kommer att användas beror på några olika faktorer.

8 Om ”från” och ”till”-flaska är samma blir det ingen hällning av.

9 `ut` kopieras nu till `t`. `t` kommer nu att förändras innan den sätts in i kön. Nästa objekt som ska skapas måste starta utifrån samma `nu`.

10-20 För att någon hällning över huvud taget ska bli aktuell måste: det finnas mjölk i ”från”-flaskan, ”till”-flaskan får inte vara full och vi ska inte hälla tillbaka till föregående flaska.

12-20 Det finns nu två olika situationer att ta hänsyn till: 1) Det som ryms i ”till”-flaskan är mindre än det som finns tillsammans i ”till”-flaskan och ”från”-flaskan. 2) Eller så ryms inte allt från ”från”-flaskan i ”till”-flaskan. När vi delat upp dessa två fall kan vi beräkna vad som kommer att finnas i de två flaskorna efter hällningen.

21-25 Vi uppdaterar nu de andra variablerna i objektet

26 innan vi placerar det i kön.

```
1   printf("Antal undersökta: %d\n",undersokt);
2   for(i=1;i<=nu.dragnr;i++)
3     printf("Drag %2d: Från %d till %d\n",
4           i,nu.dragen[i][0],nu.dragen[i][1]);
5   getch();
6 }
```

Kommentarer:

H Utskrift av resultatet.

4.1.13 Beräkning av aritmetiska uttryck

Vi önskar ett program som läser in en sträng bestående av ett aritmetiskt uttryck och som sedan beräknar dess värde. Programmet ska givetvis ta hänsyn till operatorernas prioritet.

Vi bestämmer oss för att uttrycken ska få innehålla positiva heltalskonstanter i intervallet 0 till 9, operatorerna: *multiplikation, division, addition och subtraktion* (*, /, + och -) samt *runda parenteser*. Det känns naturligt att begränsa uttryckets längd till en rad på skärmen, det vill säga 80 tecken. Detta betyder att varken antalet operander eller operatorer kan överstiga 40.

Lösa funderingar Att programmet kommer att bestå av två delar, en för uppackning av strängen och en för själva beräkningen inses snart. I det program som kommer att presenteras här finns ingen *syntaxkontroll*, ingen test av indata. Detta för att inte skymma de centrala delarna av algoritmen och för att inte göra programmet alltför långt.

Om man inte tillåter parenteser och heller inte, likt enkla räknedosor tar någon hänsyn till prioritetsreglerna för de olika räknesätten, blir programmet trivialt. Man läser helt enkelt ett tecken i taget ur strängen testar om det är en siffra eller operator. När operatorn kommer är talet avslutat. Efter operatorn kommer ett nytt tal. När det är slut får operatorn verka på dessa tal, och så vidare.

I exemplet nedan ska vi tillåta parenteser och givetvis ta hänsyn till prioritetsreglerna.

Olika skrivsätt I *infix form*, den för oss naturliga, står operatorerna mellan operanderna. Parenteser används för att ange att prioritetsordningen är en annan än vad grundreglerna säger.

```
(2+3)*4+(4+6)/(3+2)*4
```

I prefix form står operatorerna före sina operander. Skrivsättet kallas också polsk notation efter polacken *Lukasiewics*.

```
*+35-45
```

Det är förstås viktigt att ett skrivsätt som detta är entydigt, det vill säga att det endast kan tolkas på ett sätt.

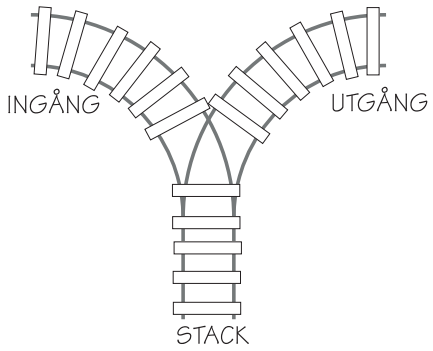
Så till sist det för datalogin viktigaste skrivsättet, *postfix* form eller *omvänd polsk notation* (RPN). Här står operatorerna efter sina operander.

```
35+45-*
```

Stack För att bestämma värdet av ett aritmetiskt uttryck skrivet i RPN används en eller flera stackar. En stack är en datastruktur på vilken man lagrar data enligt principen sist in först ut.

Algoritm För att lösa detta problem på ett bra sätt måste man först finna ett sätt att översätta ett vanligt uttryck från infix notation till postfix notation och därefter en algoritm för att bestämma det postfixa uttryckets värde.

Den första algoritmen, som har den kända datalogen *E W Dijkstra* som upphovsman, används i kompilatorer.



Figur 4.10:

Vi betecknar tecknen i uttrycket med $t_1, t_2 \ldots t_n$. Figur 4.10 av en spårväxel beskriver ganska bra algoritmens funktion. Tecknen t_i kan ses som järnvägsvagnar vilka kommer in från vänster i tur och ordning och så småningom ska föras ut till höger. Blindspåret fungerar som stack.

Algoritmen kan uttryckas på följande sätt:

- Repetera punkterna nedan tills ingången är tom
- Om t_i är en *operand* förs den direkt till utgången.
- Om t_i är en *västerparentes* förs den till stacken.
- Om t_i är en *högerparentes* överförs element från stacken till utgången, tills vänsterparentes påträffas. Båda parenteserna försvinner.
- Annars är t_i en *operator*. Det översta elementet i stacken betecknar vi t_j. För över symboler från stacken till utgången tills:
  - t_j har lägre prioritet än t_i
  - t_j är en västerparentes
  - stacken är tom.
- När ingången är tom töms sedan stacken genom att elementen förs till utgången.

Bestäm värdet för ett postfixt uttryck. För att så till sist bestämma värdet av det postfixa uttrycket kan man använda sig av följande algoritm:

1 `i=1`

2 Om t_i är en operator får den operera på de två översta elementen i stacken. Dessa ersätts sedan med resultatet. Öka sedan i med 1. Om $i \leq n$ utförs punkt 2 igen.

3 Om t_i är en operand stackar man den. Öka sedan i med 1. Om $i \leq n$ gå till punkt 2

4 Det värde som finns på stacken då $i > n$ är lika med uttryckets värde!

4.1.14 Programmet

```
1 char  stack[81];
2 float stack2[41];
3 int   antal=0,antal2=0;
```

H Två globala stackar och två räknare som håller reda på antalet element.

```
1 int prio(char c){
2   switch (c){
3     case '*': return 2; break;
4     case '/': return 2; break;
5     case '+': return 1; break;
6     case '-': return 1; break;
7     case '(': return 0; break;
8   }
```

H Prioritetsfunktionen, som tar emot en operator, `*/+-` eller en parentes och returnerar ett värde `0...2` allt beroende på operatorns prioritet

```
1 void push(char x){
2   antal++;
3   stack[antal]=x;
4 }
5 char pop(void){
6   return (stack[antal--]);
7 }
8 char top(void){
9   return (stack[antal]);
10 }
11 int tom(void){
12   return antal==0;
13 }
```

H Fyra funktioner som sköter en stack med tecken (`char`) som element.

```
1 void strchar(char s[],char c){
2   int p;
3   p=strlen(s);
4   s[p]=c;
5   s[p+1]='\0';
6 }
```

H En funktion vi behöver då vi ska bygga strängen som ska innehålla uttrycket på postfix form. Funktionen tillfogar ett tecken till en sträng.

```
1 void konvertera(char postfix[],char infix[]){
2   int k,klar;
3   char c;
4   for(k=0;k<strlen(infix);k++){
5     c=infix[k];
6     switch (c){
7       case '0': case '1':
8       case '2': case '3':
9       case '4': case '5':
10      case '6': case '7':
11      case '8': case '9': strchar(postfix,c); break;
12
13      case '*': case '/':
14      case '+': case '-' :
15      {
16        klar=0;
17        do{
18          if(tom())
19            klar=1;
20          else{
21            if(prio(c)<=prio(top()))
22              strchar(postfix,pop());
23            else
24              klar=1;
25          }
26        }while(!klar);
27        push(c);
28      }; break;
29      case '(': push(c); break;
30      case ')':
31        while (stack[antal]!='(')
32          strchar(postfix,pop());
33        pop();break;
```

```
34
35      }
36    }
37    while(!tom())
38      strchar(postfix,pop());
39  }
```

H Programmet går igenom tecken för tecken i infix och dem enligt reglerna. När funktionen är klar finns det infixa uttrycket på postfix form i strängen `postfix`

```
1  void push2(float x){
2    antal2++;
3    stack2[antal2]=x;
4  }
5  float pop2(void){
6    return stack2[antal2--];
7  }
```

H Två funktioner som hanterar en stack med element av typen `float`

```
1  float berakna(char postfix[]){
2    int k,t1,t2;
3    char c;
4    for(k=0;k<strlen(postfix);k++){
5      c=postfix[k];
6      switch (c){
7        case '0': case '1': case '2': case '3':
8        case '4': case '5': case '6': case '7':
9        case '8': case '9': push2(c-48); break;
10       case '*': push2(pop2()*pop2()); break;
11       case '+': push2(pop2()+pop2()); break;
12       case '/': push2(1.0/pop2()*pop2()); break;
13       case '-': {t1=pop2();t2=pop2();
14                  push2(t2-t1);} break;
15     }
16   }
17   return pop2();
18 }
```

H Här beräknas slutresultatet

10-13 Observera att i, till exempel, anropet av `push2(pop2()*pop2())` vet man inte med säkerhet i vilken ordning de två `pop2`-anropen utförs. Detta gör nu ingenting i just detta anrop av `push2` eftersom multiplikation är kommutativ. Värre är det vi division och subtraktion.

```
1  int main(void){
2    char postfix[80]="",infix[80];
3    do{
4      infix[0]='\0';
5      printf(": ");
6      gets(infix);
7      if(strlen(infix)>0){
8        konvertera(postfix,infix);
9        printf("%s\n",postfix);
10       printf("%.4f\n",berakna(postfix));
11     }
12   }while(strlen(infix)!=0);
13 }
```

H Huvudprogrammet

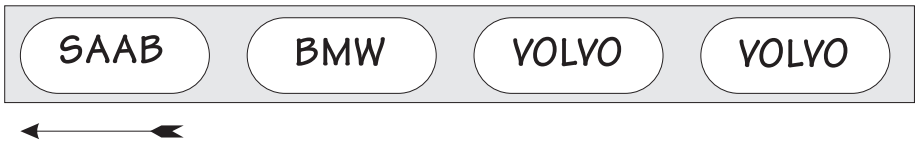
UPPGIFT 4.1

Vandringen Vi befinner oss i koordinaten (x, y) och vill hem till $(0, 0)$. Vår promenad är uppdelad i små promenader i rakt nordlig (8 steg), sydlig (3 steg), östlig (5 steg) och västlig (6 steg) riktning. Vilket är det minsta antalet små promenader vi behöver ta för att nå målet?

Skriv ett program som använder sig av tekniken *bredden först* och med följande dialog

```
Start x : 36
Start y : 27
Det krävs minst 15 små promenader
```

UPPGIFT 4.2
Bilkön



Figur 4.11:

Figuren visar en trång och tidsödande passage med en bilkö. Eftersom bara några få bilar hinner passera varje gång det blir grönt (i pilens riktning) finns det förare i slutet av kön som ångar sig och backar tillbaka.

På filen `trafik.dat` finns en beskrivning av hur kön förändras under en period. Varje rad innebär en förändring. Det finns tre olika typer av förändringar

- `ANLÄNDER`. En bil anländer till slutet av kön. Efter kommandot anges märket på den anländande bilen, exempelvis `ANLÄNDER SAAB`. Endast fyra märken kan förekomma `SAAB`, `VOLVO`, `BMW` eller `AUDI`.
- `KÖR`. Anger hur många bilar, från köns början, som hinner passera vid grönt ljus. Till exempel `KÖR 4`.
- `BACKAR`. Anger hur många bilar som backar bort från slutet av kön. Till exempel `BACKAR 3`

Skriv ett program som utifrån en tom kö går igenom filens kommandon och till slut skriver ut en lista av bilmärken som beskriver köns utseende framifrån.

Exempelfilen presenteras här i två kolumner:

```
ANLÄNDER AUDI          ANLÄNDER SAAB
ANLÄNDER AUDI          BACKAR 4
KÖR 1                  ANLÄNDER BMW
ANLÄNDER SAAB          ANLÄNDER VOLVO
ANLÄNDER AUDI          ANLÄNDER BMW
KÖR 1                  BACKAR 1
ANLÄNDER VOLVO         ANLÄNDER VOLVO
ANLÄNDER SAAB
```

Resultatet ges genom utskriften

```
SAAB
BMW
VOLVO
VOLVO
```

Filen inleds med en rad som anger antalet kommandon. De givna kommandona är alltid möjliga att utföra och filen kan aldrig innehålla fler än 100 kommandon. Mellan *kommando* och *argument* finns precis ett mellanslag. Argumenten för `BACKAR` och `KÖR` är alltid ett heltal i intervallet `1...9`. Alla bokstäver i filen är versaler.

UPPGIFT 4.3

Uppgift: Beräkning av aritmetiska uttryck. Bygg ut programmet `AritmUttryck` så att det klarar heltal > 9.

UPPGIFT 4.4

Uppgift: Hissen i lustiga huset igen. Lös problemet med hissen i lustiga huset igen. Denna gång genom att använda *bredden först*.

UPPGIFT 4.5

Uppgift: Taltriangeln. Ta reda på vilket djup i nivåer räknat programmet *Taltriangeln.c* klarar på 30 sekunder. Använd funktionen `clock` tillsammans med `CLK_TCK` i `time.h` för att fixa det hela. Du får 2 extra poäng om du tar upp tiden för flera nivåer och med hjälp av *Maple* anpassar en funktion till punkterna (tid,nivå).