Kapitel 2

Datastrukturer

2.1 Introduktion till datastrukturer

Att studera *datastrukturer* är att lära sig om metoder för att organisera data så att de passar databearbetningen. Detta ingår som en klassisk del i datalogin.

Å ena sidan har vi själva datorn – hårdvaran – med internt och externt minne. De elementära byggstenarna är *bytes* som man får tillgång till via heltalsadresser. Å andra sidan, när man skriver program, organiserar man data på en högre nivå, med hjälp av datastrukturer, i stället för att manipulera data på byte-nivå.

En datastruktur kan ses som en **container**, skapad för att lagra en samling objekt av en given typ. Dessa objekt kallar vi fortsättningsvis **element**. Med hjälp av variabler (**locators**) kan vi lokalisera (hitta fram till) data i datastrukturen. När ett element lagras i en container är det en locator som gör det möjligt att senare nå detta element.

Till varje datastruktur finns ett antal *operationer*. Operationer som till exempel lagrar nya element, tar reda på om ett visst element är representerat i containern eller returnerar antalet element i containern. Hur effektiv datastrukturen är avgörs av hur stor plats containern tar och hur lång tid operationerna behöver för att utföras. Till effektiviteten måste man nog också räkna hur lättanvänd strukturen är när man ska lösa praktiska problem.

2.1.1 Abstrakta datastrukturer

Datastrukturer är konkreta *implementationer* av *abstrakta datatyper*(ADT). Abstrakta datatyper är väl definierade datatyper med en tillhörande mängd operationer. En viss ADT kan implementeras (konkretiseras) på olika sätt. För en del av operationerna kan en viss implementation vara bättre (effektivare) än en annan.

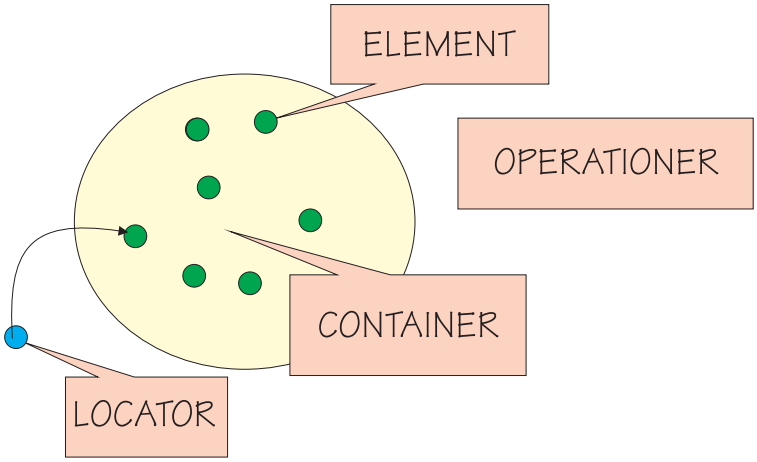
Står man inför ett större programmeringsprojekt bör man först välja vilka abstrakta datastrukturer man ska använda. Senare, under arbetet med att skriva programkoden kommer frågan hur de ska implementeras.

2.1.2 Grundläggande datastrukturer

Det finns förstås en repertoar av återkommande och mer eller mindre standardiserade abstrakta datastrukturer.

Den viktigaste kategorin är *sekvens*. Ofta vill man samla elementen i en följd, såsom i en lista eller tabell. Till denna hör *lista*, *stack* och *kö*. Några vanliga operationer är *insättning*, *sökning* och *borttagning*. Den vanligaste realiseringen av dessa datatyper är genom *array* eller *länkad lista*.

Andra abstrakta datatyper, mer eller mindre vanliga är *ordnat träd*, *binärt träd*, *lexikon*, *mängd*, *prioritetskö* och *graf*. Även textbehandling, geometri och grafik kräver ibland sina egna datastrukturer som kanske inte direkt tillhör någon av de ovan uppräknade.



Figur 2.1:

Till de fundamentala datastrukturerna hör

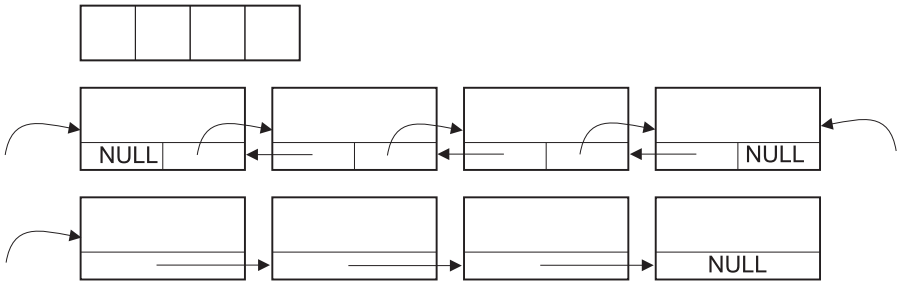
- SEKVENS (Sequence). En sekvens är en container som lagrar element i en bestämd ordning. De viktigaste operationerna är att lägga till och ta bort element på en given plats. STACK och KÖ är kanske de två mest omtalade datastrukturerna, som båda hör till denna kategori. I programspråket C implementeras dessa ofta som *länkade listor*, men arrayer kan duga lika bra.
- PRIORITETSKÖ (Priority Queue). Elementen i denna container tillhör ett *totalt ordnat universum*. Den viktigaste operationerna är att kunna ta bort och returnera det största (alternativt det minsta) elementet i containern. Strukturen har en naturlig användning är vid sortering och en effektiv implementation är genom en HEAP. Vi kommer senare i kursen, under kapitlet sortering, till HEAPSORT. Det finns en mängd varianter på HEAP som vi inte kommer att beröra i denna kurs: min-max heap, pagodas, deaps, binomial heaps och Fibonacci heaps.

- LEXIKON (Dictionary) Elementen i denna container tillhör också ett *totalt ordnat universum*. De viktigaste operationerna är *söka, lägga till* och `ta bort` element. Till varje element finns en unik nyckel. Här finns många sofistikerade metoder vid implementationen. Vi kommer att beröra HASHTABELLER. Flera metoder innehåller *balanserade träd*: AVL-TREE, RED-BLACK-TREE, 2-3-TREE, 2-3-4-TREE, WEIGHT-BALANCED-TREE, BIASED SERCH TREE, SPLAY TREE. Ofta handlar tillämpningarna om externt minne som till exempel databaser.

Sekvens

SEKVENS (S) är en container som lagrar element där ordningen är viktig

- Operationer:
  - SIZE(N) returnerar antalet element N i containern S
  - HEAD(c) tilldelar c en locator till första elementet i S
  - TAIL(c) tilldelar c en locator till sista elementet i S
  - LOCATERANK(r,c) tilldelar c en locator till det r:te elementet i S. Om $r < 0$ eller $r > N$, där N är antalet element i S, tilldelas c en NULL locator.
  - PREV(c',c") tilldelar c" en locator i S som föregår elementet med locator c'. Om c' är locator till det första elementet i S tilldelas c" en NULL locator.
  - NEXT(c',c") tilldelar c" en locator i S som följer efter elementet med locator c'. Om c' är locator till det sista elementet i S tilldelas c" en NULL locator.
  - INSERAFTER(e,c',c") sätter in elementet e i S efter elementet med locator c' och returnerar c", som är locator till e.
  - INSERTBEFORE(e,c',c") sätter in elementet e i S före elementet med locator c' och returnerar c", som är locator till e.
  - INSERTHEAD(e,c) sätter in elementet e först i S och returnerar c, som är locator till e.
  - INSERTHEAD(e,c) sätter in elementet e sist i S och returnerar c, som är locator till e.
  - INSERRANK(e,r,c) sätter in e på den r:te positionen och returnerar c, som är en locator till e. Om $r < 0$ eller $r > N$, där N är antalet element i S, tilldelas c en NULL locator.
  - REMOVE(c,e) tar bort och returnerar elementet e med locator c från S
  - MODIFY(c,e) byter ut elementet med locator c i S mot e



Figur 2.2: *Array, Dubbellänkad lista, Länkad lista*

- Kostnad när sekvensen implementeras som:

Operation	ARRAY	LÄNKAD LISTA	DUBBELLÄNKAD LISTA
SIZE	O(1)	O(1)	O(1)
HEAD	O(1)	O(1)	O(1)
TAIL	O(1)	O(1)	O(1)
LOCATERANK	O(1)	O(N)	O(N)
PREV	O(1)	O(N)	O(1)
NEXT	O(1)	O(1)	O(1)
INSERTAFTER	O(N)	O(1)	O(1)
INSERTBEFORE	O(N)	O(N)	O(1)
INSERTHEAD	O(N)	O(1)	O(1)
INSERTTAIL	O(1)	O(1)	O(1)
INSERTRANK	O(N)	O(N)	O(N)
REMOVE	O(N)	O(N)	O(1)
MODIFY	O(1)	O(1)	O(1)

N är antalet element i S

PRIORITETSKÖ (Priority Queue) Prioritetskö är en container med element i ett ordnat universum.

- Operationer:
  - SIZE(N) returnerar antalet element N i kön Q
  - MAX(c) returnera en locator c till det största elementet i Q
  - INSERT(e,c) placerar elementet e i kön Q och returnerar locator till e
  - REMOVE(c,e) tar bort och returnerer elementet e med locator c från Q
  - REMOVEMAX(e) tar bort och returnerar det största elementet e från Q
  - MODIFY(c,e) ersätter elementet med locator c med elementet e

- Kostnad när prioritetskön implementeras som:

Operation	OSORTERAD SEKVENS	SORTERAD SEKVENS	HEAP
SIZE	O(1)	O(1)	O(1)
MAX	O(N)	O(1)	O(1)
INSERT	O(1)	O(N)	O(log N)
REMOVE	O(1)	O(1)	O(log N)
REMOVEMAX	O(N)	O(1)	O(log N)
MODIFY	O(1)	O(N)	O(log N)

  N är antalet element i Q. En HEAP är binärt sökträd. Mer om det senare i kursen LEXIKON (Dictionary)

- Operationer:
  - SIZE(N) returnerar antalet element N i D
  - FIND(x,c) om D innehåller ett element med nyckeln x kommer c att vara en locator till detta element
  - LOCATEPREV(x,c) tilldelar c en locator till det element i D med den största nyckeln som är mindre än x
  - LOCATENEXT(x,c) tilldelar c en locator till det element i D med den minsta nyckeln som är sörre än eller lika med x
  - LOCATERANK(r,c) tilldelar c en locator till det r:te elementet i D
  - PREV(c',c") tilldelar c" en locator till det element i D med den största nyckeln som är mindre än nykeln till det element som har locator c'
  - NEXT(c',c") tilldelar c" en locator till det element i D med den minsta nyckeln som är större än nykeln till det element som har locator c'
  - MIN(c) tilldelar c en locator till elementet i D med den minsta nyckeln
  - MAX(c) tilldelar c en locator till elementet i D med den största nyckeln
  - INSERT(e,c) sätter in ett element e i D och returnerar dess locator c. Om det redan finns ett element e i D med samma nyckel returneras NULL locator.
  - REMOVE(c,e) Avlägsnar och returnerar elementet e med locator c från D
  - MODIFY(c,e) ersätter elementet med locator c i D med elementet e

- Kostnad när lexikonet implementeras som:

Operation	OSORTERAD SEKVENS DUBBELLÄNKAD LISTA	SORTERAD SEKVENS DUBBELLÄNKAD LISTA	SORTERAD SEKVENS ARRAY
SIZE	O(1)	O(1)	O(1)
FIND	O(N)	O(N)	O(log N)
LOCATEPREV	O(N)	O(N)	O(log N)
LOCATENEXT	O(N)	O(N)	O(log N)
LOCATERANK	O(N)	O(N)	O(1)
NEXT	O(N)	O(1)	O(1)
PREV	O(N)	O(1)	O(1)
MIN	O(N)	O(1)	O(1)
MAX	O(N)	O(1)	O(1)
INSERT	O(1)	O(N)	O(N)
REMOVE	O(1)	O(1)	O(N)
MODIFY	O(1)	O(N)	O(N)

N är antalet element i D

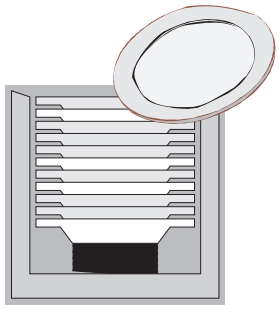
Operation	(A,B)-TREE	AVL-TREE	BUCKET ARRAY	HASH
SIZE	O(1)	O(1)	O(1)	O(1)
FIND	O(log N)	O(log N)	O(1)	O(N/M)
LOCATEPREV	O(log N)	O(log N)	O(M)	O(N+M)
LOCATENEXT	O(log N)	O(log N)	O(M)	O(N+M)
LOCATERANK	O(log N)	O(log N)	O(M)	O(N+M)
NEXT	O(log N)	O(log N)	O(M)	O(N+M)
PREV	O(log N)	O(log N)	O(M)	O(N+M)
MIN	O(1)	O(1)	O(M)	O(N+M)
MAX	O(1)	O(1)	O(M)	O(N+M)
INSERT	O(log N)	O(log N)	O(1)	O(1)
REMOVE	O(log N)	O(log N)	O(1)	O(1)
MODIFY	O(log N)	O(log N)	O(1)	O(1)

N är antalet element i D. Nycklarna är heltal i $[1, M]$ (A,B)-TREE och AVL-TREE ingår inte i kursen. BUCKET ARRAY är enkelt att implementera. Återkommer i samband med sortering. Även `Hash Tabell` kommer längre fram i kursen.

Stack

2.2 Stack

I figur 2.3 ser du den vanligaste metaforen för en stack – en tallrikshållare på en lunchrestaurang. Bilden vill visa att: *den tallrik vi lägger dit sist också är den vi först plockar bort.* En *stack* är en LIFO, *last in first out* eller hellre på svenska SIFU, *sist in först ut*



Figur 2.3: *En tallriksstack*

En programmerare sitter vid sin dator när telefonen ringer. I samma ögonblick som han svarar läggar han arbetet vid datorn på en ”stack”. Detta arbete får vila till telefonsamtalet är över. Under tiden som samtalet pågår kommer ett nytt samtal. Han ber personen från det första samtalet att vänta ett stund och kopplar in nästa. Han lägger det första telefonsamtalet på stacken och just nu har vår programmerare två ”jobb” på stacken. Precis då ställer sig en av hans kollegor i dörren och undrar om han ska med på lunch. För en kort stund hamnar alltså även det andra samtalet på stacken – när han svarar sin kollega: ”Ja, jag kommer snart”. Han återgår sedan till det andra samtalet. Han hämtar ned det från stacken. När han avslutar det samtalet hämtar han så ned det första från stacken. När även det är avslutat finns bara programmeringsjobbet kvar. Han blir påmind om lunchen. Ett jobb blir kvar på stacken i en timme till ...

2.2.1 Operationer

Vi har tidigare sagt att till en abstrakt datastruktur finns ett antal *operationer*. Först till de som är viktigast för en stack:

Push. *Push* lägger ett element *överst* på stacken. Denna operation är förstås alldeles nödvändig för en stack. För att vara säker på att man inte överskrider stackens kapacitet testar operationen så att stacken inte är full. Om stacken redan är full vet vi inte hur vi ska fortsätta och avbryter därför exekveringen.

Det är svårt att välja rätt nivå när vi ska beskriva dessa operationer. Först den högsta

tänkbara nivån.

Algorithm 2.2.1: PUSH(element)

Lägg **element** överst på stacken

Följande nivå är mer detaljerad och kanske kommer man för nära själva implementationen. Man känner på sig att *Stack* här är en array och *StackMax* är en övre gräns för hur många element stacken kan innehålla. Eftersom det finns andra sätt att konkretisera denna operation är denna nivå för låg.

Algorithm 2.2.2: PUSH(element)

if StackIndex $<$ StackMax

then { StackIndex $\leftarrow$ StackIndex $+ 1$
Stack[StackIndex] $=$ element

else { Felmeddelande
avbryt

Pop. *Pop* returnerar det *översta* elementet på stacken och tar samtidigt bort det från stacken. Under förutsättning att stacken inte är tom förstås. Även denna operation är också nödvändig för en normal stack. Om denna operation försöker utföras på en tom stack vet vi inte hur vi ska fortsätta och avbryter därför processen med ett felmeddelande.

Algorithm 2.2.3: POP()

Returnerar det översta elementet på stacken

IsEmpty. IsEmpty returnerar *true* om stacken är *tom* och *false* annars. Operationen är till för att klienten ska ha möjlighet att undvika att anropa *Pop* när stacken är tom, vilket skulle leda till att exekveringen avbryts.

Algorithm 2.2.4: ISEMPTY()

if Stacken är tom
  then return (TRUE)
  else return (FALSE)

IsFull. IsFull returnerar *true* om stacken är full och *false* annars. Denna operation är på liknande sätt som *IsEmpty* till för att undvika plötsliga avbrott. Är stacken redan full ska man inte anropa *Push*

Algorithm 2.2.5: ISFULL()

if Stacken är full
  then return (TRUE)
  else return (FALSE)

Här följer några mindre viktiga operationer som möjligtvis kan förekomma i samband med en stack.

- **Top** *Top* returnerar första värdet på stacken *utan att ta bort det*. Någon gång kanske man bara vill testa översta elementet på stacken, utan att ta bort det. Den här operationen kan man klara sig utan eftersom en *Pop*, följt av en *Push* av elementet igen, ger samma resultat.
- **InitStack** InitStack initierar stacken innan vi börjar använda den. För vissa implementationer, speciellt de som använder dynamiskt minne måste stacken först initieras (skapas).
- **FreeStack** Använder vi *InitStack* för att skapa vår stack så måste vi ha en operation, *FreeStack*, för att ta bort den från heapen när den inte ska användas längre.
- **StackDepth** För att få reda på hur många element som just nu finns på stacken.
- **IsOnStack** En operation som tar reda på om ett givet element finns på stacken. Operationen returnerar -1 om elementet inte finns. Annars returnerar funktionen platsen där elementet finns. Stacken lämnas orörd.

Från valet av operationer ovan kan diskussionen om *vem som har ansvaret* väckas. Ska operationerna omsättas till kod på ett sätt som gör klienten fri från ansvar. Det vill säga ska rutinerna som sköter stacken hålla reda på hur många element som finns där, se till att programmet inte kraschar genom tester i *Pop* och *Push*. Om klienten i stället håller reda på hur många element stacken innehåller så finns ingen risk för felaktiga anrop.

2.2.2 Implementation av en stack

Vi har nu definierat den abstrakta datastrukturen *stack*. Hur kan den då implementeras? Hur ska vi realisera den i vår programkod. Vi börjar med ett exempel:

EXEMPEL 2.1
I denna uppgift ska du skriva ett program som kan analysera parentesuttryck. Programmet ska avgöra om den inmatade strängen med parenteser är balanserad och korrekt. Strängen kan endast innehålla följande tecken

```
(  )  {  }  [  ]
```

Att ett parentesuttryck är *korrekt* betyder att det innehåller lika många *vänster-* som *höger-parenteser* av varje sort och att parenteserna dessutom kommer i rätt ordning.

Nedan följer några exempel

`(())`	Korrekt	`((()))`	Felaktigt
`[{()}]`	Korrekt	`[[[[[`	Felaktigt
`[[[((()))]]]`	Korrekt	`[({})]`	Felaktigt
`(([{}]))`	Felaktigt	`(((]))`	Felaktigt

Den inmatade strängen får högst innehålla 30 parenteser.

En mycket passande algoritm för att lösa problemet

Algorithm 2.2.6: PARENTESTEST(s)

```
INITSTACK()
for i ← 1 to LENGTH(s)
  do { if vänsterparentes
       then PUSH(s(i))
     { if högerparentes
       then { if ISEMPTY()
              then return (felaktig)
            { if POP() not matchar s[i]
              then return (felaktig)
if ISEMPTY()
  then return (korrekt)
  else return (felaktig)
```

Vi har använt oss av operationerna PUSH, POP, INITSTACK och ISEMPTY. Om vi väljer en array som stack, en array som kan lagra tecken (char) så kommer motsvarande funktioner att se ut så här.

```
1  char stack[30];
2  int plats=0;
3
4  void push(char t){
5     stack[plats]=t;
6     plats++;
7  }
8
9  char pop(void){
10    char t1;
11    plats--;
12    t1=stack[plats];
13    return t1;
14 }
15
16 bool isempty(void){
17   return (plats<0);
18 }
```

1–2 Vi tillåter oss två globala variabler här, själva stacken `stack` och `plats` som håller reda på var nästa lediga plats finns för ett ”stackat” element.

4–7 Lägger in det ”pushade” elementet i `stack` och ökar `plats`.

9–14 Minskar platsen, tar ut ett element från stacken, och returnerar elementet.

16–18 Returnerar *t*rue om `plats` är negativ, det vill säga stacken är tom.

H Vad är det som gör att man med dessa funktioner inte kan testa två parentesuttryck i samma programkörning? Med vilken tidigare nämnd operation kan man avhjälpa detta?

Det återstår en del jobb för att hålla reda på vad som är *vänsterparentes*, vad som är *högerparentes* och när de *matchar*. Men de lämnar vi. För den intresserade heter programmet `parentes.c`

2.2.3 Stackbibliotek

Även om dessa funktioner är korta och enkla att återskapa från minnet så är det otillfredsställande att skriva om dem varje gång man ska använda dem. Målet är därför att placera dem i en egen fil *stack.c* och att skapa en *stack.h* som kan inkluderas i början av klientprogrammet på vanligt sätt.

Här följer ett enkelt, första försök. Vi visar delar av de tre filerna *stack.h*, *stack.c* och *stacktest.c*

```
1 void push(char t);
2 char pop(void);
3 int isempty(void);
4 int isfull(void);
```

1–3 Ovan ser vi *stack.h*, som i detta enkla fall inte innehåller något annat än funktionsdeklarationer för fyra stackoperationer.

```
1 static char stack[30];
2 static int plats=0;
3
4 void push(char t){
5   stack[plats]=t;
6   plats++;
7 }
8 ...
```

1–2 Själva stacken deklareras tillsammans med variabeln `plats` som ska hålla reda på antalet. Båda dessa variabler deklareras som `static` vilket innebär att de inte är globala men inte kända utanför denna modul.

4–8 Sedan kommer funktionerna `push, pop, isempty` och *isfull* i en följd.

```
1 #include "stack.h"
2 #include <stdio.h>
3 void main(void){
4   ...
5   push('h');
6   c=pop();
7   ...
8 }
```

1 Lönen för vår möda – det räcker nu att skriva `#include "stack.h"` för att nå biblioteket `stack`

5–6 Länkaren hittar koden till `push` och `pop` i den obj-fil som skapas då *stack.c* kompileras.

Så här långt fungerar det hela om vi utvecklar programmet i ett projekt där dessa tre filer ingår.

UPPGIFT 2.1
Implementera enkel stack. (1) På hemsidan hittar du en `zip`-fil med länken *Enkel Stack (array)*, som innehåller filerna `main.c`, `stack.h` och `stack.c`. Skapa ett projekt som kan exekvera `main.c`.

2.2.4 Ännu generellare stack

I det stackbibliotek vi skrivit ovan kan elementen bara vara av typen *char*. Om vi vill använda en stack där elementen är av typ *int* eller kanske en av klienten definierad post, `struct`, så måste vi med den teknik vi hittills använt, skriva en uppsättning funktioner för varje datatyp. Om typen hos elementet är en post så kan vi ju aldrig förutsäga hur den kommer att se ut och måste därför skriva `stack.h` och `stack.c` för varje tillämpning.

2.2.5 Flera stackar samtidigt

Det är tänkbart att vi råkar ut för en situation där vi vill ha flera stackar igång samtidigt. Kan detta lösas med den kod vi skrivit ovan? Nej och det är uppenbart att när vi anpassar rutinerna till detta fall, så måste vi i alla funktioner lägga till en ny parameter, som anger *vilken* stack som avses. Det vi saknar här kan lösas med *objektorienterad programmering* i Java.

UPPGIFT 2.2
Implementera stack i Java. (3) I Java finns förstås redan stack implementerad. Trots det ska du här skriva en generell klass `Stack` med flera av de operationer vi nämnt ovan.

På filen `namn.dat` finns ett antal *förnamn* och vilken *fotbollsklubb* de hejar på – AIK, DIF eller HIF. Filen inleds med ett tal `n` som anger hur många personer den innehåller. Därefter följer `2n` rader, först förnamnet på en rad sedan klubben på nästa.

Använd den nyskrivna Java-klassen för att åstadkomma en utskrift där först namnen på alla AIK-are skrivs ut, följt av djurgårdarnas och hammarbyarnas namn.

2.2.6 Stacken som en länkad lista

.

```
1 struct objekt{
2   int nummer;
3   struct objekt *naesta;
4 };
5
6 typedef struct objekt objekt
7
8 objekt *start=NULL;
```

1–4 Vår första stack ska hantera heltal. Varje element i stacken skall vara av typen `objekt`. Strukturen `objekt` innehåller, i tillägg till heltalet, en pekare till nästa elementet på stacken.

6 Pekaren *start* håller reda på var stackens topp finns. Den pekar ut det sist pålagda elementet. Om `start` är `NULL` betyder det att stacken är tom.

```
1 bool tom(void){
2   return start==NULL;
3 }
```

Funktionen `tom` returnerar *true* om stacken är tom och *false* om det finns minst ett element. Ovan har vi ju sagt att stacken är tom om `start` är NULL.

```
1 void push(int nr){
2   objekt *ny;
3   ny=(objekt *)malloc(sizeof(objekt));
4   ny->nummer = nr;
5   ny->naesta=start;
6   start=ny;
7 }
```

Med hjälp av denna funktion lägger vi till element på stacken. I vardagslag använder man de svengelska ordet ”pusha” (skjuta på) för att lägga upp ett element på en stack. Ordet är just nu ett modeord – det är ofta man pushar för olika saker.

Funktionen tar emot ett heltal, men returnerar ingenting. Man räknar helt enkelt med att det kommer att lyckas och har därför ingen anledning att rapportera. Det enda som skulle kunna gå snett är att stacken inte rymmer det nya elementet. Misstänker man att detta kan inträffa är nog det bästa att ha en global variabel som noterar denna händelse. Detta för att hålla stackfunktionerna så rena som möjligt från ovidkommande parametrar.

2–4 `ny` är en pekare till ett `objekt`. Med `malloc` allokerar vi dynamiskt ett utrymme på heapen. Sen kopieras värdet på variabeln `nr` till `nummer`.

5–8 Detta element ska nu hamna överst i stacken. Om stacken är tom innan, har detta enda element ingen efterföljare. Pekaren `naesta` får då värdet `NULL` genom att sättas lika med `start` pekaren som är `NULL` om stacken är tom.

I annat fall kommer det tidigare första elementet att sjunka ner i stacken och det nya elementet har en efterföljare som tidigare pekades ut av `start`.

6 Till sist justeras `start`, som ju ska peka ut första elementet.

```
1 int pop(void){
2    int nr;
3    nr = start->nummer;
4    objekt *p = start;
5    start = start->naesta;
6    free(p);
7    return nr;
8 }
```

Funktionen `pop` hämtar ner element från stacken. Inga indata och ett heltal som utdata.

3–4 Funktionen tar i variabeln `nr` hand om värdet på heltalet till det första elementet på stacken. Pekaren `p` sätts till att peka på det första elementet på stacken. Denna pekare skall vi använda för att deallokera det minne som vi tidigare allokerade dynamiskt.

5 Det första elementet på stacken länkas sen ut ur listan genom att flytta `start`-pekaren så den pekar på nästa elementet på stacken.

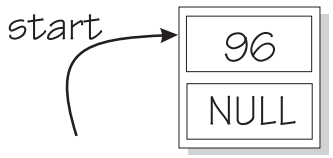
6 Utrymmet som vi allokerade dynamiskt med hjälp av `malloc` friges sen med hjälp av `free`. Vi använder här pekaren `p` som vi satt att peka på det första elementet på stacken.

7 Till sist returneras värdet på heltalet till det element som vi har tagit bort från stacken.

Om elementet är det sista på stacken kommer *start* att få värdet `NULL` i annat fall kommer `start` att peka ut elementets efterföljare. Programmeraren får själv förvissa sig om att stacken inte är tom, före anropet av `pop`, med hjälp av `tom`.

EXEMPEL 2.2

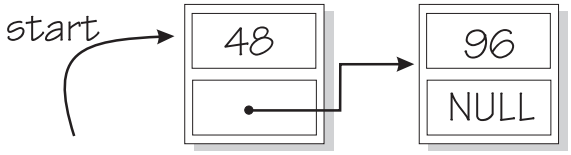
Ett program som utnyttjar de tre funktionerna för att bygga upp stacken, så som den ser ut i figurerna.



Figur 2.4: *Push anropas med ett element vars värde är 96. Så här ser stacken ut efter första anropet*

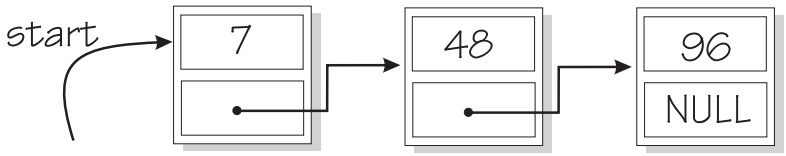
```
1 void main(void){
2    int nr;
3    nr=96; push(nr);
```

Figur 2.5: *Nu står ett element med värdet 48 på tur att "stackas"*



Figur 2.6: *Efter push(a) har talet 48 hamnat på stacken.*

```
1    nr=48; push(nr);
```



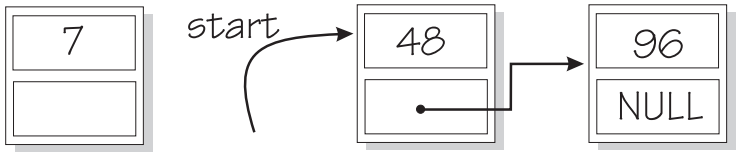
Figur 2.7: *På samma sätt hamnar där sedan ett element med värdet 7*

```
1    nr=7;  push(nr);
```

```
1    while(!tom()){
2       nr=pop();
3       printf("%d",nr)
4    }
5 }
```

Observera att vi i huvudprogrammet endast talar om vilka heltal vi vill lägga på stacken. Sen tar programmet själv hand om att skapa element av typen `objekt` som lägges på stacken.

1–4 I denna loop "poppas" ett element i taget av från stacken till stacken blir tom.



Figur 2.8: *Med hjälp av pop kan vi nu lyfta ned det översta elementet från stacken, det med värdet 7*

UPPGIFT 2.3

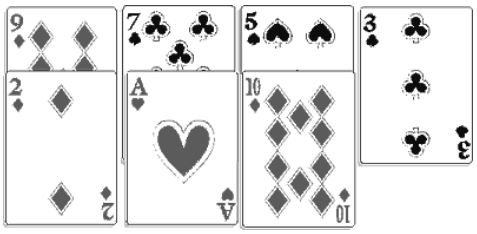
Implementera enkel stack som länkad lista. (1) På hemsidan hittar du en fil med länken *Enkel Stack (länkad lista)*, som innehåller filerna `main.c`, `stack.h` och `stack.c`. Skapa ett projekt som kan exekvera `main.c`. Skapa gärna en roligare tillämpning.

UPPGIFT 2.4

Sortering med hjälp av stackar. (2) På hemsidan hittar du en fil med länken *StackSort*, som innehåller filerna `main.c`, `stack.h`, `stack.c` och `tal.dat`. Skapa ett projekt som kan exekvera `main.c`. Sätt dig in i koden så att du kan redogöra för hur programmet fungerar.

2.3 Idioten

Många patienser brukar kallas ”Idioten”, antagligen för att de på ett eller annat sätt gör skäl för namnet. Det här är emellertid den riktiga ”Idioten”, vilket dock inte på något sätt innebär att den är dummare än alla andra. Tvärtom, det är en trevlig patiens med fler kombinationsmöjligheter än man kanske från början kan upptäcka. I *Dostojevskis* berömda roman ”Bröderna Karamasov” lägger Grusjenka en patiens som kallas ”Idioten”. Kanske var det den här.



Figur 2.9: ♢ 2 *och* ♢ 9 *kan kastas. Därefter kan* ♡ E *flyttas ner. Då blir* ♣ 7 *fri och* ♣ 3 *kan kastas, varefter* ♢ 10 *kan flyttas till en tom hög*

2.3.1 Så går den till

En vanlig kortlek med 52 kort. Man börjar med att lägga ut fyra kort med framsidan upp i en vågrät rad (se figur 2.9).

Finns bland dessa fyra kort två eller flera av samma färg, har man rätt att plocka bort de lägsta korten i denna färg. När det inte går att plocka bort fler kort, lägger man ut en ny rad med fyra kort ovanpå de förra. Esset är högsta kortet. Om några av de översta korten är av samma färg, får man återigen plocka bort de lägsta, tills det bara ligger olikfärgade kort kvar. Var gång det uppstår ett tomrum, får man lägga dit vilket som helst av de översta korten från de andra högarna.

På det sättet fortsätter man att lägga ut rader om fyra kort i taget, kastar de lägre i samma färg och flyttar kort till tomma platser. Patiensen har ”gått ut”, när endast de fyra essen ligger kvar på bordet.

På hemsidan finns programmet *Patiens demo*, med vilket du kan ’träna’ tills du förstår vad den går ut på. Du lägger ut nya kort genom att klicka på knappen *Lägg ut*, tar bort ett kort med *vänster* musknapp och flyttar ett kort till en tom plats genom att använda *höger* musknapp två gånger.

2.3.2 Ett program som lägger patiensen

Nedan finner du, med kommentarer, ett nästan färdigt program, som lägger patiensen 300 000 gånger och sedan skriver ut antalet lyckade försök.

Läs igenom kommentarerna och sätt dig in i programmet så pass att du kan lösa uppgifterna! En programfil som heter *patiens.c*, finns för övrigt innehållande koden i det skick du ser den här.

2.3.3 main

På *topdown*-vis börjar vi med huvudfunktionen, föredömligt kort. Det är svårt att se att de funktionsanrop, som finns här leder till närmare 150 rader kod.

```
1  #define ANTALFORSOK 300000
2  int main(void){
3    int lek[53],antal=0;
4    int forsok;
5
6    srand(time(0));
7    for(forsok=1;forsok<=ANTALFORSOK;forsok++){
8      blanda(lek);
9      if (lagg_patiens(lek))
10       antal++;
11   }
12   printf("%d lyckade försök \n",antal);
13 }
```

Funktionen `main` inleds med deklaration av en kortlek, `lek`, en variabel `forsok`, som håller reda på hur många försök vi har gjort och en variabel `antal`, som håller reda på antalet lyckade försök.

Rutinen inleds med `srand(time(0))`, för att slippa få samma resultat efter varje körning. `for`-loopen kommer sedan att exekveras i 300 000 varv. I varje varv börjar vi med att blanda kortleken.

Funktionen `lagg_patiens` anropas sedan med den nyblandade kortleken. Om funktionen skickar tillbaka ett tal $\neq 0$ (då alltid 1) har patiensen lyckats och räknaren `antal` ökas med ett.

2.3.4 Deklarationer

Följande deklarationer väljer vi att göra på global nivå – posttypen `kort` som innehåller kortets färg och valör. `kort` som är en array av pekare. Varje cell kommer att tjänstgöra som en stack. En stack för varje korthög.

```
1  struct kort{
2    int valor,farg;
3  };
4  struct kortelement{
5    struct kort kort;
6    struct kortelement *nasta;
7  };
8  struct kortelement *start[4]={NULL,NULL,NULL,NULL};
9  int nstack[4]={0,0,0,0};
```

1–3 En posttyp `kort` deklareras, där förstås kortets *färg* och *valör* är viktiga data.

4–7 En posttyp `kortelement` omsluter `kort` och innehåller en pekare `nasta` av typ `struct kortelement`.

8 Här deklareras, globalt, en *array med fyra pekare* (övertyga dig om det). Elementen i arrayen ska innehålla en pekare till översta elementet i var och en av de fyra stackarna (högarna) eller är `NULL` (då stacken är tom).

9 Arrayen `nstack` håller reda på hur många kort det finns i varje hög.

2.3.5 Funktioner för att hantera de fyra stackarna

Dessa funktioner, `tom, push` och `pop`, har vi redan stött på i inledningen av detta kapitel.

```
1  int tom(int nr){
2     return (start[nr]==NULL);
3  }
4
5  void push(struct kort p,int nr){
6     struct kortelement *ny;
7     ny=(struct kortelement *)malloc(sizeof(struct kortelement));
8     nstack[nr]++;
9     ny->kort=p;
10    ny->nasta=start[nr];
11    start[nr]=ny;
12 }
```

```
1  struct kort pop(int nr){
2     struct kort p;
3     struct kortelement *tmp;
4     p=start[nr]->kort;
5     tmp=start[nr];
6     nstack[nr]--;
7     start[nr]=start[nr]->nasta;
8     free(tmp);
9     return p;
10 }
11
12 int length(int nr){
13   return nstack[nr];
14 }
15
16 struct kort overst(int nr){
17 ...
18 }
```

16–18 Vid flera tillfällen, längre ned i detta program behöver vi titta på det översta elementet i en stack, utan att för den skull plocka bort det. Visst skulle det kunna ordnas genom att först anropa `pop` och sedan `push`, men bättre är nog en funktion `overst`, som returnerar en kopia av stacktoppen.

Funktionen ska ta emot ett tal `0...3` som anger vilken stack som avses och sedan returnera en post av typ `korttyp`, som är en kopia av det översta kortet på högen. Om stacken är tom ska ett kort med valören och färgen -1 returneras, (viktigt, för att det ska passa in i det övriga programmet).

2.3.6 lagg_patiens

Funktionen *lagg_patiens* är programmets hjärta.

```
1  int lagg_patiens(int lek[]){
2     int n=0,omgang,nr,hog;
3     struct kort p;
4     for(omgang=1;omgang<=13;omgang++){
5        for(nr=0;nr<4;nr++){
6           n++;
7           p.valor=(lek[n]-1)%13;
8           p.farg=(lek[n]-1)/13;
9           push(p,nr);
10       }
11       while (samma_farg(&hog)){
12          pop(hog);
13          hog_tom();
14       }
15    }
16    return lyckat();
17 }
```

4–15 Eftersom leken innehåller 52 kort och man lägger ut 4 kort åt gången, ser `for`-loopen till att det görs 13 utlägg.

5–10 Denna loop gör fyra varv – ett för varje kort i ett utlägg. Variabeln `n` håller reda på vilket kort i leken som står på tur.

Valör och färg bestäms utifrån kortets ordningsnummer. Kortnummer 1 till 13 kommer alla att få färg 0 och korten 40 till 52 får färg 3. Kortnumren $1, 14, 27, 40$ får valören 0 och kommer att motsvara lekens tvåor. På samma sätt kommer kortnumren $13, 26, 39, 52$ att få valören 12, som motsvarar essen.

När `p` har fått sina värden läggs den på rätt hög med funktionen `push`.

11–14 När korten är utlagda är det dags att ta bort de som går. Funktionen `samma_farg` returnerar 1 om det finns två högar med samma ”toppfärg”. I variabeln `hog` finns numret på den hög, i vilken kortet, som ska tas bort ligger.

12 Med `pop` för rätt hög försvinner kortet.

13 Möjligtvis kan nu högen `hog` blivit tom. Detta måste kontrolleras och eventuellt åtgärdas med `hog_tom`.

16 När vi kommit hit har alla kort lagts ut. Funktionen `lyckat` kontrollerar att de fyra högarna nu innehåller fyra ess på toppen och inget annat. Från `lyckat` returneras 1 eller 0.

2.3.7 lyckat

Funktionen som ska avgöra om patiensen lyckats.

```
1  int lyckat(void){
2     int hog,k;
3     struct kort p;
4     for(hog=0;hog<4;hog++){
5        p=overst(hog);
6        if (p.valor!=12){
7           kvar();
8           return 0;
9        }
10    }
11    k=kvar();
12    return k==4;
13 }
```

4–10 Så fort ett kort som inte är ett ess påträffas avslutas `lyckat` och returnerar `0`.

7–8 När man hamnar här har patiensen misslyckats och högarna måste tömmas inför nästa försök, innan talet `0` returneras.

11 Man kan bara hamna här när alla toppkort är ess. Detta räcker dock inte för ett lyckat försök. Funktionen `kvar` räknar därför det totala antalet kort på högarna. Om detta tal är 4 – ja då har patiensen ”gått ut” och funktionen returnerar `1`

2.3.8 kvar

Funktionen räknar hur många kort det finns kvar på de fyra högarna sammantaget och anropas av funktionen *lyckat*. Är det fyra stycken kan patiensen ha ”gått ut”.

```
1  int kvar(void){
2     ...
3  }
```

Funktionen `kvar` har inga indata men returnerar det sammanlagda antalet kort på de fyra högarna. Det är inte bara det att vi är intresserade av detta antal, dessutom vill vi rensa högarna inför nästa försök.

2.3.9 blanda

För att programmet ska ge en rättvis utvärdering av chanserna för att patiensen ska gå ut måste ”kortleken” blandas på ett korrekt sätt. I ett senare kapitel ska vi titta närmare på algoritmer av detta slag.

```
1  void blanda(int lek[]){
2     int k,plats,tmp;
3
4     for(k=1;k<=52;k++)
5        lek[k]=k;
6     for(k=1;k<=52;k++){
7        plats=k+rand()%(53-k);
8        tmp=lek[k];
9        lek[k]=lek[plats];
10       lek[plats]=tmp;
11    }
12 }
```

Här konstaterar vi bara att variabeln `lek` är en array, som innehåller 53 celler (vi använder inte 0:an)

Först fylls kortleken på med alla korten i ordning från 1 till 52 och sedan sker själva blandningen, som innebär att slumpmässigt utvalda kort byter plats i arrayen.

2.3.10 samma_farg

Vilka högar har samma färg på toppkorten, om det nu finns några högar med samma toppfärg över huvud taget.

```
1  int samma_farg(int *hog){
2     int hog1,hog2;
3     struct kort k1,k2;
4     for(hog1=0;hog1<3;hog1++){
5        k1=overst(hog1);
6        for(hog2=hog1+1;hog2<4;hog2++){
7           k2=overst(hog2);
8           if(k1.farg==k2.farg && k1.farg!=-1){
9              if(k1.valor>k2.valor)
10                *hog=hog2;
11             else
12                *hog=hog1;
13             return 1;
14          }
15       }
16    }
17    return 0;
18 }
```

Denna rutin har som uppgift att ta reda på om det finns två kort med samma färg högst upp på de fyra högarna.

En rutin med en dubbel `for`-loop, som hämtar ner kopior av av två stackars översta kort med hjälp av funktionen `overst` till `k1` och `k2`, och jämför dem.

Om korten har samma färg returnerar funktionen talet `1`. Dessutom får referensvariabeln `hog` ett värde, som anger numret på den hög ifrån vilken ett kort ska kastas.

8 Varför ingår `k1.farg!=-1` i villkoret? Hur många gånger testas villkoret i 8?

9–12 Övertyga dig om att det blir rätt hög från vilken man kastar ett kort.

2.3.11 hog_tom

Denna funktion tar reda på om det finns någon tom hög. I så fall flyttas toppkortet med högsta valören till denna hög.

```
1  void hog_tom(void){
2     int hog,fhog;
3     struct kort p;
4     for(hog=0;hog<4;hog++){
5        if(tom(hog)){
6           fhog=hogsta_topp();
7           if(fhog>-1){
8             p=pop(fhog);
9             push(p,hog);
10          }
11       }
12    }
13 }
```

Funktionen `hogsta_topp` returnerar numret på den hög i vilken det högsta kortet ligger och `hog` anger den tomma högen. Med en `pop` och en `push` är flyttningen avklarad.

2.3.12 hogsta_topp

Funktionen går helt enkelt igenom de fyra högarna och tar reda på vilken hög, som har högsta toppkortet. Om denna maxvalör finns i flera högar kommer den hög med lägsta numret att väljas.

```
1  int hogsta_topp(void){
2     int hog,hogmax=-1,max=-1;
3     struct kort p;
4
5     for(hog=0;hog<4;hog++){
6        p=overst(hog);
7        if(p.valor>max && length(hog)>1){
8           hogmax=hog;
9           max=p.valor;
10       }
11    }
12    return hogmax;
13 }
```

Här finns möjlighet till förbättringar. Som programmet fungerar nu flyttar den det högsta kortet till den tomma högen. Även om man inte tar reda på ordningen hos de oförbrukade korten kan man säkert ofta göra ett intelligentare val. Bara det att jag inte vet hur! Har du några idéer?

2.4 Programmeringsuppgifter

Tanken är nu att du ska arbeta med detta program i flera etapper, men det är också fritt fram att implementera idéerna i Java.

UPPGIFT 2.5
Fullfölj programmet. (3) Hämta in programmet `patiens.c` från hemsidan och fullfölj de två funktionerna `overst` och `kvar`.

UPPGIFT 2.6
Uppskatta sannolikheten. (1) Uppskatta sannolikheten att patiensen ska ”gå ut” genom att köra programmet ett par gånger.

UPPGIFT 2.7
Hur många kort blir kvar?. (2) Gör de tillägg i programmet som krävs för att kunna svara på frågan *Vilket är det vanligaste antalet kort som blir kvar i högarna efter en läggning av Idioten?*

UPPGIFT 2.8
En bättre strategi. (4) Försök att finna ett bättre sätt att flytta kort till tomma högar.