Kapitel 5

Träd

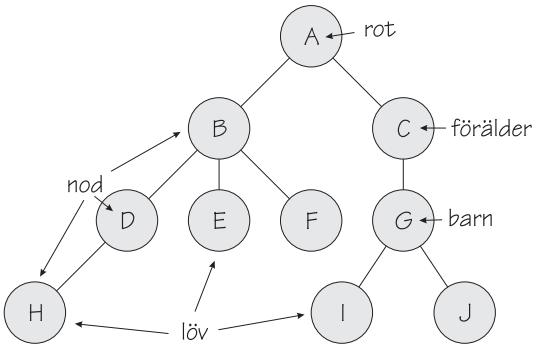
5.1 Definition av träd

I figur 5.1 ser du ett *träd*, sådant det avbildas i datalogin. Till skillnad från vanliga träd har ”dataträden” *roten* högst upp. Varje ring i figuren är ett *element* i trädet, som kallas *nod*. Varje nod utom roten har precis en föregångare – *förälder* (fader). En efterföljare till en förälder kallas *barn* (son). En nod som saknar barn kallas *löv*. Det vi från ”anfadern” och nedåt skulle kunna kalla generationer kallas här *nivåer*. Trädets *höjd* bestäms av antalet nivåer. Vi bestämmer att trädets rot finns på nivå 0.

Ett träd är en graf där man endast kan ta sig från en given nod till en annan på precis ett sätt. Detta betyder att ett träd saknar loopar. Trädets storlek bestäms av antalet noder. Ett träd med n noder har exakt $n - 1$ bågar.

Utgår vi från trädets rot kan man definiera ett träd som: *Ett träd består av en rotnod och ett ändligt antal underträd (subtrees).*

En mängd av rotade träd kallas en skog (forest).



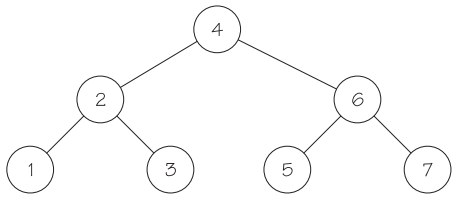
Figur 5.1: *Ett träd, sådant det presenteras i datalogin*

Här ska vi nu studera *binära träd*, mer specifikt *binära sökträd*.

5.2 Binärt sökträd

Ett *binärt träd* är ett träd där en nod kan ha högst två barn. Ett *vänsterbarn* och ett *högerbarn* eller lite mer matematiskt: Ett *binärt träd* är antingen tomt eller består av en *rotnod* och två *underträd* (subtree), som kallas *vänster underträd* och *höger underträd*

Ett *binärt sökträd* är ett binärt träd som är *ordnat* efter en given sorteringsordning. Varje nod har en nyckel, efter vilken sorteringen sker. Anledningen till att man vill hålla noderna i en datastruktur sorterade, är förstås att man snabbt ska kunna finna dem då nyckeln är given.



Figur 5.2: *Ett binärt sökträd ordnat efter definitionen nedan*

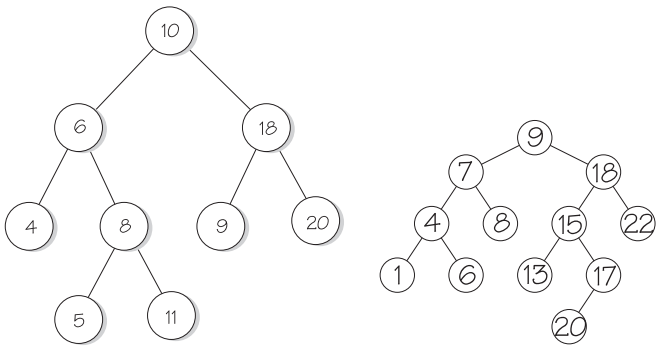
Ett exempel på ett sätt att ordna ett binärt sökträd, vilket också är det vanligaste sättet.

- Nyckeln till det vänstra barnet är mindre än eller lika med förälderns nyckel.
- Nyckeln till det högra barnet är större än eller lika med förälderns nyckel.
- För alla underträd där rotnoden har en nyckel med värdet X har nycklarna i det vänstra underträdet ett värde $\leq$ X och nycklarna i det högra underträdet $\geq$ X.

UPPGIFT 5.1
Definitioner. Utan att definiera alla ord, kan du säkert klara dessa frågor vad gäller trädet i figure 5.1

a) Vilket värde har trädets rot?
b) Vilka barn har G?
c) Vilken förälder har D?
d) Vilka avkomlingar har C?
e) Vilka syskon har F?
f) Vilka noder tillhör tredje nivån?
g) Vilken höjd har trädet?
h) Vilka noder är löv?

UPPGIFT 5.2
Inga binära sökträd. Varför är träden i figuren nedan inte binära sökträd. Vi antar att talen talen utgör nodernas nycklar?



5.2.1 Operationer

Det är just *binära sökträd* som vi nu ska betrakta som en ADT. Som vanligt ställer vi oss inledningsvis frågan om vilka operationer som ska finnas i vårt gränssnitt.

Operation	**Förklaring**
SearchTree	Söker i trädet
InsertTree	Sätter in nod i trädet
DeleteTree	Tar bort nod från trädet

5.2.2 Implementation

Precis som för *lista, kö* och *stack*, ska vi implementera binära sökträd dynamiskt. Ett passande utseende för en nod i ett binärt sökträd kan vara

```
1 struct objekt{
2   char nyckel[10];
3   struct objekt *left,*right;
4 };
5
6 typedef struct objekt objekt;
```

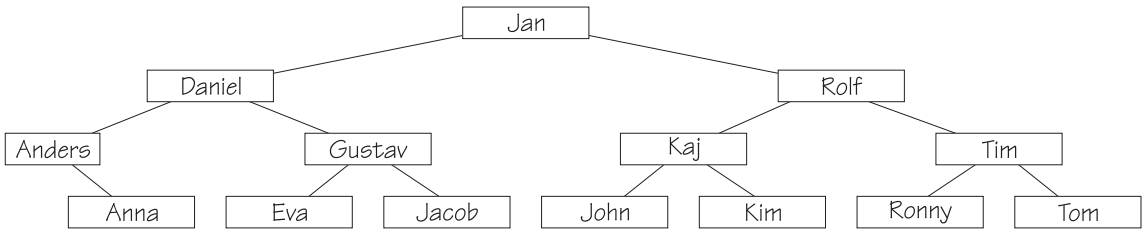
> `nyckel` är här en textsträng, men kan förstås lika väl vara ett heltal, ett flyttal, eller en annan struct. `left` och `right` är pekare till barnen. Om båda dessa är `NULL` innebär det att noden är ett löv.

5.2.3 Traversera

Att besöka alla noder i ett träd kallas att *traversera* trädet. När vi vill söka efter en speciell nyckel i ett binärt sökträd är det förstås viktigt att vi har en algoritm som vi vet kommer att söka igenom hela trädet om det nu behövs. Dessutom, eftersom trädet är ordnat, kan vi kräva att algoritmen söker i den del av trädet där nyckeln kan finnas. Man skiljer på tre olika *traverseringsordningar*:

- **Inorder**. Besök först *trädets vänstra del*, sedan *noden själv* och sist *trädets högra del*.
- **Preorder**. Besök först *noden själv*, sedan *trädets vänstra del* och sist *trädets högra del*.
- **Postorder**. Besök först *trädets vänstra del*, sedan *trädets högra del* och sist *noden själv*.

Utgår vi från följande binära träd kommer de tre metoderna att ge följande ordningar:



Figur 5.3: *Ett binärt sökträd sorterat efter förnamn*

- **Inorder:** Anders Anna Daniel Eva Gustav Jacob Jan John Kaj Kim Rolf Ronny Tim Tom
- **Preorder:** Jan Daniel Anders Anna Gustav Eva Jacob Rolf Kaj John Kim Tim Ronny Tom
- **Postorder:** Anna Anders Eva Jacob Gustav Daniel John Kim Kaj Ronny Tom Tim Rolf Jan

Innan vi tittar närmare på hur ett binärt träd kan byggas upp, utifrån givna data, preseterar vi några operationer på ett befintligt träd. Följande funktion ger en utskrift av noderna i *postorder* ordning:

```
1 void postorder(objekt *start){
2   objekt *x;
3   x=start;
4   if(x!=NULL){
5     postorder(x->left);
6     postorder(x->right);
7     printf("%s ",x->nyckel);
8   }
9 }
```

En rekursiv funktion som startar i trädets rot. Det är ju den adress som *start* har.

Så länge man kan ta sig åt vänster i trädet så följer man den vägen. När detta inte längre är möjligt börjar man gå åt höger i stället. När inte heller det är möjligt skrivs namnet på aktuell nod ut.

Jämför namnlistan ovan med funktionen och förvissa dig om att du förstår arbetssättet.

UPPGIFT 5.3

Skriv funktioner för inorder och preorder. Hämta in programmet `BinartTrad3.c` och kör det. På datafilen `bintrad3.dat` finns samma förnamn som de i figuren. Programmet ger också en utskrift vars ordning motsvarar *postorder*. Lägg till två nya funktioner *inorder* och *preorder*, så att dessa ger motsvarande ordningar.

Vilken av ordningarna tycks vara den mest användbara på det givna trädet?

UPPGIFT 5.4

Vilken ordning ska namnen, i figur 5.3 ha på filen, för att de ska komma ut i bokstavsordning då traverseringsordningen är:

a) Inorder

b) Preorder

c) Postorder

5.2.4 Maximum och Minimum.

I fortsättningen bestämmer vi oss för att vi alltid traverserar ett binära sökträd med metoden *inorder*. Två ”hjälpfunktioner” `minimum` och `maximum`. Det minsta värdet i ett binärt sökträd finner vi genom att gå åt vänster, välja vänsterbarnet, så länge det finns sådana. Det är inte överraskande att vi finner elementet med den största nyckeln genom att välja högerbarn så länge sådana finns. Tänk på att dessa funktioner inte bara fungerar då vi startar från roten. Eftersom varje nod kan ses som roten till ett underträd fungerar alltid dessa funktioner. Från figur 5.3 ser vi att `minimum` med utgångspunkt från *Rolf* är *John* och att `maximum` med utgångspunkt från *Daniel* är *Jacob*.

Vi har implementerat funktionen *minimum* rekursivt, medan funktionen *maximum* är implementerad icke-rekursivt, bara som en jämförelse.

```
1  objekt* minimum(objekt* x){
2     if(x == NULL)
3        return NULL;
4     else if(x->left == NULL)
5        return x;
6     else
7        return minimum(x->left);
8  }
9
10 objekt* maximum(objekt* x){
11    if(x != NULL)
12       while(x->right != NULL)
13          x = x->right;
14    return x;
15 }
```

1–8 Indata till funktionen är adressen till trädets rot. Vi kollar först om trädet är tomt, i så fall returnerar vi `NULL`. Annars går vi åt vänster så länge det är möjligt. När vi inte kommer längre åt vänster har vi hittat den minsta noden. Funktionen returnerar en pekare till den noden.

10-15 Kollar först om trädet är tomt och hittar sen det största värdet genom att välja högerbarn så länge sådana finns. Funktionen returnerar en pekare till noden med det största värdet.

5.2.5 Sökning

Att söka i en datamängd är den största anledningen till att bygga binära sökträd. Kom ihåg att vi bestämt att det är *inorder* som är vår traverseringsordning.

```
1  objekt *SearchTree(objekt *x,char namn[]){
2    if(x==NULL) return NULL;
3    if(x!=NULL && strcmp(x->nyckel,namn)==0)
4      return x;
5    if(x!=NULL)
6      if (strcmp(namn,x->nyckel)<0)
7        x=SearchTree(x->left,namn);
8      else
9        x=SearchTree(x->right,namn);
10   return x;
11 }
```

1 Indata är adressen till trädets rot och den nyckel man söker efter. Funktionen returnerar adressen till elementet med given nyckel. Om nyckeln inte kan återfinnas returnerar funktionen `NULL`.

Funktionen, som är rekursiv, måste så småningom hitta namnet den söker efter eller så blir alla fortsättningar ”blockerade” av `NULL`.

6 Funktionen jämför den sökta nyckeln med nyckeln i aktuell nod och väljer den fortsatta vägen efter resultatet.

5.2.6 Insättning

Här ett förslag på hur man kan bygga de binära sökträd som vi redan använt oss av. Först följer här en iterativ funktion för operationen `InsertTree`.

```
1  objekt *InsertTree2(objekt *start, char* ny){
2     objekt *x,*y,*z;
3
4     z=(objekt *) malloc(sizeof(objekt));
5     strcpy(z->nyckel, ny);
6     z->left=NULL;
7     z->right=NULL;
8     y=NULL;
9     x=start;
10    while(x!=NULL){
11       y=x;
12       if (strcmp(z->nyckel,x->nyckel)<0)
13          x=x->left;
14       else
15          x=x->right;
16    }
17    if(y==NULL)
18       start=z;
19    else{
20       if(strcmp(z->nyckel,y->nyckel)<0)
21          y->left=z;
22       else
23          y->right=z;
24       }
25    return start;
26 }
```

1 Indata till funktionen är adressen till trädets rot samt elementet som ska sättas in. I och med att detta är ett träd där nycklarna är textsträngar, blir elementet som skall sättas in av typen `char*`, men detta ändras naturligtvis om trädet har andra nycklar. Eftersom trädet ska fortsätta att var sorterat efter insättningen är det viktigt var den nya noden hamnar. Noden kommer alltid att sättas in som ett löv i trädet.

4–5 Plats bereds på heapen för den nya noden och värdet kopieras över.

6–7 Eftersom noden själv är ett löv ska `left-` och `right`-pekarna vara `NULL`.

8–9 `y` är en släppekare som pekar ut fadern till den aktuella noden.

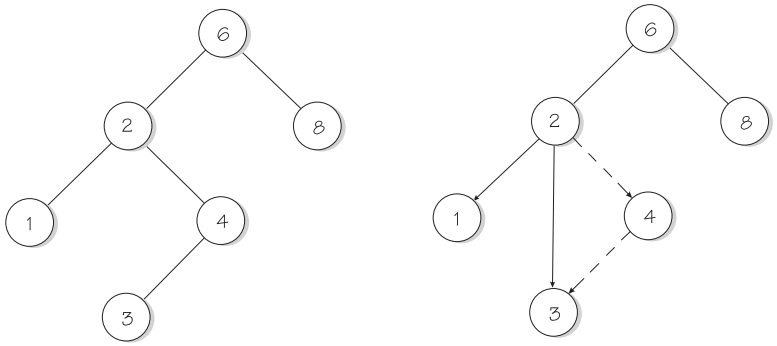
10–16 I `while`-loopen söker vi upp, med samma teknik som för sökandet, den plats där noden ska placeras.

17–25 Så återstår att uppdatera pekarna hos fadern till den insatta noden. Om det skulle vara rotnoden som uppdateras kommer trädet att få en ny adress som förstås måste returneras.

Så den rekursiva varianten, som är kortare, men är den effektivare?

```
1  objekt *InsertTree(objekt *p, char* ny){
2    objekt *z;
3
4    if(p==NULL){
5      z=(objekt *) malloc(sizeof(objekt));
6      strcpy(z->nyckel, ny);
7      z->left=NULL;
8      z->right=NULL;
9      return z;
10   }
11   else
12     if(strcmp(ny, p->nyckel)<0)
13       p->left=InsertTree(p->left,ny);
14     else
15       p->right=InsertTree(p->right,ny);
16     return p;
17 }
```

`p` blir så småningom `NULL` och när det inträffar initieras det nya elementet. Det nya elementets adress returneras och kommer vid ”återgången” av rekursionen att uppdatera rätt pekare hos fadern.

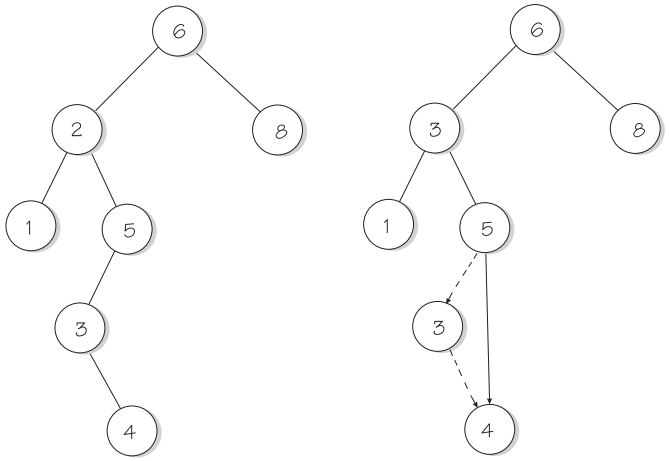


Figur 5.4: *Borttagning av en nod (4) med ett barn*

5.2.7 Borttagning

Den mest komplicerade av de tre operationerna är `DeleteTree`.

- Om noden, som skall tas bort, är ett löv är borttagning enkel. Noden kan tas bort på en gång. Förälderns pekare `NULL`-ställs
- Om noden, som skall tas bort, har *ett* barn kan noden tas bort efter att nodens förälder justerar sin pekare så att den pekar direkt på barnbarnet (se figur 5.4)
- Om noden, som skall tas bort, har *två* barn blir det mer komplicerat (figur 5.5). Den vanligaste strategin är att ersätta värdet till den nod som skall tas bort med värdet på den minsta noden i det högra underträdet. Sen kan man rekursivt ta bort den minsta noden i det högra underträdet. Eftersom den minsta noden i det högra underträdet inte kan ha ett vänsterbarn blir den sista borttagningen enkel.



Figur 5.5: *Borttagning av en nod (2) med två barn*

```
1  objekt* DeleteTree(objekt *x, char* namn){
2    objekt *tmp;
3    if(x==NULL)
4      abort();
5    if (strcmp(namn,x->nyckel)<0)
6      x->left = DeleteTree(x->left,namn);
7    else if(strcmp(namn,x->nyckel)>0)
8      x->right = DeleteTree(x->right,namn);
9    else if(x->left && x->right){
10     tmp= x->right;
11     while(tmp->left!=NULL)
12       tmp=tmp->left;
13     strcpy(x->nyckel, tmp->nyckel);
14     x->right = DeleteTree(x->right, x->nyckel);
15   }
16   else{
17     tmp = x;
18     if(x->left == NULL)
19       x = x->right;
20     else if(x->right == NULL)
21       x=x->left;
22     delete tmp;
23   }
24   return x;
25 }
```

1 Indata till funktionen är adressen till trädets rot samt elementet som skall tas bort.

3–4 Testar först om trädet är tomt. I så fall har man försökt ta bort ett element som inte finns i trädet, eller försökt ta bort ett element från ett tomt träd. I båda fallen avslutas programmet.

5–8 Med samma teknik som för sökandet, söker vi upp den plats där noden som skall tas bort finns.

9 När vi har hittat noden som skall tas bort, kollar vi först om den har två barn.

10–15 Om noden har två barn, söker vi upp den minsta noden i det högra delträdet.

13 Värdet på den minsta noden i det högre delträdet kopieras till den nod, som skall tas bort.

14 Sen tar vi bort rekursivt det minsta värdet i det högre delträdet.

16–23 Om noden som skall tas bort inte har två barn, måste den ha ett eller inget barn. I så fall får noden som skall tas bort värdet till sitt ena barn, eventuellt värdet `NULL`.

Programmet *BinSokTree.c* visar hur operationerna till binära sökträd kan användas. Utgå gärna från det för att lösa nedanstående problem.

UPPGIFT 5.5
Rita ett träd. Rita upp det binära sökträd, sådant det ser ut, efter att elementen med nedanstående nycklar lagts in i tur och ordning från vänster till höger.

7, 9, 10, 23, 45, 12, 78, 94, 100, 123, 53, 21, 1, 87, 52, 34, 89

Ge ett förslag på ordning hos talen för att det binära sökträdet ska få ett så liten höjd som möjligt. Rita även detta träd.

UPPGIFT 5.6
Hur många binära sökträd. Hur många olika binära sökträd kan man konstruera med hjälp av sju element med nycklarna 1, 2, 3, 4, 5, 6, 7 ? Vilka olika höjder kan dessa träd ha? OBS att detta är ett svårt problem!

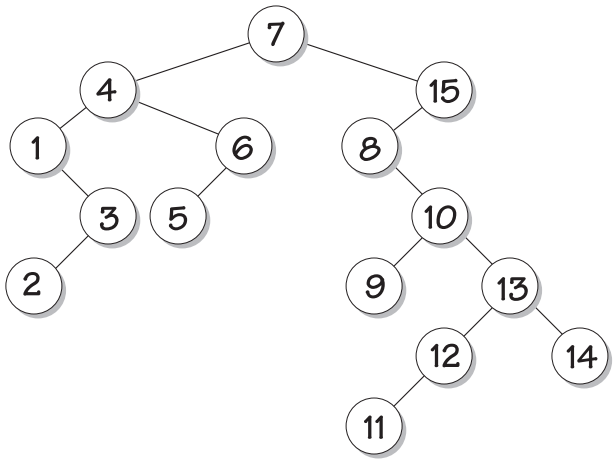
UPPGIFT 5.7
Två funktioner efterlyses. Skriv funktionerna `NumberOfLeaves` och `NumberOfNodes`, som bestämmer antalet löv respektive noder i ett binärt träd.

UPPGIFT 5.8
Bestäm höjden hos ett binärt träd. Skriv en funktion `Height`, som bestämmer höjden hos ett binärt sökträd. Använd sedan funktionen till att mäta höjden hos ett antal sådana träd som byggts upp utav med 1000 noder, där nyckeln är framslumpade heltal i intervallet [1...1000000]. Vilken är den teoretiskt lägsta höjden? Hur vanlig är den bland de 1000 träden?

UPPGIFT 5.9
Sökträd med ord. På filen *ord.txt* finns ett stort antal ord alla med sex bokstäver ord. Första raden i filen anger hur många ord den innehåller. Skriv ett program som bygger upp ett binärt sökträd av dessa ord.

- Skriv en rutin som frågar efter ett ord och tar reda på om ordet finns bland de givna.
- Jämför med blotta ögat de två insättningsrutinernas effektivitet (rekursiv eller iterativ).
- Mät trädets höjd och ta reda på hur många löv det har då du sätter in orden i trädet i den ordning de förekommer på filen.
- Vilken är den minsta höjd trädet kan ha, om man får kasta om orden på filen.
- Hur ska man skapa ett sådant träd? Det vill säga vilken ordning ska orden ha på filen?
- Vad vinner man på ett så lågt träd som möjligt?

UPPGIFT 5.10
Utskrift av binärt sökträd.



Figur 5.6: *Detta utseende får det binära sökträdet för exempelfilen* `uppg5.dat`. *Att skriva ut ett binärt sökträd nivå för nivå innebär alltid att rotens värde skrivs ut först.*

I programkoden `uppg5.cpp` finns två funktioner `Init` och `InsertTree`. Den första läser tal från filen `uppg5.dat`, som med hjälp av den senare funktionen sätts in i ett *binärt sökträd*. Din uppgift är nu att skriva funktionen `PrintTreeLevel`, som ska skriva ut talen i det binära sökträdet *nivå för nivå*. Plats finns angiven för var koden ska skrivas in. Parameter till funktionen är adressen till noden i trädets rot. Observera att det inte är tillåtet att ändra någon kod utanför funktionen `PrintTreeLevel`.

Indatafilen `uppg5.dat` inleds med ett tal `n` < 100, som anger hur många tal filen innehåller. Den medskickade testfilen, med följande utseende:

```
15
7 4 6 15 8 10 13 1 3 2 9 14 5 12 11
```

kan ge utskriften:

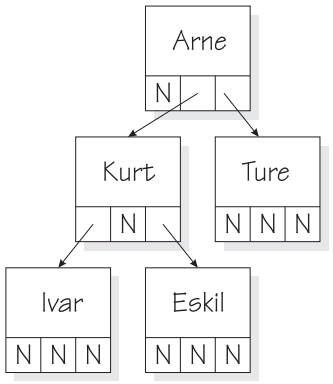
```
7 4 15 1 6 8 3 5 10 2 9 13 12 14 11
```

Jämför man med figuren så framgår klart vad som menas med ”nivå för nivå”. Talen på samma nivå i annan ordning i utskriften ger inga avdrag. Utskriften

```
7 15 4 6 8 1 3 5 10 13 9 2 12 14 11
```

betraktas alltså också som korrekt!

UPPGIFT 5.11
Namnträdet



Figur 5.7: *Ett exempel på hur datastrukturen kan se ut.* N *står för NULL*

Varje objekt i datastrukturen som visas i figuren bygger på

```
struct perstyp{
  char namn[10];
  struct perstyp *pek[3];
};
```

Förutom ett < 10 tecken långt namn finns en array med pekare, `pek`, i vilket kan lagras 3 adresser till objekt av typen `persontyp`. Alla platserna i arrayen behöver dock inte användas och innehåller då `NULL`, N i figuren.

Till ditt förfogande har du filerna `main.c`, `list.c`, *list.h* och även *list.o* (för att bygga ett projekt i DEV CPP).

Problemet består i att fullfölja funktionen *skrivut*, som beretts plats för i *main.c*. Denna funktion ska skriva ut samtliga namn som ingår i strukturen på skärmen (i godtycklig ordning). För exemplet i figuren kan utskriften bli

```
Arne Kurt Ture Ivar Eskil
```

Då du betraktar headerfilen ser du att ”biblioteket” innehåller två funktioner, `laesin` och `tabort`. Funktionen `laesin` läser in data från filen vars namn är parameter till funktionen, skapar en struktur liknande figuren ovan och returnerar adressen `start` till strukturens start. Funktionen `tabort` avlägsnar alla objekt som tidigare skapats av `laesin` på heapen och har adressen till datastrukturen som parameter. Det kan vara lämpligt att avsluta körningen med ett anrop av denna funktion.

Vilka namn som förväntas som utskrift vid körning med medskickade testfiler får du (kanske) reda på genom att titta sist i filen.

Det finns inga andra restriktioner för denna uppgift än att ingen datafil får öppnas från någon annan rutin än `laesin`.

5.2.8 Balanserade träd

Ett problem med de binära sökträd vi hanterat här är att de kan bli obalanserade. I den värsta situationen blir trädet till en länkad lista med följden att sökning i ett sådant träd blir sekvensiell. För att råda bot på detta problem har man skapat algoritmer som balanserar trädet i samband med insättningar och borttagningar. *AVL-träd* och *Red&Black-träd* är två skapelser av balanserade träd.

Här har vi genomfört ett empiriskt test av nyttan med *balanserade träd*. De två tekniker vi testat är *AVL-träd* och *Red&Black-träd*. I båda fallen har vi hittat (rekommenderad) kod på internet, skriven i C. I jämförelse med dessa metoder har vi också använt en naiv metod med sekvensiell sökning och *Binärt sökträd*.

Testen har utförts genom 20 000 000 slumpmässigt valda aktiviteter: *sökning, insättning* och *borttagning*. Först har ett slumptal `0...2` tagits fram för att avgöra vilken aktivitet som ska vidtas, därefter en nyckel, ett slumptal i intervallet `0..32767`. Därefter har aktiviteten genomförts med denna nyckel.

Samtliga försök gav förstår samma resultat

Aktivitet	**Fanns**	**Fanns ej**
Insättning	3328410	3339398
Sökning	3328487	3341238
Borttagning	3323095	3339372

Så fort nyckeln tagits fram sker en sökning för att ta reda på om nyckeln redan finns i trädet. Om aktiviteten är *insättning* sker detta endast om nyckeln inte finns i trädet. Om aktiviteten är *borttagning*, kan detta förstås endast ske om nyckeln finns i trädet.

De fyra metoderna gav följande exekveringstider

Metod	**Tid**
AVL-träd	12
Red&Black-träd	12
Binärt sökträd	11
Naiv Sekvensiell	653

Sensmoralen är att det för denna test inte spelar någon roll om vi använder *binärt sökträd* eller mer avancerade metoder. Däremot ser vi att den naiva sekvensiella metoden är förkastlig i jämförelse.

Koden till *AVL-träd* är hämtad från `www.stanford.edu/~blp/avl/` och koden till *Red&Black-träd* från
`web.mit.edu/~emin/www/source_code/red_black_tree/index.html`. Det är svårt att göra en bedömning av hur väl denna kod är skriven.
Testkoden finns i `BalanseradeTrad.zip`.

5.2.9 Kombinatoriska algoritmer

5.2.10 Generera samtliga delmängder

Vi vet att en mängd M med n element har 2^n delmängder, från den tomma mängden till hela mängden M. Nedan ser vi en funktion som genererar samtliga delmängder. Arrayen m antas ha lika många element som antalet element i mängden M. I m kommer endast 1:or och 0:or att placeras in. En 1:a i index k betyder att det k:e elementet ingår i aktuell delmängd.

```
1 void leta(int m[],int n){
2   if(n==antal)
3     testa(m);
4   else {
5     m[n]=1;
6     leta(m,n+1);
7     m[n]=0;
8     leta(m,n+1);
9   }
```

Ett exempel: Vi har följande 10 tal och undrar om summan 2000 kan bildas med några av dem:

$$231, 345, 562, 320, 827, 276, 933, 455, 254, 332$$

Genom att genererar alla 2^{10} delmängder och summera talen som ingår i varje mängd får vi svar på frågan. Det finns precis en lösning:

$$231 + 345 + 562 + 276 + 254 + 332 = 2000$$

5.2.11 Samtliga stryktipsrader

Här en algoritm som genererar samtliga stryktipsrader.

```
1 void generera(int rad[],int nr,int typ){
2   if(nr==13)
3     // Utför jobbet
4   else {
5     rad[nr]=0;
6     generera(rad,nr+1);
7     rad[nr]=1;
8     generera(rad,nr+1);
9     rad[nr]=2;
10    generera(rad,nr+1);
11  }
12 }
```

Här får vi de $3^{13} = 1594323$ raderna. Vad tror ni om följande sätt att skapa ett tipssystem? För varje match ger vi sannolikheten för varje utfall 1, X respektive 2. Ta till exempel "Tio Tidningars Tips". En möjlig fördelning för en match kan vara $0.60, 0.30, 0.10$. Sannolikheten för en viss rad kan nu bestämmas genom att multiplicera sannolikheten för varje tecken i raden. Bestäm sannolikheten för alla 1594323 raderna och rangordna dem efter denna sannolikhet.

Skapa nu ett system av rader genom att först välja den mest sannolika, den som ligger högst upp i rangordningen. Om denna rad skulle ge 13 rätt är man förstås inte intresserad av andra rader som ger 12, 11 eller 10 rätt. Det finns alltså en omgivning till den just valda raden som vi inte vill ha med. Välj därför som nästa rad, den första rad i rangordningen som skiljer sig från den tidigare med minst 4 tecken. Följande rader ska sedan skilja sig med minst 4 tecken från alla tidigare genererade. Generera på detta sätt sedan önskat antal rader.

Om man använder den här tekniken utan sannolikheter och tar raderna i den ordning de genereras kan man klämma in 981 rader. Tabellen nedan visar antalet rätt hos den bästa raden för samtliga 1594323 rätta rader.

Bästa rad	**Antal rader**
13	981
12	25506
11	306072
10	1051734
9	210030

Systemet garanterar 9 rätt och det är cirka 87% chans att få 10 rätt eller mer.

5.2.12 Delmängder med k element

Här ska vi generera all mängder med exakt k element. Här ett förslag till algoritm:

```
1  void generera(int m[],int n,int s,int k){
2    int i;
3    if(n==k)
4      kontroll(m);
5    else
6      for(i=s;i<antal;i++){
7        m[i]=1;
8        generera(m,n+1,i+1,k);
9        m[i]=0;
10     }
11 }
```

`m` är en array som från början innehåller `antal` 0:or. Algoritmen kommer att genererar alla

möjliga mängder med precis k 1:or insatta. Det finns som vi minns

$$\binom{\text{antal}}{k}$$

sådana mängder. n håller reda på hur många 1:or som finns på plats för tillfället. För till exempel `antal`$= 5$ och $k = 3$ kommer `m` att anta följande:

11100	10011
11010	01110
11001	01101
10110	01011
10101	00111

Som bekant är $\binom{5}{3} = 10$

Vi ska titta på ett alternativt sätt att generera dessa delmängder. Denna gång får man index till de element, som ska tas med, i en vektor som är k lång:

123	145
124	234
125	235
134	245
135	345

Algorithm 5.2.1: KSUBSETLEXSUCCESSOR(T, k, n)

$U \leftarrow T$
$i \leftarrow k$
while $i \geq 1$ **and** $t_i = n - k + i$
  do $i \leftarrow i - 1$
if $i = 0$
  then return ("undefined")
  else { **for** $j \leftarrow i$ **to** k
  do $u_j \leftarrow t_i + 1 + j - i$
  return (U)

- Vad utförs i `while`-loopen?
- Förvissa dig om att du förstår vad som händer i `for`-loopen

Detta är alltså ingen rekursiv funktion.

Figur 5.8:

UPPGIFT 5.12

Bästa laget Inför sommarens EM i fotboll har landslagstränare Lars Lagerbäck beställt ett program som, ska hjälpa honom att placera spelarna på rätt plats på planen och på det viset få starkaste möjliga lag.

Laget består av 11 spelare, numrerade 1...11, som ska placeras på lagets 11 platser, även de numrerade 1...11. Lagerbäck har för varje spelare listat de platser där spelaren kan användas och samtidigt givit en poäng, 1...10, för hur lyckade spelarna är på denna plats. Här ett exempel på en sådan lista (uppdelad i fyra kolumner)

```
23          3  5  2      6  1 10       9  3  7
1  7  9     3  7  8      6  5  3      10  1  1
1 10 10     3 11  2      6  8  8      10 10  2
2  2  3     4  3  8      7  6  8      11  1  7
2 11  1     4  4 10      7 11  2      11  9  6
3  4  7     5  2 10      8  8 10      11 11  1
```

Spelare 1 kan spela på platserna 7 och 10. Han är något bättre på plats 10 där Lagerbäck ger honom 10 poäng. Spelare 5 kan bara spela på plats 2. Inte heller för spelare 8 och 9 finns det något val.

Skriv ett program som från filen `laget.txt` läser in aktuella data och placerar de 11 spelarna på de 11 platserna – en spelare på varje – och samtidigt bestämmer den bästa placeringen genom givna poäng. Filen inleds med ett tal n som anger antalet spelarplaceringar. Efterföljande n rader innehåller tre tal: spelarnummer, platsnummer och poäng.

```
Högsta poängen är 75
```

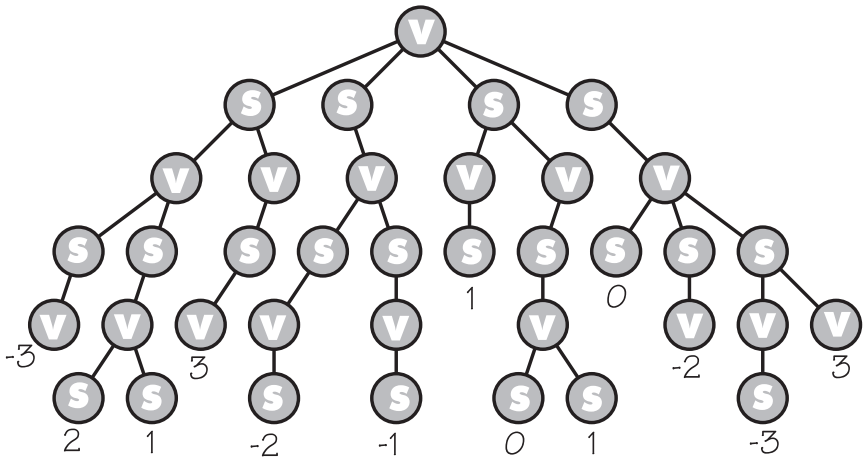
Denna poäng erhålles då man placerar

Spelare	1	2	3	4	5	6	7	8	9	10	11
Plats	7	11	5	4	2	1	6	8	3	10	9
Poäng	9	1	2	10	10	10	8	10	7	2	6

5.2.13 Spelteori – MiniMax

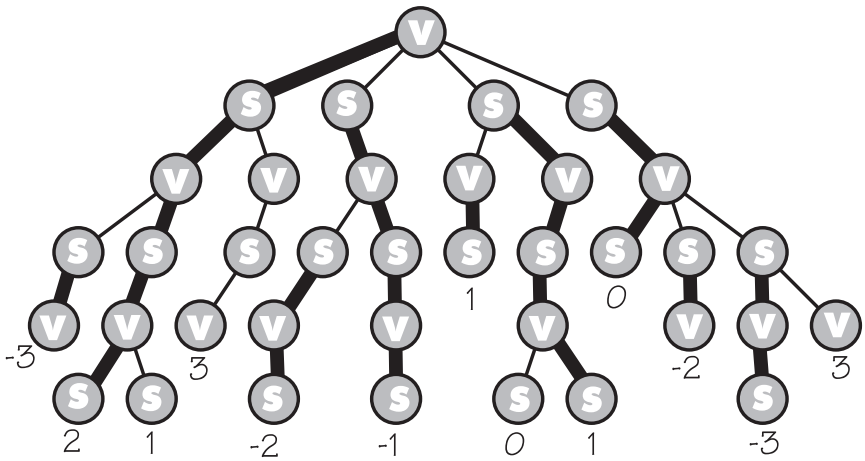
Nedan ser du ett spelträd. Noderna utgör ställningar i spelet. Bokstaven i noden anger vem som är vid draget, V vit eller S svart. Vit har alltså fyra drag att välja på från ställningen i roten.

Talen intill löven är värdet hos dessa ställningar. Ett negativt tal anger att S står bättre och för motsvarande positiva tal står V bättre. V försöker maximera och S försöker minimera. Vilket drag ska V välja för att nå en så hög poäng som möjligt?



Figur 5.9:

Det enklaste sättet att söka i ett spelträd och samtidigt det enklaste att förstå är just *Mini-Max*-algoritmen. Den söker igenom alla möjliga drag ned till ett fixt djup och bestämmer värdet hos alla slutställningar (löv). Dessa beräkningar används sedan till att bestämma poängen och draget upp till roten. Här är algoritmen:



Figur 5.10:

Algorithm 5.2.2: MINIMAX(ställning, djup)

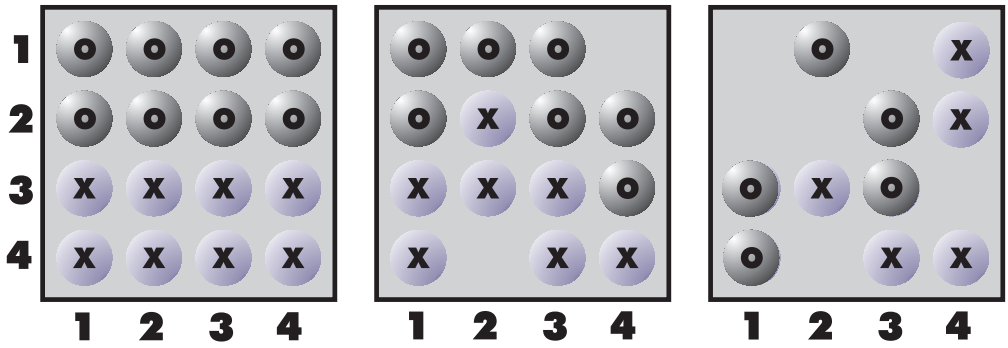
```
local struct drag lista[maxantaldrag]
local int i, antaldrag, bästavärde, värde
if testavinst(ställning)
  then { if ställning.färg=V
         then return (−∞)
         else return (∞)
if djup = 0
  then return (beräkna(ställning))
if ställning.färg=V
  then bästavärde ← −∞
  else bästavärde ← ∞
antaldrag ← genereradrag(ställning,lista)
if antaldrag = 0
  then return (hanteraingetdrag(ställning))
for i = 1 to antaldrag
  do { utfördrag(lista[i],ställning)
       värde ← minimax(ställning,djup-1)
       tatillbakadrag(lista[i],ställning)
       if ställning.färg = V)
         then bästavärde ← max(värde,bästavärde)
         else bästavärde ← min(värde,bästavärde)
return (bästavärde)
```

Idén här är att båda spelarna försöker alla möjliga drag i varje ställning och sedan väljer det drag som ger den för V högsta poängen (om vit är vid draget) och för S den lägsta (om svart är vid draget). V startar i algoritmen med $-\infty$ och försöker förbättra detta. På samma sätt startar S med ∞ som han försöker förbättra genom drag som leder till lägre poäng.

- `minmax`-funktionen tar emot en ställning och ett tal som anger hur djup analysen ska göras.
- `ställning` innehåller den aktuella ställningen som ska analyseras
- `struct drag` är objekt som innehåller information om aktuellt drag.
- `lista` är en array som kommer att innehålla samtliga tillåtna drag i form av `struct drag`-objekt
- `bästavärde` och `värde` är heltal som håller reda på en ställningvärderingen, som ett heltal från $-\infty < v < \infty$.
- Funktionen `testavinst` returnerar true om aktuell ställning är ett avslutat parti med en av spelarna som segrare.
- Om `djup`= 0 och ingen vinstställning har hittats på vägen hit utvärderas ställningen med hjälp av funktionen `beräkna`. Det returnerade värdet returneras direkt i sin tur.

- Funktionen `genereradrag` returnerar samtliga tillåtna drag i aktuell ställning i `lista`
- Samtliga drag i `lista` utförs nu i tur och ordning. `utfördrag` utför draget. Den uppkomna ställningen skickas rekursivt till `minmax`, men nu med ett djup närmare slutet och med motståndaren vid draget.
- När programmet återvänder är det viktigt att återställa draget med hjälp av `tatillbakadrag`.
- `värde` har nu ett värde som ska jämföras med tidigare beräknade värden.

Kono



Figur 5.11:

Kono är ett mycket gammalt spel med ursprung från Korea. I figur 5.11 ser du till vänster utgångsställningen. Brädet består av 4×4 brickor. Hälften av dessa, märkta med X tillhör den ena spelaren och brickorna med O den andra.

Spelarna som turas om att dra kan flytta en av sina egna brickor endast genom att hoppa över en egen intilliggande bricka och slå ut en motståndares, som ligger direkt efter den överhoppade brickan. Alla drag görs i samma kolumn eller rad (alltså inga diagonala drag). Den som först saknar möjlighet att göra ett drag har förlorat.

I ställningen i mitten av figuren har spelare X inlett med att flytta från $(4,2)$ till $(2,2)$ och slagit X's bricka på $(2,2)$. Spelare O har sedan svarat med $(1,4)$ till $(3,4)$ och slagit brickan på $(3,4)$. Spelare X som nu står i tur har följande fyra drag att välja på:

$(4,1)$ till $(2,1)$
$(3,2)$ till $(1,2)$
$(3,2)$ till $(3,4)$
$(4,3)$ till $(2,3)$

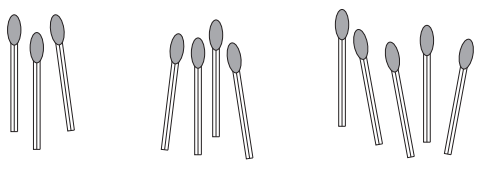
I ställningen till höger har spelare O, som står på tur endast draget $(2,3)$ till $(4,3)$. Därefter saknar spelare X drag och har därmed förlorat.

Här följer vad jag tror det optimala partiet som O vinner efter 8 drag.

```
 0000  .000  .000  ..00  ..00  ..00  ..00  ...0
 000X  000X  00XX  00XX  00XX  .00X  .X0X  .X0X
 XXXX  0XXX  0XXX  00XX  0XX.  0XX.  0XX.  0X0.
 XXX.  XXX.  XX..  XX..  XX..  XX..  X...  X...
 ----  ----  ----  ----  ----  ----  ----  ----
```

```
1  int main(void){
2    int i,n,moves[32][6],poang,max=-INFINITY,md;
3    struct position p;
4    init(&p);
5    do{
6      n=makemovelist(p,moves);
7      if(p.color==BLACK)
8        max=INFINITY;
9      else
10       max=-INFINITY;
11     if(n>0){
12       for(i=0;i<n;i++){
13         domove(moves[i],&p);
14         if(p.color==WHITE) p.color=BLACK; else p.color=WHITE;
15         poang=minimax(p,10);
16         if(p.color==WHITE) p.color=BLACK; else p.color=WHITE;
17         undomove(moves[i],&p);
18         if (p.color==WHITE && poang>=max ||
19             p.color==BLACK && poang<=max ){
20           max=poang;
21           md=i;
22         }
23       }
24       domove(moves[md],&p);
25       stallningut(p);
26       if (p.color==WHITE)
27         p.color=BLACK;
28       else
29         p.color=WHITE;
30     }
31   }while(n!=0);
```

Ett betydligt enklare spel att skriva ett program för är NIM.



Figur 5.12: *Utgångsställningen i den vanligaste varianten av NIM*

Spelet NIM i denna version har från start tre högar innehållande 3, 4 respektive 5 tändstickor. Ett drag består i att dra ett valfritt, från 1 upp till alla, stickor från valfri hög. De två spelarna turas om att dra och den som drar sista stickan vinner.

Programmen för både KONO och NIM finns för nedladdning. Vi återkommer till denna typ av 'spelprogrammering' senare i kursen, nu bara ett liten uppgift:

UPPGIFT 5.13
I den version av NIM-programmet som finns för nedladdning är det alltid programmet som gör första draget och kommer dessutom alltid att vinna, så länge högarna innehåller 3,4 och 5 stickor.

Justera programmet så att operatören får börja om denne så vill. Slumpa också antalet stickor i högarna (inom rimliga gränser) och skriv ut ställningen innan det avgörs vem som ska göra första draget.