Kapitel 3

Lista

3.1 Introduktion till lista

Listan är den mest använda av de abstrakta datastrukturerna. En lista kan jämföras med en pärm. Den innehåller ett antal blad (eller är möjligtvis tom). Man kan sätta in nya blad, ta bort gamla och och söka efter ett speciellt. Kanske vill man sortera bladen i pärmen.

En lista är en *ändlig följd av ordnade element* – linjärt ordnade. Om listan inte är tom har den ett första element och ett sista. Vi säger att den är linjärt ordnad därför att alla element (utom det sista) har precis en efterföljare.

Elementen i en lista är alla av samma typ. Denna typ kan dock väljas fritt, allt från ett enkelt tecken till en komplex post.

3.1.1 Operationer

Nedan anges de vanligaste operationerna som förknippas med den abstrakta datastrukturen *lista*. Dessa operationer bildar tillsammans ett *gränssnitt* eller *interface* mellan klient (tillämpningsprogram) och datastruktur. Operationerna ska vara klientens enda möjlighet att nå datastrukturen.

Insert. Med denna operation kan man placera in ett element var som helst i listan.

Algorithm 3.1.1: INSERT(value, pos)

Placerar ett element med värdet **value**
omedelbart före den angivna positionen **pos**

Remove. Man behöver förstås också en operation som kan ta bort element från listan.

Algorithm 3.1.2: REMOVE(pos)

Tar bort elementet i positionen **pos** från listan.

Inspect. Utan att förändra listan vill man ha en kopia av elementet på en given position

Algorithm 3.1.3: INSPECT(pos)

Returnerar värdet för elementet på angiven position **pos**

First. Returnerar en kopia av det element som finns först i listan.

Algorithm 3.1.4: FIRST()

Returnerar listans första element

End. Returnerar en kopia av listans sista element.

Algorithm 3.1.5: END()

Returnerar listans sista element

Next.

Algorithm 3.1.6: NEXT(pos)

Returnerar värdet för elementet närmast efter
angiven position pos

Observera att denna operation inte är definierad för listans sista element.

Previous.

Algorithm 3.1.7: PREVIOUS(pos)

Returnerar värdet för elementet närmast före angiven position pos

Denna operation är inte definierad för listans första element.

Isempty. Med denna operation kan man undvika att använda andra operationer felaktigt.

Algorithm 3.1.8: ISEMPTY()

Returnerar TRUE om listan är tom och FALSE annars

3.1.2 Implementera genom array.

Vårt första försök att implementera ADT *lista* sker genom en array, där vi direkt bortser från de rekommendationer angående operationer vi gjorde i förra avsnittet. Vissa av operationerna kommer att bli ganska krävande. Eftersom vi inte accepterar ”hål” i listan måste en del element flyttas när vi använder *remove*, *first* och *insert*

`assert.` Genom att bygga upp ett logiskt uttryck och testa om detta är sant, kan man förvissa sig om att efterföljande satser inte kommer att leda till att exekveringen misslyckas.

```
1  assert(c!=0);
2  a=b/c;
```

Om `c = 0` är villkoret i `assert` falskt och exekveringen avbryts med en kommentar liknande

```
1  Assertion failed: c!=0, file program.cpp line 7. Abort.
```

På detta sätt får man ett mjukare avslutning av programmet. Eftersom vissa av operationerna saknar möjlighet att avslutas på ett bra sätt, om de anropas med felaktiga parametrar, så är `assert` ett utmärkt hjälpmedel här.

`initlista`. Gemensamt för alla operationerna behövs tre globala variabler. `antal` ska hålla reda på antalet element som finns i listan. `storlek` håller reda på för hur många

element det finns plats. `lista` till sist är en pekare till den dynamiska array som utgör själva listan

Med denna funktion `initlist` inleds arbetet med en lista. Som parameter talar man om hur många element listan maximalt ska kunna hantera. Att `storlek` blir ett större än `s` beror på att vi här inte använder `lista[0]`. Om platsen i listan överensstämmer med platsen i arrayen bli koden enklare att följa. **`malloc`** använder vi för att dynamiskt allokera det utrymme vi behöver på heapen. Pekaren `lista` innehåller adressen till detta område.

```
1  #include <assert.h>
2  int antal,storlek;
3  int *lista;
4
5  int *initlist(int s){
6    int p;
7    antal=0;
8    storlek=s+1;
9    p=(int *)malloc(storlek*sizeof(int));
10   return p;
11 }
```

`insert`. Denna operationen används för att placera ett element på ett villkorligt ställe i listan. Som parametrar till funktionen anger vi värdet på elementet och positionen som det nya elementet skall läggas omedelbart före. Det nya elementet får efter insättningen denna positionen i listan.

```
1  void insert(int tal,int pos){
2    int i;
3    assert(pos>=1 && pos-1<=antal && antal<storlek);
4    for(i=antal;i>=pos;i--)
5      lista[i+1]=lista[i];
6    lista[pos]=tal;
7    antal++;
8  }
```

`remove` När man tar bort ett element ur listan uppstår ett hål, om nu inte detta element råkar vara det sista i listan. Detta hål måste ”täppas till”. Ju längre fram i listan hålet uppstår desto mer jobb.

```
1 void remove(int pos){
2   int i;
3   assert(pos<=antal && pos>0);
4   for(i=pos;i<antal;i++)
5     lista[i]=lista[i+1];
6   antal--;
7 }
```

freelist. När listan gjort sitt kan den med hjälp av denna operation tas bort den från heapen.

```
1 void freelist(void){
2   free(lista);
3 }
```

Denna implementation har samma svaghet som den av stack som vi gjorde i förra kapitlet. Är elementen av en annan typ än heltal måste vi ändra överallt.

UPPGIFT 3.1
Implementera lista som array. Implementera *lista* som en array och ta med samtliga operationer som nämns i 3.1.1, med samma namn. Skapa `lista.c` och `lista.h` och `main.c` med enkel test av operationerna. Om du väljer att använda en dynamisk array är det tillåts du att ta med operationerna **initlist** och **freelist**

3.1.3 Implementera genom länkad lista

Att implementera `lista` som så kallad *länkad lista* där varje element var för sig läggs på heapen och där elementen binds samman med hjälp av pekare är betydligt vanligare. Genom denna teknik vinner vi effektivitet för vissa operationer och förlorar för andra.

Vi inleder med ett exempel som visar på tekniken, innan vi går vidare med en mer systematisk genomgång av hur operationerna implementeras.

```
1 struct box{
2   int tal;
3   struct box *nasta;
4 };
5 typedef struct box box;
6 int main(void){
7   box *start,*ny;
```

Den inledande deklarationen av posten `box` innehåller ett tal och en pekare av typen `box`. Vi vet att `*nasta` är en pekare och att den kan peka på andra objekt av typ `box`.

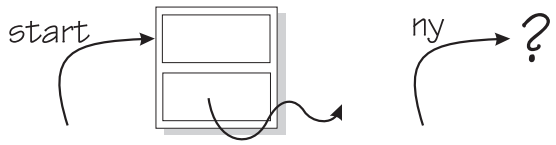
Strukturen `box` tillhandahåller i övrigt endast ett heltal, som får räcka i vårt lilla exempel.

Exemplet, som bara består av en huvudfunktion, har endast två variabler – två pekare, `start` och `ny` som kan bringas att peka på objekt av typ struct `box`. (figur 3.1)

Figur 3.1: *Två pekare som just nu inte pekar på någonting*

```
1   start=(box *)malloc(sizeof(box));
```

Genom `malloc` skapar vi ett utrymme på *heapen*. `malloc` lämnar ifrån sig en pekare som vi konverterar till en `box`-pekare. Pekaren `start` tar emot denna adress. (figur 3.2)



Figur 3.2: *Pekaren **start** bringas att peka på ett objekt*

```
1   start->tal=7;
2   start->nasta=NULL;
```

Genom *piloperatorn* `->` når vi de olika fälten i `box`-objektet. Med tanke på vad som ska följa är det kanske onödigt att NULL-ställa pekaren `nasta`. (figur 3.3)

Figur 3.3: *Genom **start** når vi utrymmet på heapen och kan tilldela fälten meningsfulla värden*

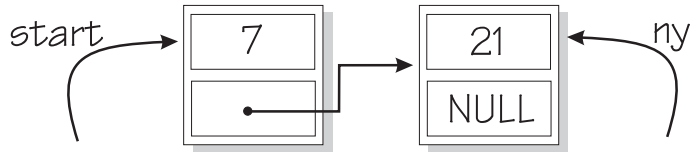
```
1   ny=(box *)malloc(sizeof(box));
2   ny->tal=21;
3   ny->nasta=NULL;
```

Ett nytt utrymme skapas, vars adress lagras i `ny`. På samma sätt, som tidigare, tilldelas fälten värden. (figur 3.4)

Figur 3.4: *Även pekaren **ny** ges en uppgift*

```
1   start->nasta=ny;
```

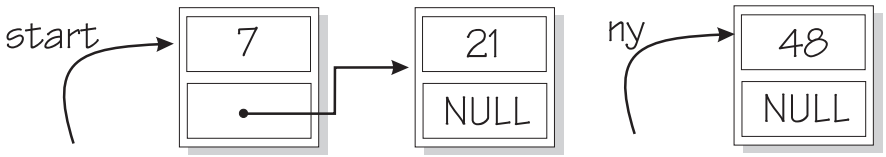
Så äntligen når vi fram till nyheten. Genom satsen ovan länkas de två objekten samman. Är du med på det? I och med detta har inte `ny` någon egentlig uppgift. Man kan nå båda objekten i listan genom adressen i `start`. (figur 3.5).



Figur 3.5: *Nu är de två objekten sammanlänkade och pekaren **ny**, har egentligen ingen uppgift längre.*

```
1   ny=(box *)malloc(sizeof(box));
2   ny->tal=48;
3   ny->nasta=NULL;
```

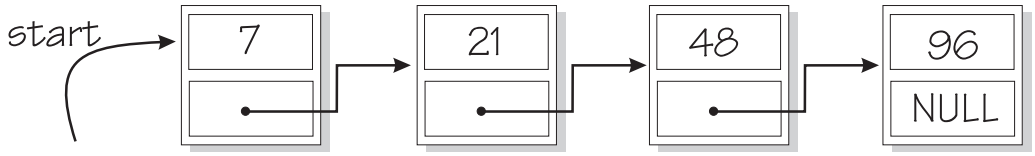
På samma sätt skapar vi så ett tredje objekt, som vi så småningom ska länka in i listan. (figur 3.6)



Figur 3.6: *Pekaren **ny** har fått en ny uppgift*

```
1   start->nasta->nasta=ny;
2   ny=(box *)malloc(sizeof(box));
3   ny->tal=96;
4   ny->nasta=NULL;
5   start->nasta->nasta->nasta=ny;
```

I rad 1 sker inlänkningen av talboxen med värdet. Ett nytt objekt skapas sedan och länkas in. När vi ser hur rad 5 växer i längd för varje objekt som ska länkas in i listan börjar vi kanske undra om detta kan göras på annat sätt, då vi till exempel ska ha 100 objekt i vår lista. (figur 3.7)



Figur 3.7: *Detta är en länkad lista bestående av fyra objekt.* ***start*** *vet var listan börjar och alla objekten vet var efterföljaren finns. Det sista objektet avslutar listan genom att nästa adress ges värdet NULL*

```
1   ny=start;
2   while(ny!=NULL){
3      printf("%d",ny->tal);
4      ny=ny->nasta;
5   }
6 }
```

Så till sist en utskrift av alla talen i listan. Det vi `direkt` når från programmet är alltså adressen till första objektet, genom `start`. Därifrån får vi sedan ta oss fram till listans slut via ”nasta”-pekarna. Pekaren `ny` får anta alla dessa adresser, med från början samma adress, som i `start`. Se till att du verkligen förstår konstruktionen ovan!

3.1.4 Implementation genom länkad lista

Här följer några av operationerna som de kan implementeras med hjälp av en länkad lista.

```
1 struct box{
2   char namn[10];
3   struct box *nasta;
4 };
5
6 typedef struct box boxtyp;
7
8 boxtyp *start;
9 int antal=0;
```

Posten *deklareras* av klienten. Förutom de fält som klienten behöver finns en pekare av samma typ som posten. Pekaren `start` sätts till `NULL` från början och pekar sedan hela tiden på första posten i listan.

Variabeln `antal` håller reda på antalet poster i listan.

```
1  boxtyp *adress(int plats){
2    boxtyp *box;
3    int n=1;
4    box=start;
5    while(n<plats){
6      box=box->nasta;
7      n++;
8    }
9    return box;
10 }
```

`adress` är en hjälprutin som söker upp adressen till elementet på platsen `plats`.

```
1  void insert(char* namn, int plats){
2
3    assert(plats >=1 && plats-1<= antal);
4    boxtyp *ny, *fore;
5    ny=(boxtyp *)malloc(sizeof(boxtyp));
6    strcpy(ny->namn, namn);
7
8    if(plats == 1){
9      ny->nasta = start;
10     start = ny;
11     antal++;
12   }
13   else{
14     fore = adress(plats-1);
15     ny->nasta = fore->nasta;
16     fore->nasta = ny;
17     antal++;
18   }
19 }
```

`insert` tar emot värdet av typen `char[]` och ett platsnummer. Insert ska sedan placera in denna post framför posten med givet platsnummer.

5–6 Utrymme för den nya platsen skapas och informationen kopieras över.

8–12 Om platsnummer är `1` betyder det att posten ska hamna först i listan. Vi behandlar detta som ett specialtillfälle därför att vi måste ändra på `start` pekaren.

14 I annat fall söker vi upp adressen till den post som ska ligga före den nya posten. Vi använder oss av hjälpfunktionen *adress*.

15-17 Två satser för att ordna pekarna och en för att notera det ökade antalet poster.

3.1.5 Länkad lista

Vi visar ytterligare ett par exempel på länkade listor, där vi mera väver in funktionerna i koden.

Dynamiska variabler och pekare

Vi startar med ett enkelt exempel. Exemplet vill visa hur man med hjälp av en `struct`, som innehåller en pekare av egen typ, kan bygga upp en kedja, en så kallad *länkad lista*, med objekt av typen `struct post`. Även om listan, endast innehåller två objekt, visar exemplet på den grundläggande tekniken.

```
1  #include <stdio.h>
2  #include <conio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  int main(void){
6    struct post{
7      char text[20];
8      struct post *next;
9    };
10   struct post *start,*tmp;
11   start=(struct post*)malloc(sizeof(struct post));
12   strcpy(start->text,"KALLE");
13   tmp=(struct post*)malloc(sizeof(struct post));
14   strcpy(tmp->text,"OLLE");
15   start->next=tmp;
16   tmp=NULL;
17   printf("%s\n",start->text);
18   printf("%s\n",start->next->text);
19   getch();
20 }
```

Kommentarer:

3 Vi måste inkludera `stdlib` för att kunna använda `malloc`

6-9 Posttypen `struct post` innehåller ett fält `text` som kan lagra upp till 20 tecken. Dessutom och det är det som är det viktiga i detta exempel, finns det ett fält som utgör en pekare av typen `struct post`. Denna pekare kommer att användas till att peka ut andra objekt av denna typ på heapen.

10 Vi deklarerar här två pekare, beredda att peka på objekt av typen `struct post`. `start` kommer att peka på det första objektet i den lista vi ska bygga upp. `tmp` kommer att få tillfälliga uppdrag.

11 `malloc` ”mutar in” ett område på heapen, vars storlek bestäms av storleken på, `sizeof`, ett objekt av typen `struct post`. `malloc` returnerar en pekare av `void`-typ, som typkonverteras till en pekare av typ `struct post`. `start` innehåller nu adressen till det just skapade objektet.

12 Fältet `text`, i det just skapade objektet tilldelas värdet `KALLE`. Observera att man inte kan nå detta fält på annat sätt än genom `start->text`. Ett alternativ till detta skrivsätt är dock `(*start).text`. Parentesparet behövs därför att operatorn *strukturmedlem* har högre prioritet än *indirektoperatorn*.

13-14 Vi upprepar samma sak en gång till, men nu med pekaren `tmp`. På heapen finns nu två objekt av typen `struct post`. Vi kan endast nå dem genom pekarna `start` och `tmp`.

15 Adressen som finns lagrad i `tmp` kopieras nu till `start->next`. Detta betyder att man kan nå det andra objektet från det första.

16 Vi tilldelar nu pekaren `tmp` värdet `NULL` och vill nu visa att vi kan nå det andra objektet, även utan `tmp`.

17 Vi skriver ut värdet hos fältet `text` för det första objektet.

18 Genom ett ytterligare steg når vi fram till fältet `text` i det andra objektet och kan därmed skriva ut värdet `OLLE`.

När man använder `malloc` för reservera minne på heapen, bör man också använda `free` för att återlämna minnet senast innan exekveringen avslutas.

Ett andra exempel

Självklart kan man inte bygga upp långa länkade listor med den metod vi använde ovan. Här följer ett exempel som kan skapa godtyckligt långa listor och som läser namnet från en textfil.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  int main(void){
5    struct post{
6      char text[20];
7      struct post *next;
8    };
9    FILE *infil;
10   char namn[20];
11   struct post *start,*tmp1,*tmp2;
12   int n,i;
```

Kommentarer:

H Vi använder samma `struct` som i tidigare exempel.

11 Vi behöver nu två temporära pekarvariabler.

```
1    infil=fopen("namn.dat","rt");
2    fscanf(infil,"%d",&n);
3    start=(struct post*)malloc(sizeof(struct post));
4    tmp1=start;
5    for(i=1;i<=n;i++){
6      fscanf(infil,"%s",namn);
7      strcpy(tmp1->text,namn);
8      if(i<n){
9        tmp2=(struct post*)malloc(sizeof(struct post));
10       tmp1->next=tmp2;
11       tmp1=tmp2;
12     }
13     else
14       tmp1->next=NULL;
15   }
16   fclose(infil);
```

Kommentarer:

1-2 Vi öppnar filen och läser in det tal som anger hur många namn filen innehåller.

3-4 Självklart ska arbetet göras genom en `for`-loop. Men först skapar vi det första objektet i listan. Pekaren `start` sätts att peka på detta startobjekt. Detsamma gör `tmp1`.

5-15 `for`-loopen kommer att snurra lika många varv som det finns namn på filen, `n`.

6-7 Nästa namn läses in från filen och kopieras till aktuellt objekt.

8-12 Om vi inte är inne på sista varvet, skapas här ett nytt objekt, som inledningsvis pekas ut av `tmp2`

10 Det är föregående objekt som ska peka ut detta nyskapade. Det ordnar vi genom att kopiera adressen i `tmp2` till `next` i föregående objekt.

11 I nästa varv blir detta nyskapade objekt, objektet som föregår nästa nyskapade. För att det ska stämma med adresserna måste `tmp2` kopieras till `tmp1`

14 Om det är sista varvet i loopen ska inget nytt objekt skapas. Istället blir det viktigt att placera in adressen `NULL` i listans sista `next`. Detta är det enda sätt på vilket vi kan avgöra att listan tar slut här.

Rita gärna en figur under tiden du sätter dig in i detta.

```
1   tmp1=start;
2   while(tmp1!=NULL){
3     printf("%s\n",tmp1->text);
4     tmp1=tmp1->next;
5   }
```

Kommentarer:

1 `tmp1` bringas att peka på listans första objekt.

2-5 `while`-loopen kommer nu att snurra tills vi träffar på adressen `NULL` i `next`. Visst hade vi kunna använda en `for`-sats här eftersom vi ju vet hur många objekt det finns i listan.

3 Vi skriver ut namnet i det objekt som `tmp1` pekar på.

4 `tmp1` kommer att peka på nästa objekt i listan. Alternativt blir `tmp1` `NULL` vilket leder till att loopen avbryts.

```
1   tmp1=start;
2   while(tmp1!=NULL){
3     tmp2=tmp1;
4     tmp1=tmp1->next;
5     free(tmp2);
6   }
7 }
```

Kommentarer:

H Nu är det dags att ta bort alla objekten från heapen, eller att återlämna minnet till systemet. Det skulle vara ett stort misstag att skriva `free(start)`, ty då återlämnas endast det första objektet i listan. Resten av listan finns kvar på heapen, men vi har ingen pekare, som kan leda oss dit!

3 Först kopierar vi adressen till aktuellt objekt till `tmp2`.

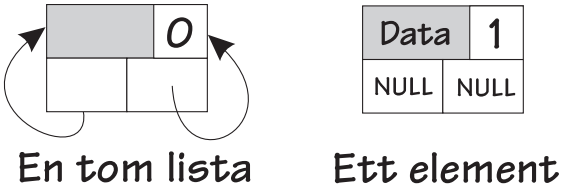
4 Nu använder vi `tmp1` till att peka ut nästa objekt i listan.

5 När vi nu frigör aktuellt objekt, har vi i `tmp1` adressen till resten av listan.

3.1.6 Dubbellänkde listor

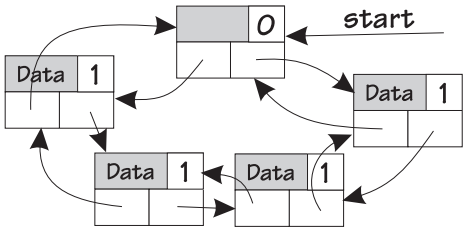
Funktioner för att hantera DubbelLänkad Lista

Nedan kommer vi att diskutera ett ”paket”, med rutiner som kommer att behövas för att administrera en *dubbellänkad lista*, men först en bild som förklarar vad det egentligen handlar om. En dubbellänkad lista är en kedja av objekt, där objekten knyts samman,



Figur 3.8: *En dubbellänkad lista består av ett huvud och ett antal element. Huvudet skapas först och för ett ögonblick finns då en tom lista. Ett element som ska placeras i listan måste först skapas och eventuellt tilldelas data.*

genom pekare, en till sin föregångare och en till sin efterföljare. Dessutom finns plats för objektets data. Varje länkad lista startar med ett huvud. Detta huvud har inga egna data, utan bara två pekare – till listans första respektive sista objekt. Ett fält i strukturen, *typ* talar om ifall posten i fråga är ett huvud (0) eller ett vanligt element (1). Det är inte, på något sätt, klart vilka rutiner som ska eller måste ingå i en sådant här ”paket”, inte heller hur den dubbellänkade listan ska se ut, utan här en personliga tolkning av begreppet.



Figur 3.9: *Här en bild över en dubbellänkad lista med fyra element. Pekaren* `start` *är programmets enda länk till listan. Skulle adressen i* `start` *förstöras kan inga av de data som finns i listan längre nås.*

Definition av strukturer

```
1  struct datatyp {
2    char namn[10];
3    int alder;
4  };
5
6  struct elementtyp{
7   int typ;
8   struct elementtyp *fore,*efter;
9   struct datatyp data;
10 };
```

För första gången ser vi en struktur som har en annan struktur i sig. `elementtyp` innehåller alltså fyra fält: två pekare, ett typfält och så datafältet. Detta fält kan enkelt bytas ut mot vad man för studen behöver eftersom ingen av följande funktioner använder sig av innehållet i ”datastrukturen”. `data`.

Att skapa en tom lista

Allt man behöver deklarera i main-funktionen är en eller ett par pekare av typ `struct elementtyp`. Dessa pekare används sedan som argument vid anrop av funktionerna nedan. Funktionen `newhead`, skapar en ny tom lista genom att skapa ett huvud där pekarna `fore` och `efter`, sätts att peka på huvudet självt. Dessutom sätts `typ` till 0, för att indikera att det rör sig om ett huvud.

```
1  void newhead(struct elementtyp **h){
2    *h=(struct elementtyp *) malloc(sizeof(struct elementtyp));
3    (*h)->typ=0;
4    (*h)->fore=*h;
5    (*h)->efter=*h;
6  }
```

Denna funktion anropas med `newhead(&start)`, där `&start` är den i main deklarerade pekaren. Varför skrivs parametern `**h`? Genom att studera figur 2 förstår du att det är en pekares värde, i main, som ska uppdateras.

Att skapa ett nytt element

På ett sätt, som påminner, om det i funktionen `newhead`, skapas här ett nytt element, som svävar fritt på heapen. Inplaceringen i listan kommer senare! (se figur 5)

```
1  void newlink(struct elementtyp **h){
2    *h=(struct elementtyp *) malloc(sizeof(struct elementtyp));
3    (*h)->typ=1;
4    (*h)->fore=NULL;
5    (*h)->efter=NULL;
6  }
```

Att ta ut ett element från listan

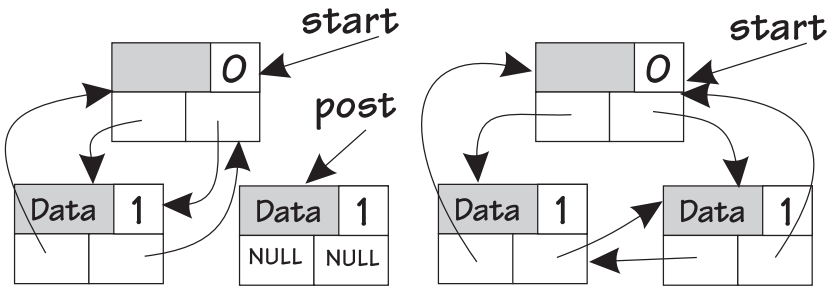
När ett element ska plockas bort från listan handlar det inte bara om att återlämna det minne som elementet tar upp. Man måste dessutom justera pekarna i elementen framför och efter det som ska tas bort. Det är det som denna funktion sköter om.

Observera alltså att denna funktion endast *tar ut* elementet från listan. För att ta bort ett element, helt och hållet, används `disposelink` (se nedan).

```
1 void out(struct elementtyp *e){
2   if (e->fore!=NULL) {
3     e->fore->efter=e->efter;
4     e->efter->fore=e->fore;
5     e->fore=NULL;
6     e->efter=NULL;
7   }
8 }
```

Att sätta in ett element före ett annat i listan

När ett nyskapat element ska placeras in i en lista måste man först med en egendefinierad funktion finna den plats där elementet ska in. Kanske vill man ha elementen sorterade. När



Figur 3.10: *För att placera in ett element i listan måste fyra pekare uppdateras – dels två i elementet själv och dels en i föregående respektive efterföljande element*

man väl funnit den plats där elementet ska in kan denna funktion användas för inplaceringen. Parametrarna är två pekare – den första till det nya elementet och det andra till den plats i listan framför vilken det nya elementet ska in.

```
1 void precede(struct elementtyp *e, struct elementtyp *x) {
2   out(e);
3   x->fore->efter=e;
4   e->fore=x->efter;
5   e->efter=x;
6   x->fore=e;
7 }
```

Att sätta in element efter ett annat i listan

Kanske kan man klara sig utan denna funktion när man har funktionen `precede`?

```
1 void follow(struct elementtyp *e, struct elementtyp *x) {
2   out(e);
3   x->efter->fore=e;
4   e->fore=x;
5   e->efter=x->efter;
6   x->efter=e;
7 }
```

Att sätta in ett element sist i listan

Så länge man vill sätta in ett nytt element sist eller först i listan räcker det med att söka upp listans huvud. Därifrån kan man sedan lätt justera de berörda pekarna.

```
1 void into(struct elementtyp *e, struct elementtyp *h) {
2   out(e);
3   h->fore->efter=e;
4   e->fore=h->fore;
5   e->efter=h;
6   h->fore=e;
7 }
```

Att sätta in ett element först i listan

Denna funktion är helt analog med `into`.

```
1 void intoasfirst(struct elementtyp *e, struct elementtyp *h) {
2   out(e);
3   h->efter->fore=e;
4   e->fore=h;
5   e->efter=h->efter;
6   h->efter=e;
7 }
```

Att erhålla en pekare till elementet före ett annat

Första funktionen i ”paketet”, som returnerar ett värde, här en pekare till elementet *före* det aktuella. Om elementet före är ett huvud returneras adressen `NULL`.

```
1 struct elementtyp *lpred(struct elementtyp *e) {
2   if (e->fore->typ==1)
3     return e->fore;
4   else
5     return NULL;
6 }
```

Att erhålla en pekare till elementet efter ett annat

Denna funktion är helt analog med `lpred`. Observera alltså att dessa två funktioner returnerar en adress, därför asterisken (*) före namnet.

```
1 struct elementtyp *lsucc(struct elementtyp *e) {
2   if (e->efter->typ==1)
3     return e->efter;
4   else
5     return NULL;
6 }
```

Att ta reda på om en lista är tom

Eftersom pekarna i en tom lista bringas att peka på huvudet själv är det enkelt att med ett enda villkor avgöra om listan är tom.

```
1 int empty(struct elementtyp *h) {
2   return h->efter->typ==0;
3 }
```

Att erhålla en pekare till första elementet

Lika enkelt är det att ta reda på adressen till första elementet i listan. Det enda man måste tänka på är: *att om lista är tom så finns det inget sådan element*.

```
1 struct elementtyp *first(struct elementtyp *h) {
2   if (!empty(h))
3     return h->efter;
4   else
5     return NULL;
6 }
```

Att erhålla en pekare till sista elementet

Denna funktion är helt analog med `*first`

```
1 struct elementtyp *last(struct elementtyp *h) {
2   if (!empty(h))
3     return h->fore;
4   else
5     return NULL;
6 }
```

Att ta reda på antalet element i listan

Funktionen `cardinal` räknar antalet element i listan. En liten fråga: *hur kan adressen* `e` *bli* `NULL` *, för det finns ju inga sådana adresser i listan.* Svaret är att funktionen `lsucc` returnerar `NULL` då rutinen når tillbaka till huvudet.

```
1  int cardinal(struct elementtyp *h) {
2    int k=0;
3    struct elementtyp *e;
4    e=first(h);
5    while (e!=NULL) {
6      k++;
7      e=lsucc(e);
8    }
9    return k;
10 }
```

Att ta bort ett element från listan

Tidigare har vi sagt att en del pekare måste uppdateras för att ett element ska kunna tas bort från listan. Det sköter funktionen `out` om. När det är klart återlämnas minnesutrymmet med hjälp av `free(*e)`.

```
1  void disposelink(struct elementtyp **e) {
2    out(*e);
3    free(*e);
4    *e=NULL;
5  }
```

Att ta bort alla element i listan

Att ta bort alla elementen i en lista gör man förstås enklast genom att anropa `disposelink` tills listan är tom.

```
1  void clear(struct elementtyp *h) {
2    struct elementtyp *e;
3    while (!empty(h)) {
4      e=first(h);
5      disposelink(&e);
6    }
7  }
```

Att ta bort hela listan, inklusive huvud

När även listans huvudet ska bort används denna funktion, som i sin tur anropar funktionen `clear`. Totalt hur många funktioner aktiveras egentligen genom ett anrop av denna funktion?

```
1 void disposehead(struct elementtyp **h) {
2   clear(*h);
3   free(*h);
4   *h=NULL;
5 }
```

Några exempel

För att detta med länkade listor ska klarna presenteras här några exempel, som använder funktionerna ovan.

Det första programmet gör inget annat än lägger upp tre poster på listan och när de är på plats skrivs de ut.

```
1  int main(void) {
2    struct elementtyp *start,*post;
3    newhead(&start);
4
5    newlink(&post);
6    strcpy(post->data.namn,"OLLE");
7    post->data.alder=49;
8    intoasfirst(post,start);
9
10   newlink(&post);
11   strcpy(post->data.namn,"KALLE");
12   post->data.alder=48;
13   intoasfirst(post,start);
14
15   newlink(&post);
16   strcpy(post->data.namn,"PELLE");
17   post->data.alder=52;
18   into(post,start);
19
20   skrivut(start);
21   disposehead(&start);
22 }
```

- Två pekare `*start` och `*post` deklareras, de är båda beredda att peka på variabler av typ `struct elementtyp`.
- Ett nytt huvud skapas genom `newhead(&start)`. `start` blir nu den pekare som kommer att hålla reda på adressen till hela listan. Skulle den tappas bort är därmed alla data som finns i listan förlorade.
- Pekaren `post` används nu att peka på nyskapade element, skapade med `newlink(&post)`. När elementet placerats in i listan är pekaren `post` klar att användas till annat.
- Fälten i `data` tilldelas värden.

- Med hjälp av funktionerna `intoasfirst` och `into` placeras sedan de nyskapade elementen i i listan. Ingen speciell sorteringsordning tillämpas.
- Funktionen `skrivut` skriver ut hela listan. Denna funktion måste vi skriva själva. Hur ska den se ut? (se nedan)
- Det ingår i god programmerarsed att städa efter sig. Därför plockar man här, till sist, bort hela listan genom `disposehead`

Funktionen skrivut

Genom parametern `*x` får vi adressen till listans huvud. Därifrån kan vi sedan ta oss genom hela listan och för varje element skriva ut det som önskas.

```
1 void skrivut(struct elementtyp *x) {
2   while (x->efter->typ!=0) {
3     printf("%d ",x->efter->data.alder);
4     x=x->efter;
5   }
6 }
```

`while`-loopen snurrar på så länge det finns nya element i listan. Då efterföljade element är av typ 0 har rutinen nått slutet. Gör klart för dig att du förstår satsen `x=x->efter`, det är den som är rutinens ”hjärta”.

Sortera in element i ordning

Vid varje användning av dubbellänkade listor blir man tvungen att skriva en del egna funktioner, liknande `skrivut` ovan. Här en funktion, som sorterar in elementen i en lista efter fallande ordning.

```
1 void inordning(struct elementtyp *e, struct elementtyp *h) {
2   struct elementtyp *x;
3   x=h;
4   while (x->efter->typ!=0 &&
5          x->efter->data.alder>e->data.alder) {
6     x=x->efter;
7   }
8   follow(e,x);
9 }
```

- Parametrarna till funktionen håller reda på adresserna till elementet som ska sättas in (`*e`) och själva listan (`*h`)
- För att finna den plats där elementet ska in, söker rutinen vidare, tills talet i listan är mindre än (eller lika med) talet i det nya elementet. Men man måste dessutom kontrollera att man inte nått fram till huvudet. Detta inträffar då listan från början är tom, eller då det nya elementets tal är mindre än alla tidigare inlagda.
- När `while`-loopen är bruten sker inplaceringen med hjälp av funktionen `follow`.

Att plocka bort element

I den här funktionen vill man plocka bort element vars ålder ligger i ett givet intervall. Listan förutsätts inte sorterad, så därför måste alla elementen granskas.

```
1  void tabort(int min, int max,struct elementtyp *h) {
2    struct elementtyp *x,*y;
3    x=h;
4    while (x->efter->typ!=0) {
5      if (x->efter->data.alder>=min && x->efter->data.alder<=max) {
6        y=x->efter;
7        disposelink(&y);
8      }
9      else
10       x=x->efter;
11   }
12 }
```

- Variablerna `max` och `min` innehåller intervallets gränser.
- `while`-loopen avslutas då listan sökts igenom.
- I `if`-satsen avgörs om talet ligger i intervallet $[\texttt{min}, \texttt{max}]$. om så är fallet plockas elementet bort ur listan med hjälp av funktionen `disposelink`.
- Genom satsen `x=x->efter` stegar man sig framåt i listan. Varför ska denna sats inte exekveras efter att ett element tagits bort?

3.2 Programmeringsuppgifter

UPPGIFT 3.2

Flytta längst fram. Programmet LNKLST1.CPP läser in data i form av förnamn, ($\leq$ 10 tecken långa), från en textfil, NAMN.DAT och lägger upp namnen i en länkad lista, i samma ordning som de förekommer på filen. Programmet avslutas med en utskrift av den länkade listans utseende.

Du ska nu skriva en funktion, OVERST. Som söker efter ett givet namn i listan och om det påträffas flyttar det till första platsen i listan. Funktionen har två parametrar: *adressen till första elementet i listan* och en sträng med *det eftersökta namnet*.

Testa programmet genom att placera in ett antal funktionsarop av OVERST mellan LAS_IN och SKRIV_UT, som till exempel

```
1     overst(&start,"PELLE");
2     overst(&start,"KURT");
```

Med ursprungslistan KALLE PELLE OLLE KURT SVEN, kommer resultatet att bli KURT PELLE KALLE OLLE SVEN.

Om namnet i andra parametern inte finns i listan ska listan förstås lämnas oförändrad.

Självklart ska programmet fungera för olika indatafiler NAMN.DAT Dock alltid sådana att de två funktionerna LAS_IN och SKRIV_UT fortfarande fungerar som det är tänkt.

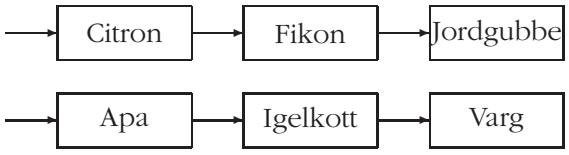
UPPGIFT 3.3

Samsortering. I `lnklst2.cpp` finns delar av ett program till vilket du ska skriva funktionen *merge*.

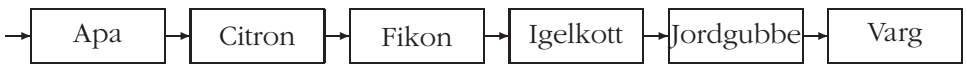
Programmet läser inledningsvis in ord från två textfiler och placerar dem i två länkade listor (med hjälp av funktionen `laes_in`). Eftersom orden i filen är sorterade i bokstavsordning kommer också de länkade listorna att vara sorterade.

Din uppgift blir nu att *samsortera* de två listorna till en länkad, sorterad, lista. Koden för denna samsortering ska finnas i funktionen *merge*.

Till exempel med följande två ursprungslistor



ska resultatet bli:



Funktionen *skriv_ut* skriver till sist ut den samsorterade listan.

Filerna innehåller ett ord på varje rad. Inget ord innehåller mer än 9 tecken och heller inte bokstäverna *å, ä* och *ö*. Till ditt förfogande finns fem olika indatafiler `uppg3a.dat` till `uppg3e.dat`. Ingen annan kod än filnamnen får ändras utanför funktionen *merge*.

UPPGIFT 3.4

Lista med damer. På filen *damnamn.dat* finns ett antal flicknamn lagrade – ett på varje rad. Med hjälp av programmet *damnamn.c* kan du läsa in dessa namn och lagra dem i en länkad lista – det sköter funktionen *init* om.

Med hjälp av funktionen *skrivlista* kan du sedan få listans innehåll, i form av flicknamn, utskriven på skärmen.

I *main* ser du ett antal funktionsanrop där motsvarande funktioner inte finns tillgängliga. Det blir nu din uppgift att skriva dem så att programmet kan exekveras.

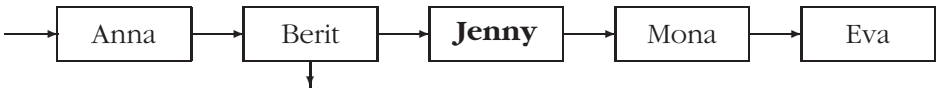
Nedan följer en närmare beskrivning av vad funktionerna uträttar. De beskrivs i stigande svårighetsgrad och det är därför lämpligt att du utvecklar dem i den ordningen.

antal(liststart) ska returnera antalet element i en lista. *liststart* innehåller adressen till listans början.

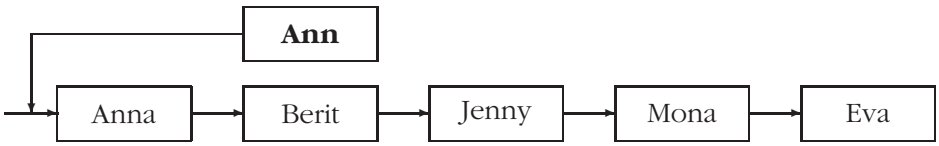
finns(liststart,"Ulla") ska returnera 1 om namnet "Ulla" finns i listan, annars returneras 0.

efter(liststart,"Kristina") ska returnera adressen till det element som följer efter elementet med namnet *Kristina*. Om namnet *Kristina* inte kan återfinnas eller om detta element finns sist i listan ska adressen NULL returneras.

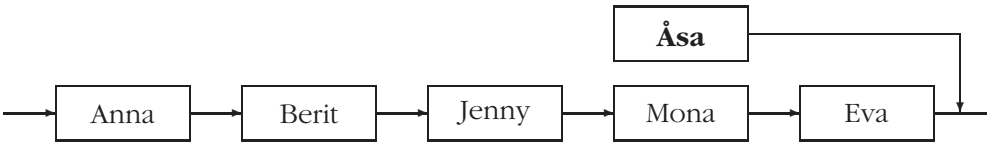
fore(liststart,"Jenny") ska returnera adressen till det element som föregår elementet med namnet *Jenny*. Om namnet *Jenny* inte finns med i listan eller om elementet inleder listan, så ska adressen NULL returneras.



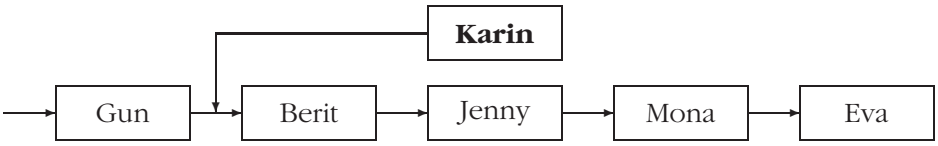
placeraforst(&liststart,"Ann") ska placera in ett element först i listan med namnet "Ann". Självklart ska adressen till listans start uppdateras. Funktionen returnerar ingenting.



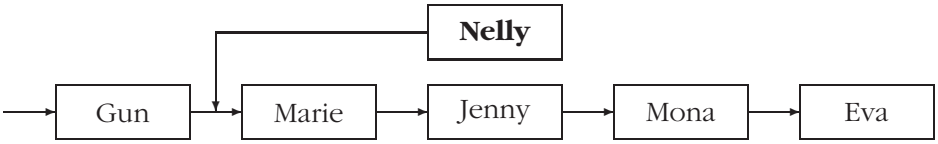
placerasist(&liststart,"Åsa") ska skapa ett element sist i listan med namnet *Åsa*. I vilken speciell situation ska kan adressen till listan ändras i denna rutin? I övrigt returnerar funktionen ingenting.



placeraefter(liststart,"Karin","Gun") ska placera in ett nytt element med namnet *Karin* efter elementet med namnet *Gun*. Om *Gun* inte kan återfinnas sker förstås ingen insättning och funktionen returnerar 0. Om allt "gått väl" returneras däremot 1.



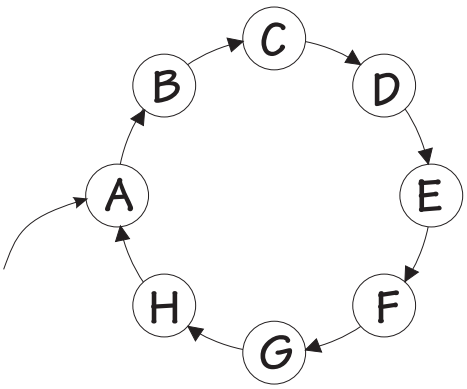
placerafore(&liststart,"Nelly","Marie") ska placera in ett nytt element med namnet *Nelly* före *Marie*. Om *Marie* inte återfinns i listan sker ingen insättning samtidigt som 0 returneras. I lyckade fall returneras 1.



tabortplats(&liststart,4) ska ta bort det fjärde elementet i listan. Självklart ska elementet inte bara tas bort från listan utan också från heapen. Lyckat genomförande returnerar 1 annars 0.

tabortnamn(&liststart,"Daniella") ska ta bort elementet med namnet *Daniella* från både lista och heap. Om elementet har kunnat avlägsnas returneras 1 annars 0.

Observera att funktionerna ska fungera även om listan från början är tom, det vill säga *liststart=NULL*. Använd funktionen *skrivlista* före och efter dina anrop, så att du kan se att allt fungerar som det ska. Tänk på att testa flera av funktionerna på ett sådan sätt att *första* och *sista* elementet är inblandade.



Figur 3.11:

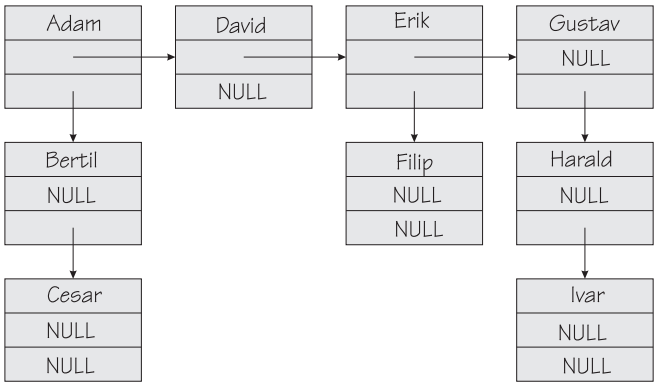
UPPGIFT 3.5
Ringlek

Figur 3.11 består av 8 barn i en ring, beredda att *räkna* om vem som ska *stå*, nu när de ska leka *kurra gömma*. Ramsan de använder innehåller 11 ord. Eftersom räknandet börjar med A och sedan går vidare *medurs* i pilarnas riktning, har de nått fram till C vid det 11:e ordet. Därför får C lämna ringen och räknandet startar om med början från D. Nästa barn som får lämna ringen är G. Proceduren fortsätter sedan tills endast ett barn återstår – det som ska stå – D i vårt exempel.

Till denna uppgift finns ett program `Uppg3.cpp`. Programmet består av tre funktioner.

- `skapa`, som bygger upp ringen i form av en dynamisk struktur. Egentligen inget annat än en länkad lista, där sista elementet pekar tillbaka på det första. De $2 \leq n \leq 20$ barnen döps till A, B, C... i tur och ordning. Funktionen returnerar alltid adressen till barnet A. **I denna funktion får inget ändras**.
- `rakna`, räknar och returnerar hur många barn det finns i ringen. Indata är, adressen till det barn där nästa räkning kommer att starta. **I denna funktion får inget ändras**.
- `raknaut` är funktionen du ska skriva. Funktionen ska ta emot adressen till det barn i ringen där räknandet ska starta, samt antalet ord $2 \leq m \leq 30$ i ramsan. Funktionen ska returnera adressen till det barn där räknandet ska börja nästa gång, men dessförinnan ska det uträknade barnet länkas bort från ringen (och helst tas bort från heapen).
- `main` anropar först `skapa` och därefter, genom en do-loop `raknaut` tills endast ett barn återstår. Adressen till detta barn används sedan för att skriva ut dess ”namn”. **I denna funktion får heller inget ändras**. Utom define-raderna för att ge andra värden på *antal barn* och *antal ord*

UPPGIFT 3.6
Länkade namn



Figur 3.12:

I figur 3.12 ser du en datastruktur, som vi kan kalla *lista av länkade listor*. Problemet består i att skriva en funktion `skrivut`, som skriver ut namnet i det sista objektet hos varje *vertikal* lista. Från exemplet i figuren skulle det innebära:

```
Cesar  David  Filip  Ivar
```

Till ditt förfogande har du filerna `main.c`, `list.h` och `list.o` (antagligen DEV CPP specifik). Då du betraktar *headerfilen* ser du att ”biblioteket” innehåller två funktioner, `laesin` och `tabort`. Funktionen `laesin` läser in data från filen `lista1.dat`, skapar en struktur liknande figuren ovan och returnerar adressen till strukturens start. Funktionen `tabort` avlägsnar alla objekt som tidigare skapats av `laesin` på heapen och har adressen till datastrukturen som parameter. Båda dessa funktioner ska anropas en gång precis som de görs i `main.c`.

Dessutom finns fyra ytterligare testfiler `lista2.dat`, `lista3.dat`, `lista4.dat` och `lista5.dat` där det förväntade svaret finns inskrivet i slutet. Observera dock att filerna måste döpas om till *lista.dat* för att *laesin* ska finna den.

Det kan betraktas som vanligt att man enbart har objektfilen, som här `list.o`, istället för `list.c`. I projektet ingår då bara källfilerna. `list.o` används inte förrän vid länkningen. I DEV CPP ställer man in detta under *Projekt – Projektalternativ*

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