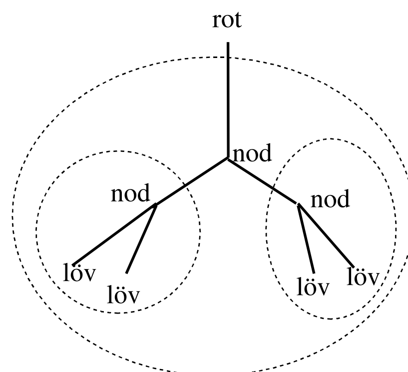
5 Sökträd och söktabeller

I en vektor kan man snabba upp en sökning efter en nyckel, genom att först sortera vektorn och sedan använda binärsökning. Detta innebär att man alltid halverar datamängden, som man ska söka i. Att göra samma sak i dynamiska datastrukturer, som listor, är inte lika lätt, eftersom elementen ej är indexerade. Man kan naturligtvis skriva över alla element i listan till en vektor och sedan binärsöka, men då förlorar man dynamiken i hanteringen. Istället använder man sig av *speciella dynamiska datastrukturer som sökträd och söktabeller,* för att snabba upp sökningar. Dessa datastrukturer är *hierarkiskt uppbyggda* och därmed minskar man ner den datamängd, som man varje gång måste söka i. I ett sökträd vet man alltid vilken gren man ska välja nästa gång och i en söktabell vet man vilket index man ska börja sökningen på.

5.1 Sökträd

Ett träd är uppbyggt av noder. Varje nod *innehåller data och pekare* till efterföljande noder. Datastrukturen träd kan jämföras med ett upp och nedvänt verkligt träd med rot, grenar och löv. Noderna utgör greningspunkterna och innehåller data. De yttersta noderna kallas för löv. I ett binärt träd har en *nod maximalt två efterföljande noder* enligt:



När man ska söka efter data i ett träd ser man till att *data är ordnat* på något sätt så att man alltid kan välja rätt väg från roten och ner till den nod, som eftersökt data finns i.

Ett binärt träd är en *rekursiv datastruktur* eftersom varje träd antingen är tomt eller består av en nod, vänsterträd och högerträd där vänsterträd i sin tur är tomt eller består av nod, vänsterträd och högerträd osv. Samma struktur *upprepas* hela tiden och då är det enklast att använda rekursion i sina algoritmer. *Rekursion är ett verktyg inom programmering som kan användas istället för iteration.*

5.1.1 Rekursion

Ett kraftfullt verktyg, vid utveckling av algoritmer och program, *är möjligheten för en funktion att anropa sig själv,* för att lösa ett problem. Denna teknik som kallas för rekursion kan och måste ibland användas vid problemlösning istället för iteration. *Vid rekursion anropar en funktion sig själv med nya aktuella parametervärden.*

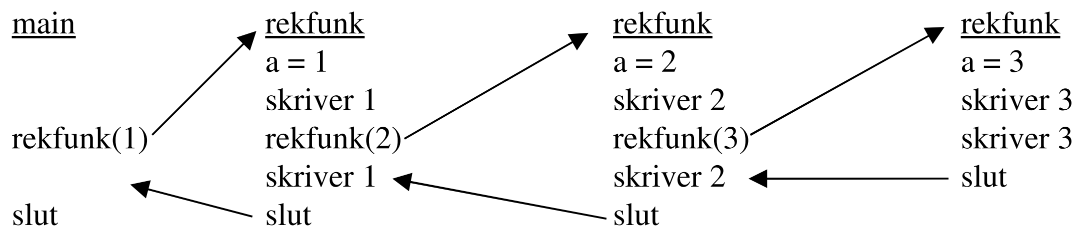
Ex: Vad gör nedanstående program?

```
#include <stdio.h>

void rekfunk(int a)
{
   printf("%d\n", a);
   if ( a < 3 )
      rekfunk(a+1);      /* OBS! Rekursivt anrop */
   printf("%d\n", a);
}

int main()
{
   rekfunk(1);
   return 0;
}
```

En körning visar följande på skärmen:

1
2
3
3
2
1

Varje nytt anrop av rekfunk *skapar en ny upplaga* av rekfunk med nya lokala variabler i minnet på stacken enligt:



Rekursion användes även i matematiken. Man kan definiera en matematisk funktion rekursivt genom att referera till samma funktion.

Ex: Ge en rekursiv definition av n-fakultet, n! = 1*2*3*4*….*(n-1)*n.

$$n! = \begin{cases} 1 & \text{om } n = 0 \\ n * (n-1)! & \text{om } n > 0 \end{cases}$$

Enligt definitionen beräknas exempelvis 3! enligt:

3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6

När man skriver rekursiva funktioner i sina program, ska *man bygga koden på den rekursiva definitionen* och ej tänka på hur funktionen i detalj löser problemet.

Ex: Skriv en rekursiv funktion för fakultetsberäkningen ovan.

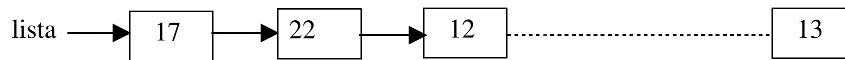
```
int rekfak(int tal)
{
   if ( tal > 0 )
      return tal * rekfak(tal-1);
   else
      return 1;
}
```

Rekursiva funktioner kräver mycket datorkraft, eftersom nya upplagor av funktioner med parametrar och lokala variabler skapas vid varje nytt anrop. Man ska enbart använda rekursion då koden ej går att skriva, *eller blir betydligt enklare*, med iteration. Fakultetsberäkningar skrivs lika enkelt med iteration som med rekursion och då bör man använda iteration istället.

Ex: Skriv en iterativ funktion för fakultetsberäkning.

```
int iterfak(int tal)
{
   int f = 1, i;

   for ( i = 2; i <= tal; i++)
      f = f * i;
   return f;
}
```

Ex: Skriv en rekursiv sökfunktion för en länkad lista av heltal enligt :



När man skriver en rekursiv algoritm för en länkad *lista ska man utgå ifrån den rekursiva definitionen av listan*. En länkad lista kan rekursivt definieras som:

En länkad lista är antingen
tom
eller så består den av
ett första element och en resterande länkad lista.

Att detta är en rekursiv definition ser man på att den innehåller *en referens till sig själv*, nämligen resterande länkad lista. Den rekursiva funktionen ska ha precis samma modell som den rekursiva definitionen. Alla komponenter i definitionen ska ingå i funktionen.

Om man antar att typdefinitionen :

```
typedef
struct link
{
   int data;
   struct link *next;
} linktyp;
```

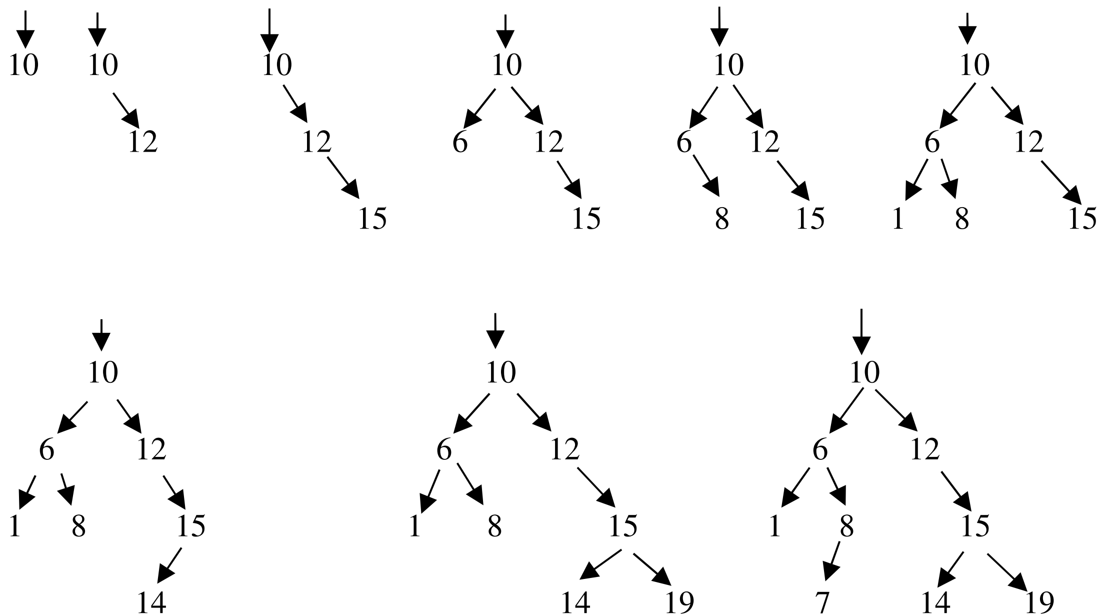
finns och låter sökfunktionen returnera en pekare till den sökta länken blir den enligt:

```
linktyp *searchlist(linktyp *lp, int key)
{
   if ( lp == NULL )
      /* tom lista */
      return NULL;
   else if ( key == lp->data)
      /* träff med första element */
      return lp;
   else
      /* sök i resterande lista */
      return searchlist(lp->next, key);
}
```

5.1.2 Binära sökträd

När man skapar ett binärt sökträd ser man vid skapandet till att *skapa ordning i trädet* genom att exempelvis alltid sätta in mindre element i vänster nod och större i höger nod.

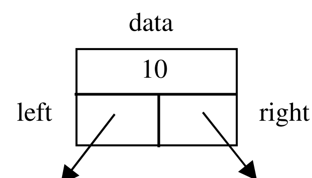
Ex: Skriv ett program som stoppar in heltalen 10, 12, 15, 6, 8, 1, 14, 19 och 7 i ett binärt sökträd i ordning enligt:



Programmet ska sedan skriva ut talen på skärmen i ordning 1, 6, 7, 8, 10, 12, 14, 15 och 19 och sedan fortsätta med att fråga efter ett nyckeltal och leta upp om nyckeln finns i trädet eller ej. Avslutningsvis ska trädet elimineras genom att allokerat minne frigörs.

När det gäller definition av datatypen för ett binärt träd så använder man sig av noder (länkar), som innehåller data och två pekare till de närmaste noderna.

```
struct nod
{
   int data;
   struct nod *left, *right;
};
```



När man ska skriva funktionerna för att stoppa in data i trädet, visa data i trädet på skärmen, söka efter data och eliminera trädet, ska man *bygga på det binära trädets rekursiva definition* enligt:

Ett binärt träd är antingen
  tomt
eller så består det av
  nod, vänsterträd och högerträd

Alla funktioner ska arbeta på tomt träd, nod, vänsterträd och högerträd.

```
/* taltree.c */

#include <stdio.h>
#include <stdlib.h>


struct nod
{
   int data;
   struct nod *left, *right;
};



void intotree(struct nod **rpp, int tal)
/* Stoppar in data i ordning i ett binärt sökträd */
{
   if ( *rpp == NULL )                        /* Tomt träd */
   {
      *rpp = malloc(sizeof(struct nod));      /* Skapa nod */
      (*rpp)->data = tal;
      (*rpp)->left = NULL;
      (*rpp)->right = NULL;
   }
   else if ( tal < (*rpp)->data )             /* Vänsterträd */
      intotree(&(*rpp)->left, tal);
   else                                       /* Högerträd */
      intotree(&(*rpp)->right, tal);
}


void showtree(struct nod *rp)
/* Visar data i ordning i ett binärt sökträd */
{
   if ( rp != NULL )                          /* Om ej tomt */
   {
      showtree(rp->left);                     /* Vänsterträd */
      printf("%d\n", rp->data);               /* Nod */
      showtree(rp->right);                    /* Högerträd */
   }
}
```

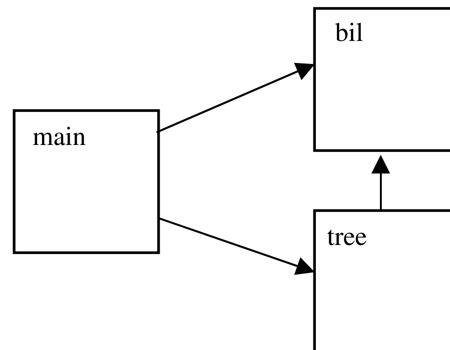
```
struct nod *searchtree(struct nod *rp, int key)
/* Söker efter key i ett binärt sökträd */
{
   if ( rp == NULL )                          /* Tomt träd */
      return NULL;
   else if ( key == rp->data )                /* Nod */
      return rp;
   else if ( key < rp->data )                 /* Vänsterträd */
      return searchtree(rp->left, key);
   else                                       /* Högerträd */
      return searchtree(rp->right, key);
}



void elimtree(struct nod **rpp)
/* Avallokerar ett binärt sökträd */
{
   if ( *rpp != NULL )                        /* Om ej tomt */
   {
      elimtree(&(*rpp)->left);                /* Vänsterträd */
      elimtree(&(*rpp)->right);               /* Högerträd */
      free(*rpp);                             /* Nod */
      *rpp = NULL;
   }
}



int main()
{
   int nyckel, i, vek[9] = {10, 12, 15, 6, 8, 1, 14, 19, 7};
   struct nod *rotp = NULL;       /* OBS! */

   for (i = 0; i < 9; i++)
      intotree(&rotp, vek[i]);

   showtree(rotp);

   printf("Ge nyckel : ");
   scanf("%d", &nyckel);
   if (searchtree(rotp, nyckel)!= NULL)
      printf("Nyckel finns!\n");
   else
      printf("Nyckel finns ej !\n");

   elimtree(&rotp);

   return 0;
}
```

OBS! Rotpekaren *måste initieras till NULL* för att funktionerna ska fungera.

Ovanstående data och funktioner fungerar bara för binära sökträd som innehåller data i form av heltal. Vill man göra en mer generell enhet för hantering av binära sökträd gör man på motsvarande sätt som ovan med twolist en *abstrakt datatyp som exporterar data och funktioner för binära sökträd.*

Ex: Skriv ett program som från en binärfil bestående av bilar med registreringsnummer och ägare läser in alla bilar till ett binärt sökträd ordnat efter registreringsnummer, skriver ut alla bilar i sökträdet, frågar efter ett registreringsnummer och skriver ut den eftersökta bilen, om den finns.

Ett beroendediagram för programmet blir:




```
/* Specifikation av bil - bil.h */

#ifndef BIL_H
#define BIL_H

typedef
struct bilpost
{
   char regnr[10];
   char agare[30];
} biltyp;

void read_bil(biltyp *bil);
/* Läs in data från tangentbordet till bil */

void write_bil(biltyp bil);
/* Skriv ut data om bil på skärmen */

int less_agare(biltyp b1, biltyp b2);
/* Returnerar 1 om b1.agare < b2.agare */

int less_regnr(biltyp b1, biltyp b2);
/* Returnerar 1 om b1.regnr < b2.regnr */

int lika_bil(biltyp b1, biltyp b2);
/* Returnerar 1 om b1.regnr == b2.regnr */

#endif
```

```
/* Implementation av bil - bil.c */

#include <stdio.h>
#include <string.h>
#include "bil.h"


void read_bil(biltyp *bil)
{
   printf("\nGe bilens regnr : ");
   gets(bil->regnr);
   printf("Ge bilens ägare : ");
   gets(bil->agare);
}


void write_bil(biltyp bil)
{
   putchar('\n');
   puts(bil.regnr);
   puts(bil.agare);
}


int less_agare(biltyp b1, biltyp b2)
{
   return (strcmp(b1.agare, b2.agare) < 0 );
}


int less_regnr(biltyp b1, biltyp b2)
{
   return (strcmp(b1.regnr, b2.regnr) < 0 );
}


int lika_bil(biltyp b1, biltyp b2)
{
   return (strcmp(b1.regnr, b2.regnr) == 0);
}
```

För att kunna anropa funktionerna för binära träd med olika aktuella funktioner för jämförelse, likhet och utskrift, exempelvis de som gäller för bilar, måste man som tidigare visats ta in *dessa funktioner som parametrar.*

Det aktuella dataobjektet *inkluderas i tree.h där också datatyp definieras*.

```
/* Specifikation av binärt träd -- tree.h */

#include "bil.h"        /* Här inkluderas aktuellt objekt */
typedef biltyp datatyp; /* Här definieras datatyp */

typedef
struct nod
{
   datatyp data;
   struct nod *left, *right;
} nodtyp;

void intotree(nodtyp **rpp, datatyp d,
              int (*is_less)(datatyp d1, datatyp d2));
/* Stoppar in d i trädet ordnat enligt is_less */

void showtree(nodtyp *rp, void (*write_data)(datatyp d));
/* Skriver ut trädet med write_data funktionen */

nodtyp *searchtree(nodtyp *rp, datatyp key,
                int (*is_equal)(datatyp d1, datatyp d2),
                int (*is_less)(datatyp d1, datatyp d2));
/* Returnerar pekare till key om den finns annars NULL */
```

```
/* Implementation av binära trädet -- tree.c */

#include <stdlib.h>
#include "tree.h"

void intotree(nodtyp **rpp, datatyp d,
              int (*is_less)(datatyp d1, datatyp d2))
{
   if ( *rpp == NULL )
   {
      *rpp = malloc(sizeof(nodtyp));
      (*rpp)->data = d;
      (*rpp)->left = NULL;
      (*rpp)->right = NULL;
   }
   else if ( is_less(d, (*rpp)->data) )
      intotree(&(*rpp)->left, d, is_less);
   else
      intotree(&(*rpp)->right, d, is_less);
}

void showtree(nodtyp *rp, void (*write_data)(datatyp d))
{
   if ( rp != NULL )
   {
      showtree(rp->left, write_data);
      write_data(rp->data);
      showtree(rp->right, write_data);
   }
}
```

```
nodtyp *searchtree(nodtyp *rp, datatyp key,
                   int (*is_equal)(datatyp d1, datatyp),
                   int (*is_less)(datatyp d1, datatyp d2))
{
   if (rp == NULL)
      return NULL;
   else if ( is_equal(key, rp->data) )
      return rp;
   else if (is_less(key, rp->data))
      return searchtree(rp->left, key, is_equal, is_less);
   else
      return searchtree(rp->right,key,is_equal,is_less);
}
```

```
/* Huvudprogram -- bilsok.c */

#include <stdio.h>
#include "bil.h"
#include "tree.h"

int main()
{
  nodtyp *bilrotp = NULL, *keyp = NULL;
  biltyp bil;
  FILE *bfil;


  bfil = fopen("bil.dat", "rb");
  fread(&bil, sizeof(biltyp), 1, bfil);
  while ( !feof(bfil) )
  {
     intotree(&bilrotp, bil, less_regnr);
     fread(&bil, sizeof(biltyp), 1, bfil);
  }

  showtree(bilrotp, write_bil);

  printf("Ge sökt registreringsnummer : ");
  gets(bil.regnr);

  keyp = searchtree(bilrotp, bil, lika_bil, less_regnr);
  if (keyp != NULL)
     write_bil(keyp->data);
  else
     printf("Bilen finns ej i registret!\n");

  return 0;
}
```

5.2 Söktabeller

Ett vanligt sätt att snabba upp sökningar är att använda sök- eller hashtabeller (hash betyder finhacka). En hashtabell utnyttjar *samma sökmetod som man använder i en telefonkatalog*. Man har alla Pettersson under samma index P. När man ska söka efter O.Pettersson, söker man inte igenom katalogen från början utan man slår upp index P direkt och letar sedan upp det eftersökta namnet.

Hashmetodiken går ut på att *man omvandlar sina data med någon hashfunktion till en mindre mängd,* som man använder som index i en tabell. I en telefonkatalog omvandlar hashfunktionen namnet till första bokstaven och använder denna bokstav som index i hashtabellen enligt:

A -- O.Andersson, I.Aronsson, …………….
B -- …………………………………………
.
P -- O.Pettersson, E.Persson, K.Pålsson, ….
.

När man använder hashsökning innebär det att man *får kollisioner* mellan data. Man har alltid flera efternamn som börjar på P. I telefonkatalogen hanterar man kollisioner genom att man stoppar in namnen sorterade i en lista vid varje index. I programmering hanterar man kollisioner antingen med *länkade listor eller med öppen adressering*.

När det gäller index i tabeller, måste man också se till att hashfunktionen *ger ett heltal som resultat* om man ska kunna använda vektorer. Har man ej hela tal att arbeta med måste man på något sätt först omvandla data till heltal. Har man namn som data kan man exempelvis ta första bokstavens ASCII-kod eller också summera ihop alla tecknens ASCII-koder och sedan använda de två sista siffrorna i dessa summor.

Antag att man har en stor mängd data i form av hela tal. Om man som hashfunktion väljer att använda de två sista siffrorna i talen som index i hashtabellen får man följande:

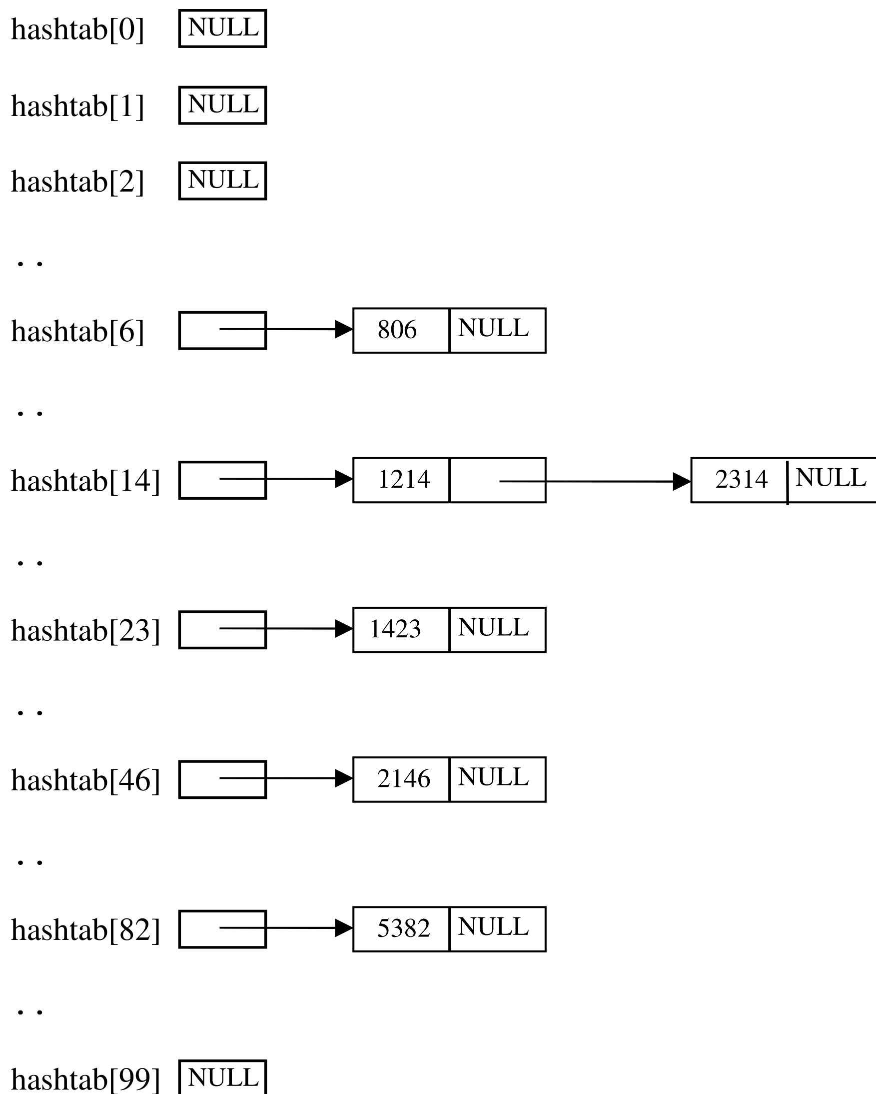
Data i mängden	Hashindex
5382	82
1423	23
806	6
2146	46
2314	14
1214	14

De två sista data visar ett exempel på en kollision. Båda får samma index i tabellen. Frågan är hur man hanterar detta. Det finns i princip två sätt att hantera kollisioner, nämligen med *länkade listor eller med öppen adressering*.

5.2.1 Kollisionshantering med länkad lista

När man använder länkade listor för att hantera kollisioner skapar man en *tabell eller vektor av pekare* till aktuell data, som man från början NULL-ställer. Då data ska instoppas använder man en LIFO-lista för varje index. Hashtabellen kommer alltså *att bestå av ett antal LIFO-listor,* vars ingångar man kommer åt med index i tabellen.

Ovanstående data instoppas enligt:



I hashtabellen, vars index fås med den aktuella hashfunktionen, sparas pekarna till alla LIFO-listor. Datatypen för att avbilda hashtabellen är alltså en *vektor av pekare till länkar*.

Ex: Skriv ett program som stoppar in heltalen ovan i en länkad hashtabell, frågar efter nyckel och skriver ut om nyckeln finns eller ej.

```
/* talhash.c */

#include <stdio.h>
#include <stdlib.h>

#define HASHTABSIZE 100

struct link
{
   int data;
   struct link *next;
};


void intolinkhash(struct link *htab[], int tal)
/* Stoppar in tal i en länkad hashtabell htab */
{
   int index;
   struct link *lp;

   /* Beräkna index och stoppa tal i länk */
   index = tal % HASHTABSIZE;
   lp = malloc(sizeof(struct link));
   lp->data = tal;

   /* Stoppa in länken i LIFO-listan */
   lp->next = htab[index];
   htab[index] = lp;
}



struct link *searchlinkhash(struct link *htab[], int nyckel)
/* Returnerar pekare till sökt länk om finns annars NULL */
{
   int index;
   struct link *lp;

   /* Gå till beräknad index */
   index = nyckel % HASHTABSIZE;
   lp = htab[index];

   /* Sök igenom listan */
   while (lp != NULL && lp->data != nyckel)
      lp = lp->next;

   /* Returnera aktuell pekare */
   return lp;
}
```

```
int main()
{
   struct link *hashtab[HASHTABSIZE];
   int i, key, vek[6] = { 5382, 1423, 806, 2146, 2314, 1214 };

   /* NULL-ställ tabellen */
   for (i = 0; i < HASHTABSIZE; i++)
      hashtab[i] = NULL;

   /* Stoppa in talen i en hashtabell */
   for (i = 0; i < 6; i++)
      intolinkhash(hashtab, vek[i]);

   /* Läs nyckel och sök i hashtabell */
   printf("Ge nyckel : ");
   scanf("%d", &key);

   if ( searchlinkhash(hashtab, key) != NULL )
      printf("Nyckeln finns!\n");
   else
      printf("Nyckeln finns ej!\n");

   return 0;
}
```

OBS! Hashtabellen måste *NULL-ställas helt först,* för att programmet ska fungera med instoppning i listan och efterföljande sökning.

OBS! Denna modell av instoppning i hashtabell stoppar in *alla nya element* i tabellen även om det råkar ha precis samma data. Vill man ej ha kopior av data i tabellen kan man exempelvis söka först och sedan stoppa in, om ej sökt data finns i tabellen.

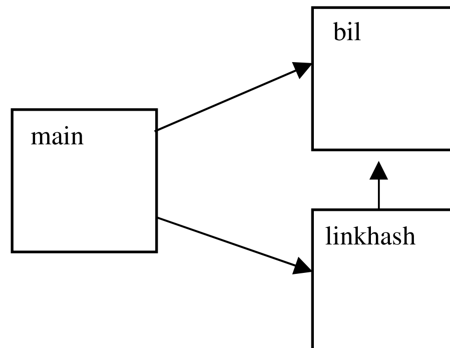
OBS! Man ska eftersträva att få så få kollisioner som möjligt i sina hashtabeller för då går sökningen snabbare. Med lämpligt val av hashfunktion och storlek på hashtabell kan man minska antalet kollisioner. I ovanstående exempel borde man exempelvis ha valt ett primtal som HASHTABSIZE, exempelvis 97.

Även hashtabeller ska man ha färdiga i sitt bibliotek. Man ska försöka att tillverka en generell *abstrakt datatyp som exporterar typen för hashtabellen och lämpliga funktioner på denna typ*, som att initiera tabellen, stoppa in i tabellen samt söka i tabellen.

För att få en generellare abstrakt datatyp för hashtabeller och kunna stoppa in även andra data än heltal i tabellen, ska man ta in en hashfunktion eller delar av den som parameter till instoppning- och sökfunktionen.

Ex: Skriv ett program som från en binärfil bestående av bilar med registreringsnummer och ägare läser in alla bilar till en länkad hashtabell, med summan av ASCII-koderna för tecknen i bilarnas registreringsnummer som hashfunktion och som sedan frågar efter ett registreringsnummer och skriver ut den eftersökta bilen om den finns.

Ett beroendediagram för programmet blir:




```
/* Specifikation av bilar - bil.h */
#ifndef BIL_H
#define BIL_H

typedef
struct bilpost
{
   char regnr[10];
   char agare[30];
} biltyp;

..

int hashfunk_bil(biltyp b);
/* Returnerar summan av ASCII-koderna för b.regnr */

#endif


/* Implementation av bilar - bil.c */

#include <stdio.h>
#include <string.h>
#include "bil.h"

..

int hashfunk_bil(biltyp b)
{
   int i, sum = 0;

   for ( i = 0; i < strlen(b.regnr); i++)
      sum += b.regnr[i];
   return sum;
```

```
}

/* Specifikation av länkad hashtabell -- linkhash.h */

#include "bil.h"
typedef biltyp datatyp;

#define HASHTABSIZE 97

typedef
struct link
{
   datatyp data;
   struct link *next;
} linktyp;

void initlinkhash(linktyp *htab[]);
/* NULL-ställer hashtabellen */

void intolinkhash(linktyp *htab[], datatyp d,
                  int (*hashfunk)(datatyp d));
/* Stoppar in d i hashtabell enligt hashfunk */

linktyp *searchlinkhash(linktyp *htab[], datatyp nyckel,
              int (*hashfunk)(datatyp d),
              int (*is_equal)(datatyp d1, datatyp d2));
/* Söker efter nyckel i hashtabell */




/* Implementation av länkad hashtabell -- linkhash.c */

#include <stdlib.h>
#include "linkhash.h"

void initlinkhash(linktyp *htab[])
{
   int i;

   for ( i = 0; i < HASHTABSIZE; i++)
      htab[i] = NULL;
}


void intolinkhash(linktyp *htab[], datatyp d,
                  int (*hashfunk)(datatyp))
{
   int index;
   linktyp *lp;

   index = hashfunk(d) % HASHTABSIZE;
   lp = malloc(sizeof(linktyp));
   lp->data = d;
   lp->next = htab[index];
   htab[index] = lp;
}
```

```
linktyp *searchlinkhash(linktyp *htab[], datatyp nyckel,
              int (*hashfunk)(datatyp d),
              int (*is_equal)(datatyp d1, datatyp d2))
{
   int index;
   linktyp *lp;

   index = hashfunk(nyckel) % HASHTABSIZE;
   lp = htab[index];
   while (lp != NULL && !is_equal(nyckel, lp->data))
      lp = lp->next;
   return lp;
}
```

```
/* Huvudprogram -- bilhash.c */

#include <stdio.h>
#include "bil.h"
#include "linkhash.h"

int main()
{
  linktyp *hashtab[HASHTABSIZE], *keyp = NULL;
  biltyp bil;
  FILE *bfil;


  initlinkhash(hashtab);
  bfil = fopen("bil.dat", "rb");
  fread(&bil, sizeof(biltyp), 1, bfil);
  while ( !feof(bfil) )
  {
     intolinkhash(hashtab, bil, hashfunk_bil);
     fread(&bil, sizeof(biltyp), 1, bfil);
  }
  fclose(bfil);

  printf("Ge sökt registreringsnummer: ");
  gets(bil.regnr);
  keyp = searchlinkhash(hashtab, bil, hashfunk_bil, lika_bil);
  if (keyp != NULL)
     write_bil(keyp->data);
  else
     printf("Bilen finns ej i registret!\n");
  return 0;
}
```

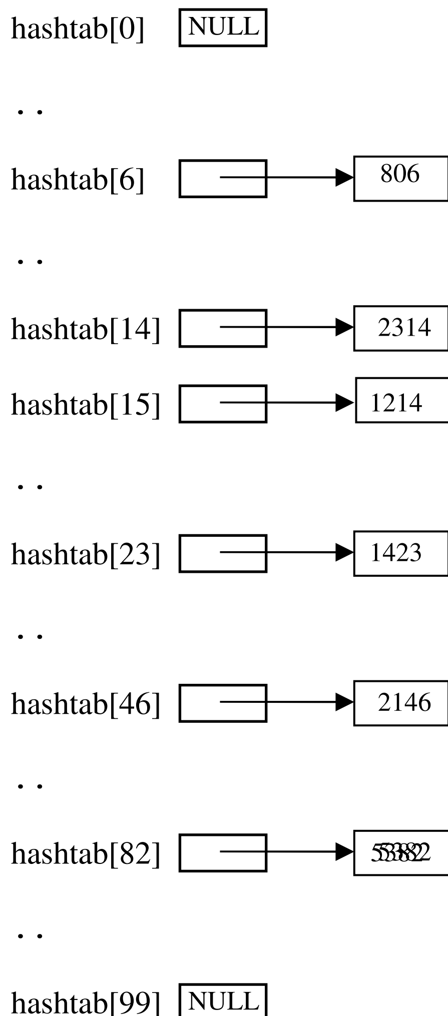
5.2.2 Kollisionshantering med öppen adressering

När man använder öppen adressering för att hantera *kollisioner, stoppar man in all data i tabellen* och använder alltså ej någon länkad lista. Då kollision inträffar hoppar man med en hoppfunktion framåt i tabellen tills man har hittat en tom plats, där data kan instoppas.

Använder man hoppfunktionen :

hopp = 1
hopp = hopp + 2

för att stoppa in talen 5382, 1423, 806, 2146, 2314 och 1214 ser en öppen adresserad hashtabell ut som:



Om man skulle sätta in ytterligare ett tal 3514 ger hoppfunktionen 3 och talet stoppas in i hashtab[18]. Var hamnar 5614? Det hamnar i hashtab[30] eftersom index 23 är upptagen.

I hashtabellen, vars index fås med den aktuella hash och hoppfunktionen, sparas pekarna till talen. Datatypen för att avbilda hashtabellen är alltså *en vektor av pekare till heltal.*

Ex : Skriv ett program som stoppar in heltalen ovan i en öppen hashtabell, frågar efter nyckel och skriver ut om nyckeln finns eller ej.

```
/* talohash.c */

#include <stdio.h>
#include <stdlib.h>

#define HASHTABSIZE 100

void intoopenhash(int *htab[], int tal)
/* Stoppar in tal i hashtabellen htab */
{
   int index, hopp;

   /* Initiera index och hopp */
   index = tal % HASHTABSIZE;
   hopp = 1;

   /* Leta fram en ledig plats */
   while (htab[index] != NULL)
   {
      index = (index + hopp) % HASHTABSIZE;
      hopp += 2;
   }

   /* Stoppa in pekare till tal */
   htab[index] = malloc(sizeof(int));
   *htab[index] = tal;
}


int *searchopenhash(int *htab[], int nyckel)
/* Returnerar pekare till sökt nyckel */
{
   int index, hopp;

   /* Initiera index och hopp */
   index = nyckel % HASHTABSIZE;
   hopp = 1;

   /* Leta upp nyckel */
   while (htab[index] != NULL && *htab[index] != nyckel)
   {
      index = ( index + hopp) % HASHTABSIZE;
      hopp += 2;
   }

   /* Returnera pekare till nyckel */
   return htab[index];
}
```

```
int main()
{
   int *hashtab[HASHTABSIZE];
   int i, key, vek[6] = { 5382, 1423, 806, 2146, 2314, 1214 };

   /* NULL-ställ hashtabellen */
   for (i = 0; i < HASHTABSIZE; i++)
      hashtab[i] = NULL;

   /* Stoppa in talen i en öppen hashtabell */
   for (i = 0; i < 6; i++)
      intoopenhash(hashtab, vek[i]);

   /* Läs nyckel och sök i den öppna  hashtabellen */
   printf("Ge nyckel : ");
   scanf("%d", &key);
   if ( searchopenhash(hashtab, key) != NULL )
      printf("Nyckeln finns!\n");
   else
      printf("Nyckeln finns ej!\n");

   return 0;
}
```

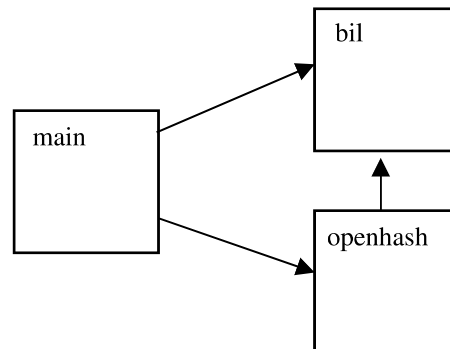
OBS! Här behövs *inga poster med next-pekare* eftersom man ej använder listor. Man kunde även ha sparat data och ej pekare till data i tabellen. Detta medför dock *svårigheter att testa om det finns plats i tabellen*. NULL är ett bättre värde att testa på.

OBS! Öppet adresserade hashtabeller är *inte lika dynamiska* som länkade. Man har ett begränsat utrymme i tabellen och som då alltid måste vara något större än antalet data. Funktionen för instoppning bör därför testa om hashtabellen har plats för ytterligare data. Man kan exempelvis, eftersom tabellindex räknas cirkulärt med tal % HASHTABSIZE, begränsa och avsluta instoppningen med ett felmeddelande då man letat efter en ledig plats ett antal varv. Man kan i loopen som letar efter tom plats sätta in en varvräknare, som ökas med 1 då index + hopp blir större än HASHTABSIZE.

Ska man använda öppen adressering för andra typer av data är det lämpligt att på motsvarande sätt som för en länkad hashtabell tillverka en *abstrakt datatyp som exporterar data och funktioner* för hantering av öppen adresserade hashtabeller.

Ex: Skriv ett program som från en binärfil bestående av bilar med registreringsnummer och ägare läser in alla bilar till en öppen adresserad hashtabell, med summan av ASCII-koderna för tecknen i bilarnas registreringsnummer som hashfunktion och som sedan frågar efter ett registreringsnummer och skriver ut den eftersökta bilen om den finns.

Ett beroendediagram för programmet blir:




```
/* Specifikation och implementation av bil se ovan */
..


/* Specifikation av öppen hashtabell -- openhash.h */

#include "bil.h"
typedef biltyp datatyp;

#define HASHTABSIZE 97


void initopenhash(datatyp *htab[]);
/* NULL-ställer hashtabellen */

void intoopenhash(datatyp *htab[], datatyp d,
                int (*hashfunk)(datatyp d));
/* Stoppar in d i hashtabell enligt hashfunk */

datatyp *searchopenhash(datatyp *htab[], datatyp nyckel,
              int (*hashfunk)(datatyp d),
              int (*is_equal)(datatyp d1, datatyp d2));
/* Söker efter nyckel i hashtabell */
```

```c
/* Implementation av öppen hashtabell -- openhash.c */

#include <stdlib.h>
#include "openhash.h"

void initopenhash(datatyp *htab[])
{
   int i;

   for ( i = 0; i < HASHTABSIZE; i++)
      htab[i] = NULL;
}


void intoopenhash(datatyp *htab[], datatyp d,
                  int (*hashfunk)(datatyp d))
{
   int index, hopp;

   index = hashfunk(d) % HASHTABSIZE;
   hopp = 1;

   while (htab[index] != NULL)
   {
      index = (index + hopp) % HASHTABSIZE;
      hopp += 2;
   }

   htab[index] = malloc(sizeof(datatyp));
   *htab[index] = d;
}



datatyp *searchopenhash(datatyp *htab[], datatyp nyckel,
                int (*hashfunk)(datatyp d),
                int (*is_equal)(datatyp d1, datatyp d2))
{
   int index, hopp;

   index = hashfunk(nyckel) % HASHTABSIZE;
   hopp = 1;

   while (htab[index]!=NULL && !is_equal(nyckel,*htab[index]))
   {
      index = ( index + hopp) % HASHTABSIZE;
      hopp += 2;
   }

   return htab[index];
}
```

```
/* Huvudprogram -- bilohash.c */

#include <stdio.h>
#include "bil.h"
#include "openhash.h"

int main()
{
  biltyp *hashtab[HASHTABSIZE], *keyp = NULL, bil;
  FILE *bfil;

  initopenhash(hashtab);
  bfil = fopen("bil.dat", "rb");
  fread(&bil, sizeof(biltyp), 1, bfil);
  while ( !feof(bfil) )
  {
     intoopenhash(hashtab, bil, hashfunk_bil);
     fread(&bil, sizeof(biltyp), 1, bfil);
  }

  printf("Ge sökt registreringsnummer: ");
  gets(bil.regnr);

  keyp = searchopenhash(hashtab, bil, hashfunk_bil, lika_bil);
  if (keyp != NULL)
     write_bil(*keyp);
  else
     printf("Bilen finns ej i registret!\n");
  return 0;
}
```

OBS! Hashing är en *sökmetod* och ingen sorteringsmetod. Data i en hashtabell ser ofta rent kaotiskt ut för ett mänskligt öga men datorn kan, via hashfunktion och hoppfunktion, snabbt leta upp det som söks.

OBS! Nackdelen med öppen adressering är att den är begränsad och att man alltid behöver dimensionera en något större tabell än antal data. Dessutom är det svårare att ta bort data från tabellen jämfört med en länkad hashtabell eller ett sökträd. Fördelen är att man har enklare datatyper och algoritmer.