Opgave 9: Sorteret database

Lav en simpel database over telefonnumre. Hver post skal indeholde navn og telefonnummer på en person. Programmet læser databasen fra en komma-separeret fil ligesom i opgave 8 og gemmer alle data i et array af structures, for eksempel således:

```
struct  Record  {
    char  name[50];
    char  address[50];
    int  age;
    unsigned  int  phone;
};

Record list[100];
```

I opgave 8 brugte vi lineær søgning til at finde en person med et bestemt telefonnummer. Nu skal vi ændre programmet så det i stedet bruger binær søgning (se kapitel 6.10.2)) i en sorteret liste.

Først læses posterne/records fra filen ind i et array af records. Programmet skal derefter sortere posterne efter telefonnummer. Du kan bruge *bubble sort* metoden som beskrevet nedenfor. Og i c bogen afsnit 6.8 s. 318

Efter at listen er sorteret kan programmet bruge binær (i c bogen s. 328 afsnit 6.10.2) søgning til hurtigt at finde en person med et bestemt telefonnummer. Binær søgning er ligeledes beskrevet nedenfor.

Hvis du ønsker det, kan du i stedet sortere listen alfabetisk efter navn og gøre det muligt at søge på navn i stedet for telefonnummer.

Sortering

Der er mange måder at sortere en liste på. En simpel algoritme kaldes *bubble sort*. Se f.eks. www.algolist.net/Algorithms/Sorting/Bubble_sort og bogen s318

Denne algoritme fungerer ved at man sammenligner hvert element med det næste i listen. Hvis et element er større end det næste i listen bytter man om på de to elementer. Dette bliver man ved med indtil alle elementer er mindre end det næste i rækken. Hvis vi f.eks. har et array af integers:

```
#define listsize 100   //defines array size
int list[listsize];   //an array of int
```

så kan vi sortere denne liste således:

```
35  void sortList() {
36  bool swapped = true;          // remember if we have swapped elements
37  int i=0;                      // index
38  int j = 0;                    // counting number of runs
39  int temp=0;                   // temporary storage during swapping
40  while (swapped) {             // repeat as long as there is something to swap j++;
41                                // count number of runs
42  swapped = false;
43  for (i = 0; i < listsize-j; i++) { // loop through list
44
45  if (list[i] > list[i+1]) { // if element is bigger than the next temp = list[i];
46                             // swap list[i] and list[i+1] list[i] = list[i+1];
47  list[i+1] = temp;
48  swapped = true;               // remember that we swapped
49  }
50  }
51  }
```

Binær søgning

Hvis man skal finde et element i en lang liste der ikke er sorteret, er man nødt til at starte fra en ende af og sammenligne elementerne et ad gangen med den søgte værdi. Dette kaldes lineær søgning. Hvis listen indeholder n elementer kan man risikere at skulle sammenligne n gange hvis det søgte element er det sidste eller det ikke findes i listen. Men hvis listen er sorteret er der en meget hurtigere metode. Den hedder binær søgning. Husker du opgave 2b "Gæt et tal"? Her skulle vi gætte et tal mellem 0 og 99 som maskinen genererede tilfældigt. Brugte du op til 100 forsøg på at gætte det rigtige tal? Nej, det tror jeg ikke du gjorde. Mon ikke du startede med 50, og hvis tallet var mindre gættede du på 25; hvis det var større gættede du på 75, osv.

Binær søgning går ud på at man hele tiden halverer intervallet af mulige elementer. Med denne metode kan man finde et element i en liste af n sorterede elementer med kun $\log_2(n)$ sammenligninger, altså 2-tals logaritmen til antallet af elementer i listen. Hvis listen f.eks. indeholder en million elementer kan man finde et bestemt element med kun 20 sammenligninger, fordi $2^{20} > 1.000.000$.

Binær søgning er hurtigere end lineær søgning. Se s. 326 i c-bogen Til gengæld skal man bruge lang tid på at sortere listen først. Derfor kan denne metode kun betale sig hvis man skal lave mange søgninger i den samme liste.

Her er et eksempel på binær søgning i ovenstående liste:

```
9   int binarySearch(int x) {    // search for an element = x int low;    // low end of search interval
10  int high;                    // high end of search interval
11  int mid;                     // middle of search interval
12  int low = 0;                 // index to first element
13  high = listsize - 1;         // index to last element
14  while (low < high) {         // continue until search interval is zero length mid = (low + high) / 2;
15                               // find middle point
16  if (x <= list[mid]) {        // compare middle element
17  high = mid;                  // new interval is from low to mid
18  }
19  else {
20  low = mid + 1;               // new interval is from mid+1 to high
21  }
22  }
23  if (list[low] == x) {        // did we find the value we were looking for? return low;
24                               // yes. return index to where the value was found
25  }
26  else {
27  return -1;                   // value was not found. return an error code
28  }
29  }
30
```

Kode vedlagt som bilag.