17. Suchen und Sortieren

Das Suchen in Datenbeständen ist ein häufig vorkommendes Programmierproblem. Eng damit zusammen hängt das Sortieren, denn normalerweise sortiert man ja Datensätze, um ein späteres Suchen zu erleichtern - entweder dem Programm oder dem Benutzer. In der praktischen Anwendung werden die unterschiedlichsten Datensätze sortiert. Das können einfach Strings sein, aber auch komplexe Strukturen. In diesem Fall muss man sich natürlich darüber im Klaren sein, welches Feld als Such- und Sortierschlüssel dienen soll.

In den folgenden Beispielen soll der jeweilige Algorithmus verdeutlicht werden. Daher werden nur Zahlen sortiert; eine Anpassung an andere Situationen ist leicht. In den Beispielen werden folgende globale Deklarationen, soweit nicht anders angegeben, vorausgesetzt:

```
#define TableSize 20;
int Table [TableSize];
```

Suchen in Tabellen

Dies ist der Standardfall. Wie in der Einleitung beschrieben, handelt es sich bei den Datensätzen, die durchsucht werden sollen, um Zahlen. Ein Array könnte beispielsweise so aussehen:

```
  12 07 56 32 44 44 18
```

Wenn man jetzt nach dem Schlüssel 56 sucht, muss die Suchfunktion als Index 2 zurückgeben. Suchet man nach 44, könnte sie entweder 4 oder 5 zurückgeben. Manchmal ist es wichtig, das *erste* Vorkommen eines Schlüssel zu finden (also 4). Ausserdem kann es vorkommen, dass ein Schlüssel gar nicht gefunden wird. Die vorgestellten Funktionen signalisieren dies, indem sie den Index -1 zurückgeben.

Lineare Suche

Wenn man nichts über die Anordnung der Komponenten im Array weiss, bleibt einem *nichts anderes übrig*, als sie beginnend bei der ersten Komponente linear zu durchsuchen, bis man entweder die gesuchte Komponente gefunden hat, oder bis das Ende des Arrays erreicht wurde. Im statistischen Mittel werden bei N Tabelleineinträgen N/2 Suchschritte benötigt, im günstigsten Fall nur einer und im schlechtsten Fall N Schritte.
Um nicht jedesmal prüfen zu müssen, ob das Ende des Arrays schon erreicht ist, wenden wir einen kleinen Trick an: Am Ende des Arrays fügen wir noch ein Element an, das genau dem gesuchten Wert entspricht - auf diese Weise wird dann jedesmal ein Wert gefunden. Natürlich muss anschließend geprüft werden, ob der gefundene Wert nun ein "echter" Treffer oder die eigene Endemarke war. Diese Endemarke wird "Sentinel" genannt. Die Konstante `Tablesize` muss entsprechend großzügig gewählt werden.
Die Funktion `LinSearch` hat als Parameter den Suchschlüssel und gibt den gefundenen Index zurück:

```
int LinSearch(int iKey, int iaTable[])
  {
     int iIndex;
     iaTable[TableSize] := iKey;
     iIndex := 0;
     while (iaTable[iIndex] != iKey)
```

```
      iIndex++;
    if (iIndex = TableSize)
    iIndex := -1;
    return(iIndex);
  }
```

Wie man sieht, wird im ersten Schritt der Sentinel gesetzt, dann wird der Index auf 0 initialisiert und dann so lange erhöht, bis der Key gefunden wurde. Zum Schluß wird noch geprüft, ob der Index außerhalb des gültigen Bereichs liegt. In diesem Fall wird der Index auf -1 korrigiert, denn das ist der Wert für "nicht gefunden". Dass der Algorithmus stets das erste Vorkommen des Keys findet, ist offensichtlich.

Binäre Suche

Um die Suche zu beschleunigen, müssen die Elemente in der Tabelle sortiert vorliegen. Dies kann man entweder durch eines der später beschriebenen Sortierverfahren erreichen, oder indem man neue Elemente gleich an der richtigen Stelle einfügt. Beides benötigt zusätzlichen Aufwand, spart aber Zeit beim Suchen. Wie nutzt man nun die Ordnung in der Tabelle aus?

Man sucht das mittlere Element der Tabelle und prüft, ob es größer oder kleiner als dder Key ist. Ist es größer oder gleich, braucht man nur noch in der unteren Hälfte der Tabelle zu suchen, ist es kleiner, sucht man nur noch in der oberen Hälfte. Irgendwann ist die Größe des Suchintervalls auf 1 geschrumpft, und dann hat man das gesuchte Element gefunden, oder es ist nicht vorhanden. Die Zahl der Suchschritte reduziert sich bei N Tabelleneinträgen auf `ln(N)` (ln: Logarithmus zur Basis 2).

Der folg}e Algorithmus benutzt die Variablen `L(inks)` und `R(echts)`, um den Suchbereich zu definieren, sowie `Middle`, um die Mitte zu markieren. Soll oberhalb der Mitte weitergesucht werden, wird `L` zu `Middle + 1`, wird im unteren Teil weitergesucht, so ergibt sich `R = Middle`.

```
int BinSearch (int Key, int Table[])
  {
  int L, R, Middle;

  L := 1;
  R := TableSize + 1;
  while (L < R)
    {
    Middle := (L + R) / 2;
    if (Table[Middle] < Key) L := Middle + 1;
    else                     R := Middle;
    }
  if (Table[R] == Key)
return(R);
  else
return(-1);
  }
```

Wenn die Funktion das gesuchte Element früher als erwartet findet (z. B. wenn es genau in der Mitte liegt), könnte sie eigentlich abbrechen. Es ist dann nicht mehr garantiert, dass man das erste Vorkommen von `Key` findet.

Sortieren von Tabellen

Das Sortieren ist eines der wichtigsten Gebiete der Datenverarbeitung. Etwa 25% aller Rechenzeit im kommerziellen Bereich wird auf das Sortieren von Daten verwendet. An Beispielen zum Sortieren mangelt es nicht:

- Statistiken
- Telefonbücher, Wörterbücher
- Listen, ...

Die Problemstellung: Gegeben ist ein Array mit Komponenten, die nach ihrem Schlüssel sortiert werden können, über deren momentane Anordnung jedoch nichts bekannt ist (das heißt auch, dass sie theoretisch schon sortiert sein oder in der umgekehrten Reihenfolge vorliegen könnten). Der Sortieralgorithmus soll die Elemente in die richtige Reihenfolge bringen.

Definition:

"Unter Sortieren versteht man allgemein den Prozess des Anordnens einer gegebenen Menge von Objekten in einer bestimmten Ordnung."

Das Sortieren ist ebenfalls einer der Grundalgorithmen in der Informatik insbesondere für die effiziente Speicherung von Informationen hinsichtlich der Auswertung bzw. Suche. Bei den Sortierverfahren unterscheidet man die

- ***interne Sortierung:*** als Bearbeitung von Listen im Hauptspeicher,
- ***externe Sortierung:*** als Bearbeitung von Listen, die sich auch auf externen Speichermedien (Magnetband, Festplatte) befinden können.

Bubblesort

Bubblesort ist einer der einfachsten Sortieralgorithmen. Im ersten Durchgang wird das Array vom Ende bis zum Anfang bearbeitet und bei jedem Schritt die aktuelle Komponente mit der nächsten verglichen. Ist die untere Komponente kleiner als die obere, werden die beiden vertauscht. Die größere Komponente behält also ihren Platz und die jeweils kleinere wandert weiter nach oben.
Im nächsten Durchlauf wird dann die zweitkleinste Komponente gesucht und nach oben durchgereicht. Logischerweise muss dieser Durchlauf nicht ganz bis zum Anfang gehen, sondern nur bis Index-1. Der dritte Durchlauf läuft nur noch bis Index-2, usw. - bis das Verfahren abgeschlossen ist. Der Name "Bubblesort" kommt daher, dass die kleineren Komponente aufsteigen wie Blasen in einer Flüssigkeit.

```
void Bubblesort (int Table[])
  {
  int x;
  int Run, Index;
  for (Run = 1;  Run < TableSize; Run++)
   {
```

```
      for (Index = TableSize; Index >= Run; Index--)
     {
          if (Table[Index] < Table[Index - 1])
           {
                x := Table[Index];
                Table[Index] := Table[Index - 1];
                Table[Index - 1] := x;
           }
     }
  }
}
```

Deutlich zu erkennen ist der Vertauschungsvorgang, bei dem die "untere" Komponente zunächst in der Variablen x zwischengespeichert wird, dann den Wert der "oberen" zugewiesen bekommt und anschließend x im "oberen" Index gespeichert wird.

Bubblesort genießt keinen besonders guten Ruf, weil es sehr langsam ist. Trotzdem ist es häufig anzutreffen, nicht zuletzt wegen seines hohen Bekanntheitsgrades.

Sortieren durch Auswählen

Selectionsort ist ebenfalls ein sehr einfacher Algorithmus. Das Prinzip dahinter besteht darin, dass das Array von vorn nach hinten durchlaufen wird, und für jeden Platz die passende Komponente herausgesucht wird.

Man beginnt mit `Table[0]` und durchläuft dann das Array. Dabei merkt sich die Funktion das kleinste Element. Dieses vertauscht sie dann mit der ersten Komponente, so dass nun die kleinste Komponente ganz vorne steht. Im nächsten Durchgang beginnt man bei `Table[1]` und durchsucht wieder das Array nach der kleinsten Komponente, wobei natürlich `Table[0]` unberücksichtigt bleibt. Im dritten Durchgang bleiben dann die ersten beiden Tabellenelemente unberührt usw.

```
void SelectionSort (int Table[])
 {
 int Komponente, Smallest;
 int Index, Smallidx;

 for (Komponente = 0; Komponente < TableSize; Komponente++)
  {
     Smallidx := Komponente;
     Smallest := Table[Smallidx];
     for (Index = Komponente + 1; Index <= TableSize; Index++)
     {
          if (Table[Index] < Smallest)
           {
              Smallest := Table[Index];
              Smallidx := Index;
           }
    }
    if (Smallidx != Komponente)
    {
          Table[Smallidx] := Table[Komponente];
          Table[Komponente] := Smallest;
   }
  }
}
```

DI Harald Sander

In der Variablen `Smallest` merkt sich der Algorithmus den Wert die bislang kleinste Komponente, in `Smallidx` ihre Position. Beide werden zunächst auf die Komponente initialisiert, deren Position besetzt werden soll. Dann wird das übrige Array durchsucht, und beim Auftauchen einer kleineren Komponente entsprechend aktualisiert. Die Vertauschungsvorgang wird nur ausgeführt, wenn tatsächlich eine neue Position gefunden wurde. Selectionsort ist schneller als Bubblesort (ca. Faktor 2).

Sortieren durch Einfügen

Es wird angenommen, dass ein Teil am Anfang des Arrays schon sortiert ist (zu Beginn ist dieser Teil natürlich 0 Komponenten groß!). Nun wird die erste Komponente aus dem unsortierten Teil genommen und an der richtigen Stelle in den sortierten Teil eingefügt. Der sortierte Teil wächst dabei natürlich um eine Komponente, bleibt aber sortiert, wohingegen der unsortierte Teil um eine Komponente schrumpft.

Der Algorithmus durchläft die Arraykomponenten von Anfang bis Ende. Für jede Komponente geschieht nun folgendes: Von seiner Position aus bewegt er sich im sortierten Teil in Richtung Tabellenanfang, solange die Komponenten noch größer oder gleich der in Frage stehenden Komponente sind.

Dabei wird jede Komponente, die "überquert" wird, nach hinten verschoben. So entsteht an der jeweils aktuellen Position eine freie Array-Komponente. Diese "Lücke" wird dann an der richtigen Position mit dem einzufügenden Wert gefüllt. Auf diese Weise verbindet der Algorithmus die Suche nach der richtigen Position mit dem Verschieben der darüberliegenden Komponenten.

```
void InsertionSort (int Table[])
 {
  int x;
  int Komponente, Index;

  for (Komponente = 1; Komponente <= TableSize; Komponente++)
   {
   x := Table[Komponente];
   Index := Komponente;
   while ((Index > 0) && (Table[Index - 1] >= x))
   {
            Table[Index] := Table[Index - 1];
            Index--;
    }
   Table[Index] := x;
   }
 }
```

Wie man sieht, wird die aktuelle Komponente in x gespeichert. Dann beginnt das Verschieben, bis entweder eine Komponente gefunden wird, die kleiner als x ist, oder bis man bei Index 0 angelangt ist. Im letzteren Fall hat man offenbar keine kleinere Komponente im sortierten Teil und plaziert die Komponente an Position 0.

Shellsort

Shellsort ist ein Kompromiss aus Einfachheit und Geschwindigkeit. Das Array wird zunächst in mehrere Gruppen aufgeteilt, zwischen deren Mitgliedern der gleiche Abstand besteht. Angenommen, man verwendet den Abstand 3, dann gehören die Komponenten 1, 4, 7, ... in

eine Gruppe, genauso 2, 5, 8, ..., 3, 6, 9, ... usw. Diese Gruppen werden nun einzeln sortiert, dann wird der Abstand verringert und die Vorgehensweise wiederholt. Das macht man so lange, bis der Abstand 1 ist, so dass im letzten Schritt gar keine Unterteilung mehr stattfindet.

Zum Sortieren der Gruppen wird ein Bubblesort-ähnlicher Algorithmus verwendet. Für jeden Schritt gibt die Variable `Gap` den Abstand zwischen den Komponenten an. Die Variable `Index` läuft dann von `Gap + 1` bis zum Ende der Tabelle durch. Für jeden Wert von `Index` wird ein Sortierlauf gestartet, mit `Index` als oberer Grenze. Welche Gruppe dabei sortiert wird, hängt also von Index ab. Die Variable `j` wird auf die nächstkleinere Komponente der Gruppe initialisiert (also `Index - Gap`), dann wird es mit der nächsthöheren verglichen und gegebenenfalls vertauscht. Dann geht `j` wieder einen Schritt nach unten (Schrittweite `Gap`), usw., bis `j` kleiner 0 ist.

Wenn `Index` auf die n-te Komponente einer Gruppe verweist, sind alle Komponenten (dieser Gruppe) unterhalb von `Index` sortiert. Ist `Index` bis zum Ende des Arrays durchlaufen, sind auch alle Gruppen sortiert.

```
void ShellSort (int Table[])
 {
 int x;
 int Gap, Index, j;
 Gap := TableSize / 2;
 while (Gap > 0)
 {
    for (Index = Gap + 1;  Index <= TableSize; Index++)
    {
            j := Index - Gap;
            while (j > 0)
              {
                  if (Table[j] <= Table[j + Gap])
                      j := 0;
                  else
                 {
                     x := Table[j];
                     Table[j] := Table[j + Gap];
                     Table[j + Gap] := x;
                }
                j = J - Gap;
              }
  }
  Gap := Gap / 2;
  }
 }
```

Shellsort zählt zu den schnelleren Algorithmen. Leider kommt es mit *sehr* großen Arrays überhaupt nicht zurecht.

Quicksort

Quicksort ist in den meisten Fällen der schnellste Algorithmus. Man greift sich eine beliebige Komponente des Arrays heraus - beispielsweise die mittlere - und teilt das Array in zwei Gruppen: Eine mit den Komponenten, die größer sind als die herausgegriffene, und eine mit den kleineren (oder gleichen). Diese beiden Hälften übergibt man wieder an Quicksort. Es handelt sich also um eine rekursive Funktion. Irgendwann sind die Arrays nur noch eine Komponente groß und damit sortiert. Um nicht bei jedem Aufruf immer Arrays erzeugen zu

müssen, arbeitet Quicksort mit linker und rechter Grenze. Dafür müssen dann natürlich alle Komponenten, die größer sind als die herausgegriffene, in den unteren Teil verschoben werden, die anderen in den oberen Teil. Darin besteht die eigentliche Arbeit von Quicksort:

```
void QSort (int Links, int Rechts, int Table[])
 {
   int i, j;
   int x, w;
   i := Links;
   j := Rechts;
   x := Table[(Links + Rechts) / 2];
   do
  {
     while (Table[i] < x) i++;
     while (Table[j] > x) j--;
     if (i <= j)
    { /* Element vertauschen */
        w := Table[i];
        Table[i] := Table[j];
        Table[j] := w;
        i++; j--;
    }
  }
  while (i <= j);
  if (Links < j) QSort (Links, j, Table);
  if (Rechts > i) QSort (i,Rechts, Table);
 }
```

Quicksort ist, wie eingangs gesagt, in der Regel das schnellste Suchverfahren. Im schlimmsten Fall jedoch, wenn das Array bereits sortiert ist, fällt Quicksort leider auf das Niveau von Bubblesort herab.

Praktischer Einsatz von Sortierverfahren

Suchen mit `bsearch`

Die Bibliotheksfunktion `bsearch()` führt eine binäre Suche in einem Datenarray durch, wobei sie nach einem Array-Element Ausschau hält, das mit einem gesuchten Wert (dem Schlüssel oder "key") übereinstimmt. Um `bsearch()` anwenden zu können, muss das Array in aufsteigender Reihenfolge sortiert sein. Außerdem muss das Programm die Vergleichsfunktion bereitstellen, die `bsearch()` benötigt, um festzustellen, ob ein Datenelement größer, kleiner oder gleich einem anderen Element ist. Der Prototyp von `bsearch()` steht in `stdlib.h` und lautet:

```
void *bsearch(const void *key, const void *base, size_t num, size_t size,
     int (*cmp)(void *elem1, void *elem2));
```

Dieser Prototyp ist ziemlich komplex. Deshalb sollten Sie ihn sorgfältig studieren. Das Argument `key` ist ein Zeiger auf das gesuchte Datenelement und `base` ein Zeiger auf das erste Element in dem zu durchsuchenden Array. Beide werden mit dem Typ `void` deklariert, so dass sie auf irgendein beliebiges C-Datenobjekt zeigen können.

Das Argument `num` ist die Anzahl der Arraykomponenten und `size` die Größe einer Komponente in Byte. Üblicherweise wird der `sizeof()`-Operator verwendet, um die Werte für `num` und `size` anzugeben.

`cmp` ist ein Zeiger auf die Vergleichsfunktion. Dabei kann es sich um eine vom Programmierer aufgesetzte Funktion handeln, oder - wenn Stringdaten durchsucht werden - um die Bibliotheksfunktion `strcmp()`. Die Vergleichsfunktion muss die folgenden zwei Kriterien erfüllen:

- Sie muss Zeiger auf zwei Datenelemente übernehmen.
- Sie muss einen der folgenden `int`-Werte zurückgeben:
  - `< 0`: Parameter 1 ist kleiner als Parameter 2.
  - `0`: Parameter 1 ist gleich Parameter 2.
  - `> 0`: Parameter 1 ist größer als Parameter 2.

Der Rückgabewert von `bsearch()` ist ein Zeiger vom Typ `void`. Die Funktion gibt einen Zeiger auf die erste Array-Komponente zurück, die dem Schlüssel entspricht, oder `NULL`, wenn keine Übereinstimmung gefunden wird. Der Typ des zurückgegebenen Zeigers muss entsprechend umgewandelt werden, bevor der Zeiger verwendet werden kann.

Der `sizeof()`-Operator kann die `num`- und `size`-Argumente wie folgt bereitstellen. Wenn `array[]` das zu durchsuchende Array ist, dann liefert die Anweisung

```
sizeof(array[0]);
```

den Wert für `size`, das heißt die Größe eines Array-Elements in Byte. Da der Ausdruck `sizeof(array)` die Größe eines ganzen Arrays in Byte zurückliefert, können Sie mit der folgenden Anweisung den Wert von `num`, der Anzahl der Elemente im Array, ermitteln:
`sizeof(array)/sizeof(array[0])`

Sortieren mit `qsort`

Die Bibliotheksfunktion `qsort()` ist eine Implementierung des Quicksort-Algorithmus, der von C.A.R. Hoare erfunden wurde. Diese Funktion sortiert ein Array in eine vorgegebene Reihenfolge. Normalerweise wird aufsteigend sortiert, aber `qsort()` kann auch absteigend sortieren. Der Prototyp dieser Funktion ist in `stdlib.h` definiert und lautet:

```
void qsort(void *base, size_t num, size_t size,
           int (*cmp)(void *elem1, void *elem2));
```

Das Argument `base` zeigt auf das erste Element in dem Array, `num` ist die Anzahl der Komponenten im Array und `size` die Größe einer Array-Komponente in Byte. Das Argument `cmp` ist eine Zeiger auf eine Vergleichsfunktion. Für diese Vergleichsfunktion gelten die gleichen Regeln wie bei `bsearch()`. Oft verwendet man für `bsearch()` und `qsort()` die gleiche Vergleichsfunktion. Die Funktion `qsort()` hat keinen Rückgabewert.

Das folgende Listing veranschaulicht den Einsatz von `qsort()` und `bsearch()`. Das Programm sortiert und durchsucht ein Array von Integer-Werten. Zu Beginn des Programms können Sie bis zu `MAX` Werte eingeben. Die Werte werden sortiert und dann in der neuen Reihenfolge ausgegeben. Danach können Sie einen Wert eingeben, nach dem im Array gesucht werden soll. Eine abschließende Meldung informiert Sie darüber, ob der Wert im Array gefunden wurde oder nicht.

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

#define MAX 20

int intvgl(const void *v1, const void *v2)
```

```
  { return (*(int *)v1 - *(int *)v2); }

int main(void)
 {
 int arr[MAX], count, suche, *zgr;

 /* Werte einlesen */
 printf("Geben Sie %d Integer-Werte ein.\n", MAX);
 for (count = 0; count < MAX; count++)
  scanf("%d", &arr[count]);

 /* Sortiert das Array in aufsteigender Reihenfolge */
 qsort(arr, MAX, sizeof(arr[0]), intvgl);

 /* Gibt das sortierte Array aus. */
 for (count = 0; count < MAX; count++)
  printf("\narr[%d] = %d.", count, arr[count]);

 /* Suchwert eingeben */
 printf("Geben Sie einen Wert für die Suche ein: ");
 scanf("%d", &suche);

 /* Suche durchfuehren */
 zgr = (int *)bsearch(&suche, arr, MAX, sizeof(arr[0]),intvgl);

 if ( zgr != NULL )
  printf("%d bei arr[%d] gefunden.\n", suche, (zgr - arr));
 else
  printf("%d nicht gefunden.\n", suche);
 return(0);
 }
```

Das nächste Listing macht im Prinzip das Gleiche wie das vorstehende, nur dass diesmal Strings sortiert und gesucht werden. Die Strings werden sortieren, indem das Array der Zeiger sortiert wird. Dazu muss allerdings die Vergleichsfunktion angepasst werden. Der Vergleichsfunktion werden Zeiger auf die zwei Elemente in dem Array übergeben, die verglichen werden. Sie wollen jedoch das Array von Zeigern nicht nach den Werten der Zeiger selbst, sondern nach den Werten der Strings, auf die die Zeiger verweisen, sortieren.

Deshalb müssen Sie eine Vergleichsfunktion verwenden, der Zeiger auf Zeiger übergeben werden. Jedes Argument an `vergl()` ist ein Zeiger auf eine Array-Komponente und da jede Komponente selbst ein Zeiger auf einen String ist, so ist das Argument ein Zeiger auf einen Zeiger. Innerhalb der Funktion selbst dereferenzieren Sie die Zeiger, so dass der Rückgabewert von `vergl()` von den Werten der Strings abhängt, auf die verwiesen wird.

Die Tatsache, dass die Argumente, die `vergl()` übergeben werden, Zeiger auf Zeiger sind, schafft auch noch in anderer Hinsicht Probleme. Sie speichern den Suchbegriff in `puffer[]` und Sie wissen auch, dass der Name eines Arrays (in diesem Fall `puffer`) ein Zeiger auf das Array ist. Sie müssen jedoch nicht `puffer` selbst übergeben, sondern einen Zeiger auf `puffer`. Das Problem dabei ist, dass `puffer` eine Zeigerkonstante ist und keine Zeigervariable. `puffer` selbst hat keine Adresse im Speicher; es ist ein Symbol, das zur Adresse des Arrays auswertet. Deshalb können Sie keinen Zeiger erzeugen, der auf `puffer` zeigt, indem Sie den Adressoperator vor `puffer` (wie in `&puffer`) verwenden.

Wie verhält man sich in einem solchen Fall? Zuerst erzeugen Sie eine Zeigervariable und weisen ihr den Wert von `puffer` zu. In unserem Programm trägt diese Zeigervariable den

Namen `suche`. Da `suche` eine Zeigervariable ist, hat sie eine Adresse, und Sie können einen Zeiger erzeugen, der diese Adresse als Wert aufnimmt. Wenn Sie schließlich `bsearch()` aufrufen, übergeben Sie als erstes Argument `suche1` - einen Zeiger auf einen Zeiger auf den Suchstring. Die Funktion `bsearch()` übergibt das Argument an `vergl()`, und alles läuft wie geschmiert.

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

#define MAX 20

int vergl(const void *s1, const void *s2)
  { return (strcmp(*(char **)s1, *(char **)s2)); }

int main(void)
  {
  char *daten[MAX], puffer[80], *zgr, *suche;
  int count;

  /* Eine Liste von Wörtern einlesen. */
  printf("Geben  Sie %d Wörter ein.\n",MAX);
  for (count = 0; count < MAX; count++)
    {
    printf("Wort %d: ", count+1);
    fgets(puffer,80,stdin);
    puffer[strlen(puffer)-1] = '\0';
    daten[count] = malloc(strlen(puffer)+1);
    strcpy(daten[count], puffer);
    }

  /* Sortiert die Woerter (genauer: die Zeiger) */
  qsort(daten, MAX, sizeof(daten[0]), vergl);

  /* Die sortierten Woerter ausgeben */
  for (count = 0; count < MAX; count++)
    printf("\n%d: %s", count+1, daten[count]);

  /* Suchbegriff einlesen */
  printf("\n\nGeben Sie einen Suchbegriff ein: ");
  fgets(puffer,80,stdin);
  puffer[strlen(puffer)-1] = '\0';

  /* Fuehrt die Suche durch */
  /* &suche ist Zeiger auf den Zeiger auf den Suchbegriff.*/
  suche = puffer;
  zgr = bsearch(&suche, daten, MAX, sizeof(daten[0]), vergl);
  if (zgr != NULL)
    printf("%s gefunden.\n", puffer);
  else
    printf("%s nicht gefunden.\n", puffer);
  return(0);
  }
```