Datenorganisation

Literatur

- Niklaus Wirth, Algorithmen und Datenstrukturen, Teubner-Verlag
- Donald E. Knuth, The art of computer programming, Vol. 3, Sorting and searching, Addison-Wesley, 1973
- K. Mehlborn, Datenstrukturen und effiziente Algorithmen. Band 1: Sortieren und Suchen, Teubner
- Ottmann/Widmayer, Algorithmen und Datenstrukturen

1. Sortieralgorithmen

Der wichtigste Anwendungsbereich von Sortieralgorithmen ist die statistische Auswertung großer Datenmengen, dabei ist der Algorithmus zur Schaffung einer sortierten Datenmenge nötig. In neuerer Zeit hat zusätzlich die Präsentation von Daten, und sei es nur die alphabetisch sortierte Ausgabe von Dateinamen eines Verzeichnisses, auf Grund der immer komfortabler werdenden Programmoberflächen an Bedeutung gewonnen. Ein Sortieralgorithmus ist dann sinnvoll anwendbar, wenn eine Vielzahl von Daten in eine geordnete Reihenfolge gebracht werden soll.

Die Vielfalt der Algorithmen beruht auf den verschiedenen Rahmenbedingungen der jeweiligen Sortierprobleme. Bis in die sechziger Jahre hinein konzentrierte sich das Forschungsinteresse auf die Entwicklung von Algorithmen. Die folgenden Jahre waren im wesentlichen durch theoretische Untersuchungen der entwickelten Algorithmen gekennzeichnet.

In heutiger Zeit hat sich das Hauptinteresse der Forschung auf die Übertragung der Sortieralgorithmen auf Parallelrechnerstrukturen verschoben. Weiterhin wird der Einfluß einer Vorsortierung auf das Laufzeitverhalten eines Algorithmus intensiv untersucht.

1.1. Terminologie

Für den Sortieralgorithmus ist der spezielle Datentyp nicht wichtig. Man kann also mit derselben Prozedur eine Menge bestehend aus Äpfeln und Birnen, aus Wörtern, aus booleschen Ausdrücken oder aus Zahlen sortieren.

Gegeben seien die n Elemente: Array

```
a[1], a[2], ..., a[n].
```

Das Sortieren besteht im Umordnen dieser Elemente in eine Reihenfolge:

```
a[k1], a[k2], ..., a[kn],
```

so dass für eine gegebene **Ordnungsfunktion f** gilt:

```
f(a[k1]) <= f(a[k2]) <= ...<=f(a[kn]).
```

Gewöhnlich wird die Ordnungsfunktion nicht nach einer bestimmten Vorschrift berechnet, sondern als explizite Komponente (Feld) eines jeden Elementes (item) gespeichert. Ihr Wert heißt **Schlüssel (key)** des Elementes. Folglich ist die Strukturart RECORD für die Darstellung der Elemente `a[i]` besonders geeignet.

Wir definieren deshalb einen Typ **item**, wie er in allen folgenden Sortieralgorithmen verwendet wird:

```
TYPE item    = RECORD key: INTEGER;
               {andere Komponenten sind hier aufzuführen}
               END
TYPE index   = 0 .. n;
VAR a: ARRAY [1 .. n] OF item;
```

Die anderen Komponenten stellen wichtige Daten bezüglich der Elemente der Sammlung dar. Der Schlüssel dient nur zur Identifizierung der Elemente. Für unsere Sortier-Algorithmen ist jedoch der Schlüssel die **einzige** wichtige Komponente, und es ist nicht notwendig, irgendwelche weiteren Komponenten zu definieren. Die Wahl von INTEGER als Schlüsseltyp ist willkürlich. Natürlich könnte ebenso jeder andere Typ verwendet werden, für den eine vollständige Ordnungsrelation definiert ist.

1.2. Klassifizierung der Algorithmen

Eine Sortiermethode heißt **stabil**, wenn die relative Ordnung der Elemente mit gleichen Schlüsseln beim Sortieren unverändert bleibt. Stabilität beim Sortieren ist oft erwünscht, wenn die Elemente bereits nach einem zweitrangigen Schlüssel geordnet (sortiert) sind, d.h. nach Eigenschaften, die nicht durch den (Haupt-) Schlüssel selbst ausgedrückt werden.

Die Sortieralgorithmen sind anhand weiterer Merkmale klassifizierbar. Dazu gehören der Speicherbedarf für den Algorithmus, die Laufzeiten im Mittel, im besten und schlechtesten Fall, das Sortierprinzip und die optimale Datenstruktur.

Man unterscheidet weiterhin zwischen **interner** und **externer** Sortierung. Interne Sortieralgorithmen setzen voraus, daß alle Daten im RAM des Computers Platz finden. Bei externer Sortierung bestimmen die zeitlichen Eigenschaften des externen Speichermediums für Datenübertragung oder Positionierung die Effizienz des Algorithmus. Das klassische Beispiel für ein externes Speichermedium war in der damaligen Zeit das Magnetband.

Die Algorithmen werden auch bezüglich ihres Speicherbedarfes klassifiziert. Wenn unabhängig von Zahl und Typ der Daten ein Verfahren wie Heapsort oder Bubblesort für den Programmcode und die Hilfsvariablen nur einen konstant großen Raum im Speicher einnimmt, so handelt es sich um ein **In-situ**-Verfahren. Rekursive Verfahren wie Quicksort oder listenorientierte Verfahren wie das Listmerge-Verfahren mit einem zusätzlichen Pointer pro Element zählen daher nicht zu den In-situ-Verfahren.

1.3. Vorsortierung

Eine Folge mit vier Elementen (4, 3, 2, 1) soll aufsteigend sortiert werden. Man sieht sofort, daß diese Folge invers sortiert ist. Nun haben sich die Informatiker in den letzten Jahren mit der Frage auseinandergesetzt, inwieweit ein Sortierverfahren eine solche Vorsortierung ausnutzt. Um allgemeinere Aussagen treffen zu können, sind dazu drei Maßzahlen eingehender untersucht worden. Als erstes bestimmt man die Zahl der Inversionen (Fehlstellungen). In obigen Beispiel sind sechs Paare (4, 3), (4, 2), (4, 1), (3, 2), (3, 1), (2, 1) fehlgeordnet. Mit der Zahl der Inversionen hat man somit ein Maß für die globale Vorsortierung, da mit Zunahme der Inversionen auch die Elemente immer weiter von ihrem korrekten Platz entfernt sind. Bei einer inversen Sortierung nimmt die Maßzahl ein Maximum an.

Im Gegensatz dazu steht die Maßzahl der längsten aufsteigenden Subfolge LAS. Im obigen Beispiel hat LAS (longest ascending subsequence) den Wert eins, da entweder (4), (3), (2) oder (1) als längste Subfolge auftreten können. In vielen Arbeiten wird statt der Maßzahl LAS die Maßzahl REM (removes) verwendet. REM ist dabei einfach als die Differenz

zwischen Datenlänge n und dem LAS-Wert definiert. Die Maßzahl REM wird vor allem deswegen bevorzugt, weil sie wie die Inversionenzahl mit zunehmender Vorsortierung kleiner wird.

1.4. Laufzeiten

Unter der **Ordnung O(N)** versteht man die Funktion, die die Zahl der Vergleiche oder der Elementbewegungen für große Datenmengen mit der Größe N im wesentlichen bestimmt. Im Prinzip ist die Funktion der Ordnung der Grenzwert der Vergleiche oder Elementbewegungen für N gegen unendlich.

Im Idealfall einer schon sortierten Folge mit N Daten benötigt man lediglich N - 1 Vergleiche. Die meisten Algorithmen nutzen diesen Umstand jedoch nicht, daher benötigen sie eine größere Laufzeit.

Für den worst case gibt es ebenso eine untere Grenze, die der folgende Satz festlegt:

„Jedes allgemeine Sortierverfahren kann zum Sortieren von N verschiedenen Schlüsseln sowohl im schlechtesten Fall als auch im Mittel mit N * log(N) Vergleichen auskommen." Für den Beweis dieser Behauptung sei auf „Ottmann/Widmayer, Algorithmen und Datenstrukturen" verwiesen. Ein Verfahren wird als **worst-case-optimal** bezeichnet, wenn bei beliebiger Anordnung der Elemente eine Laufzeitordnung von O(N * log(N)) garantiert ist.

1.5. Elementare Algorithmen

Die elementaren Algorithmen stellen keinerlei Forderungen an die Datenstruktur. Deshalb können die Elemente außer in wichtigen Datenstrukturen wie Arrays auch in einer Menge (die keine Reihenfolge festlegt) zusammengefaßt sein.

Zu den elementaren Verfahren zählen das Sortieren durch Auswählen, das Sortieren durch Einfügen und auch Bubblesort. Bubblesort hat mit den anderen elementaren Verfahren die gleiche Laufzeitordnung $O(N^2)$ im Falle der zufälligen Datenanordnung und die stabile Sortierung gemeinsam.

1.6. Direktes Einfügen

Das Sortieren durch Einfügen nimmt ein beliebiges Element aus der Menge und fügt es an der richtigen Stelle in der sortierten Folge ein. Kurz zusammengefasst ergeben sich folgende Schritte:

1. Eine zu sortierende Reihe wird aufgeteilt in eine linke sortierte und eine rechte unsortierte Reihe.
2. Am Anfang enthält der sortierte Teil nur ein Element
3. Bei jedem Schritt wird das linkeste Element des unsortierten Teils als neues Element genommen, und alle Elemente des sortierten Teils, die größer sind als das neue um eins nach rechts verschoben (beginnend von rechts). Anschließend wird das neue Element an die freigewordene Stelle eingesetzt.

-> Folie: Beispiel-Sortierung + Stabilitäts-Demo - 13

Algorithmuseigenschaften:

- Speicherbedarf — In-situ-Verfahren
- Stabilität — gewährleistet
- Vorsortierung — relativ optimal bei Vorsortierung
- Datenstruktur — Array, Liste, Menge oder andere

Laufzeitverhalten:

	Vergleiche	Elementbewegung
• Datenanordnung		
• sortiert	O(N)	O(N)
• invers sortiert	$O(N^2)$	$O(N^2)$
• zufällige Anordnung	$O(N^2)$	$O(N^2)$

```
var i,j : index;
      t : item;

BEGIN
FOR i := 2 TO n DO
  BEGIN
  t := a[i];
  a[0].key := t.key;
  j := i-1;
  WHILE t.key < a[j].key DO
    BEGIN
    a[j+1] := a[j];
    j := j-1;
    END;
  a[j+1] := t;
  END;
END
```

-> Folie: Berechnung des Gesamtaufwandes - 14

1.7. Direktes Auswählen

Das Sortieren durch Auswählen wählt das jeweils kleinste Element der Menge aus und hängt es an das Ende der vorher schon sortierten Folge. Kurz zusammengefasst ergeben sich folgende Schritte:

1. Das Array enthält nach jedem Schritt einen linken sortierten und einen rechten unsortierten Teil
2. Am Anfang enthält der sortierte Teil keine Elemente
3. Bei jedem Schritt wird das kleinste Element des unsortierten Teils gesucht und mit dem linkesten Element des unsortierten Teils vertauscht => der sortierte Teil wird um eins größer.

-> Folie: Beispiel-Sortierung - 15

Algorithmusverhalten:

• Speicherbedarf	In-situ-Verfahren
• Stabilität	gewährleistet
• Vorsortierung	ohne Belang
• Datenstruktur	Array, Liste, Menge oder andere

Laufzeitverhalten:

	Vergleiche	Elementbewegung
• Datenanordnung		
• sortiert	$O(N^2)$	O(N)
• invers sortiert	$O(N^2)$	O(N)
• zufällige Anordnung	$O(N^2)$	O(N)

```
FOR i := 1 TO n-1 DO
  BEGIN
  { bestimme die Position des kleinsten Elementes }
  min := i;
  FOR j := i +1 TO n DO
    IF a[j].key < a[min].key THEN min := j;
  { vertausche Elemente a[i] und a[min] }
  t := a[min]; a[min] := a[i]; a[i] := t;
  END;
```

1.8. Bubble-Sort

Lokales Vertauschen. Kurz zusammengefasst ergeben sich folgende Schritte:

1. Die letzten beiden Elemente werden verglichen - falls nötig ausgetauscht
2. das drittletzte und das vorletzte Element werden verglichen - falls nötig ausgetauscht, u.s.w. bis zum ersten und zweiten Element
3. Wiederholung von 1. und 2., jedoch nur bis zum 2. u. 3. Element
4. wie 3., jedoch bis zum 3. u. 4. Element, etc. bis zu den beiden letzten Elementen

-> Folie: Beispiel-Sortierung - 16

quadratischer Aufwand wäre gegeben, wenn der Alg. immer komplett durchlaufen würde

```
var i,j : index;
      x : item;

FOR i := 2 TO n DO
  FOR j := n DOWNTO i DO
    IF a[j - 1].key > a[j].key THEN
      BEGIN
      x := a[j - 1];
      a[j - 1] := a[j];
      a[j] := x;
      END
END;
```

1.8.1. Shaker-Sort

Verbesserung von Bubble-Sort. Kurz zusammengefasst ergeben sich folgende Verbesserungen:

1. Bei jedem Durchlauf abprüfen, ab überhaupt noch eine Vertauschung notwendig war (merken der Position der letzten Vertauschung)
   => die äußere Schleife ist ggf. (abhängig von der zu sortierenden Liste) früher fertig
2. Richtung abwechseln

-> Folie: Beispiel-Sortierung - 17

Algorithmuseigenschaften:

- Speicherbedarf — In-situ-Verfahren
- Stabilität — gewährleistet
- Vorsortierung — relativ optimal bei Vorsortierung
- Datenstruktur — Array, Liste oder Menge und andere

Laufzeitverhalten:

• Datenanordnung	Vergleiche	Elementbewegung
• sortiert	O(N)	0
• invers sortiert	$O(N^2)$	$O(N^2)$
• zufällige Anordnung	$O(N^2)$	$O(N^2)$

```
var j,k,l,r : index;
          x : item;

BEGIN
l := 2; r := n; k := n;
REPEAT
   FOR j := r DOWNTO l DO
      IF a[j - 1].key > a[j].key THEN
         BEGIN
         x := a[j - 1];
         a[j - 1] := a[j];
         a[j] := x;
         k := j;
         END
   l := k + 1;
   FOR j := l TO r DO
      IF a[j + 1].key > a[j].key THEN
         BEGIN
         x := a[j - 1];
         a[j - 1] := a[j];
         a[j] := x;
         k := j;
         END
   r := k - 1;
UNTIL l > r;
END;
```

1.9. *Einfügen über große Entfernungen (Shell-Sort)*

Verfeinerung der Methode des direkten Einfügens. Kurz zusammengefasst ergeben sich folgende Schritte:

1. Aufspalten in 3 Teilarrays (die ersten beiden Teilarrays müssen eine Potenz von 2 sein!)
2. die ersten beiden Teilarrays elementweise vergleichen: die kleineren Elemente in 1. Teilarray
3. das 2. und 3. Teilarray elementweise vergleichen. Falls eine Vertauschung notwendig ist, dann auch mit den entsprechenden Elementen des 1. Teilarrays vergleichen und ggf. tauschen.
4. jedes der 3 Teilarrays wieder aufspalten in 2 weitere Arrays
5. wie bei 2. und 3. verfahren (jetzt aber mit insgesamt 6 Teilarrays; alle wieder paarweise vergleichen und bei Vertauschung vorhergehende Teilarrays miteinbeziehen)
6. solange in weitere Teilarrays aufteilen, bis sich eine durchgehende Liste ergibt (weiterhin alle Teilarrays paarweise vergleichen und bei Vertauschung vorhergehende Teilarrays miteinbeziehen)

-> Folie: Beispiel-Sortierung - 19

Algorithmuseigenschaften:

- Speicherbedarf In-situ-Verfahren
- Stabilität nicht gewährleistet
- Vorsortierung ohne Belang
- Datenstruktur Array und Strukturen mit berechenbaren Indices
- Anwendung hauptsächlich von akademischem Interesse

Laufzeitverhalten:

	Vergleiche	Elementbewegung
• Datenanordnung		
• zufällige Anordnung	$O(N * \log^2(N))$	$O(N * \log^2(N))$

```
var q,i,j : index;
        x : item;

BEGIN
q := 2;
WHILE q < n DO q := q * 2;
q := q DIV 2;
WHILE q > 1 DO
  BEGIN
  q := q DIV 2;
  FOR i := q + 1 TO n DO
       BEGIN
       x := a[i];
       j := i - q;
       WHILE (j > 0) AND (x.key < a[j].key) DO
          BEGIN
          a[j + q] := a[j];
          j := j - q;
          END;
       a[j + q] := x;
       END;
  END;
END;
```

1.10. Quick-Sort

Vertauschen über große Entfernungen. Kurz zusammengefasst ergeben sich folgende Schritte:

1. Als Vergleichswert wird der Mittelwert aus erstem und letztem Element errechnet
2. Von oben elementweise mit Vergleichswert vergleichen, stehen bleiben, falls das Element größer oder gleich dem Vergleichswert ist
3. Von unten ebenfalls elementweise mit Vergleichswert vergleichen, stehen bleiben, falls das Element kleiner oder gleich dem Vergleichswert ist
4. die beiden Elemente der Teilarrays vertauschen, dann weiter mit 2. und 3. bis sich beide Vergleichsketten treffen => oberes Array beinhaltet nur noch Werte kleiner als der Vergleichswert (unteres größer oder gleich dem Vergleichswert)
5. die Teilarrays wieder aufspalten, verfahren wie 1., 2., 3., 4. bis ein sortiertes Array übrig bleibt.

-> Folie: Beispiel-Sortierung - 20

Algorithmuseigenschaften:

- Speicherbedarf Ein In-situ-Verfahren ist programmierbar, aber bei rekursiver Implementierung ist das Verfahren schneller.
- Stabilität nicht gewährleistet
- Vorsortierung Sortierung wird nicht berücksichtigt.
- Datenstruktur Array und bedingt auch Listen.
- Anwendung Schnelles Verfahren für die interne Sortierung, wenn es bei zufällig angeordneten Daten nicht sogar das schnellste Verfahren ist

Laufzeitverhalten:

Datenanordnung	Vergleiche	Elementbeweg.	Stack-Bedarf
worst case	$O(N^2)$	$O(N^2)$	O(N)
zufällige Anordnung	O(N * log(N))	O(N * log(N))	O(log(N))

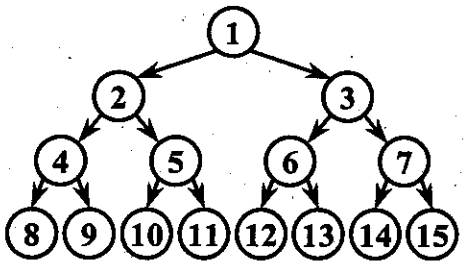
```
PROCEDURE sort (o, u : integer);
VAR i, j : index;
    x, w : item;
BEGIN
i := o; j := u
x.key := (a[o].key + a[u].key) div 2
REPEAT
  WHILE a[i].key < x.key DO i := i + 1;
  WHILE x.key < a[j].key DO j := j - 1;
  IF i <= j THEN
    BEGIN
    w := a[i];
    a[i] := a[j];
    a[j] := w;
    END
UNTIL i > j;
IF o < j THEN sort(o, j);
IF i < u THEN sort(i, u);
END;

BEGIN
sort (1, n);
END;
```

Bemerkungen:

- Im schlechtesten Fall wird jedes Mal ein 1-elementiges Array als Teilarray abgespalten (wenn der Mittelwert entweder zu klein oder zu groß ist)
  => Anzahl der Durchläufe: n - 1
- Wenn der Aufwand für den k-ten Durchlauf: O(n - k + 1) ist:
  => Gesamtaufwand: $O((n - 1)(n - k + 1)) = O(n^2 - ...) = O(n^2)$
  => obwohl dieser Fall recht unwahrscheinlich ist, ist Quicksort nicht zu empfehlen, wenn in festgelegter Zeit sortiert werden muss.
- Mittelwert kann ggf. besser getroffen werden, wenn mehrere Werte dazu herangezogen werden.

1.11. Direktes Auswählen und Heapstruktur (Heap-Sort)

Der Heap ist hier ein vollständig binärer Baum, in dem für jeden Knoten gilt: Der Vorfahr ist größer als seine beiden Nachfahren. Die Indices des Array (im Beispiel für 15 Elemente) sind im Baum wie folgt abgebildet:



Kurz zusammengefasst ergeben sich folgende Schritte:

1. Heap mit der o.g. Bedingung aufbauen. Anfangen mit dem größten Index des Arrays (im Bsp. Knoten 15), dabei:
   - die beiden Sohnknoten miteinander vergleichen, den größeren der beiden mit dem Vaterknoten vergleichen. Ist der Vaterknoten kleiner, dann vertauschen
   - wenn vertauscht wurde, dann bis zu den Blättern zurückverfolgen
   - dann nächsten Vaterknoten (im Bsp. zuerst Knoten 7, dann 6,5,4,3,2,1)
2. erstes und letztes Element (Reihenfolge ergibt sich durch Array-Indices) vertauschen, da nach dem Heap-Aufbau immer das größte Element in der Wurzel steht (das Array verkleinert sich successive um 1).
3. neuen Heap mit (n-1)-Elementen aufbauen, dabei mit Knoten 1 beginnen und wie in Schritt 1:
   - die beiden Sohnknoten miteinander vergleichen, den größeren der beiden mit dem Vaterknoten vergleichen. Ist der Vaterknoten kleiner, dann vertauschen
   - wenn vertauscht wurde, dann bis zu den Blättern zurückverfolgen
4. weiter mit 2. und 3. bis n=1 => fertig sortiertes Array

-> Folie: Beispiel-Sortierung - 22

Algorithmuseigenschaften:

- Speicherbedarf — In-situ-Verfahren
- Stabilität — nicht gewährleistet
- Vorsortierung — ohne Belang
- Datenstruktur — Array und Strukturen mit berechenbaren Indices
- Verbesserung — „Smoothsort“ von Dijkstra kann eine Vorsortierung ausnutzen
- Anwendung — einziges worst-case-optimales In-situ-Verfahren
- Anmerkung — Kompliziert zu programmieren, wenn der Anfangsindex des Array ungleich eins ist.

Laufzeitverhalten:

Datenanordnung	Vergleiche	Elementbewegung
zufällige Anordnung	O(N * log(N))	O(N * log(N))

```
PROCEDURE shift (i, m : integer);
BEGIN
WHILE i <= m DIV 2 DO
  BEGIN
  j := 2 * i;
  IF (j < m) AND (a[j].key < a[j+1].key) THEN j := j+1;
  IF a[i].key < a[j].key THEN
    BEGIN
    temp := a[i];
    a[i] := a[j];
    a[j] := temp;
    i := j;
    END
  ELSE i := m;
  END;
END;

BEGIN
FOR i := (n DIV 2) DOWNTO 1 DO shift (i, n)
FOR i := n DOWNTO 2 DO
  BEGIN
  t := a[i];
  a[i] := a[1];
  a[1] := t;
  shift (1, i-1);
  END;
END;
```

1.12. Merge-Sort

1.12.1. Files und Listen

Die Grundidee aller Mergesort-Verfahren besteht in dem Zusammenfügen zweier sortierter Teilfolgen. Da Daten häufig intern als Liste (verkettete Listen von Datenblöcken bei Dateisystemen, etc.) oder extern als Files vorliegen, ist es ein gut geeignetes Verfahren. Alle Mergesort-Varianten sind, sofern sie brauchbar programmiert wurden, worst-case-optimal. Die Verfahren berücksichtigen sogar eine Vorsortierung der Daten. Aufgrund des sequentiellen Zusammenfügens zweier sortierter Teilfolgen ist immer eine stabile Sortierung gewährleistet. Damit zählen einzig die Mergesort-Verfahren zu den worst-case-optimalen, stabilen Sortierverfahren.

1.12.2. Schlüsselvergleich

Der Aufwand und damit die Zeit für einen Schlüsselvergleich steigt proportional mit der Größe des Schlüssels. Ein Filename unter MSDOS besteht aus acht Buchstaben für den Namen und drei Buchstaben für die Extension. In diesem Beispiel bietet es sich an, den Schlüssel in zwei oder mehr Teilschlüssel aufzuteilen. Dieses Prinzip der Schlüsselteilung behandeln die meisten Informatikbücher unter dem Schlagwort „Radixsort". Dabei kommt den Teilschlüsseln immer eine unterschiedliche Priorität zu. Im Beispiel der Filenamen könnte man fordern, daß diese in erster Linie nach den Extensionen und in zweiter Linie nach den Namen alphabetisch sortiert werden sollen. Damit werden alle Filenamen mit der Extension COM vor die Filenamen mit der Extension EXE sortiert. Erst innerhalb der Klasse von Filenamen gleicher Extension wären die Filenamen alphabetisch sortiert.

Verfahren der Teilschlüsselbildung:

- Einerseits können die Elemente erst nach dem Teilschlüssel mit der niedrigeren Priorität sortiert werden, um sie dann in einem zweiten Durchgang nach dem übergeordneten Teilschlüssel zu sortieren.
  - Diese Art der Sortierung wird in Anlehnung an die alten Lochkarten-Sortiermaschinen als Sortieren durch Fachverteilung bezeichnet und setzt voraus, daß man eine stabile Sortierung gewährleisten kann.
  - Im Beispiel der Filenamen sortiert man also erst nach den Namen und im zweiten Durchgang nach den Extensions.
- Andererseits können die Elemente erst nach dem Teilschlüssel mit der höheren Priorität und im zweiten Sortierdurchgang nach dem Teilschlüssel mit der niedrigeren Priorität sortiert werden.
  - Dieses Verfahren bezeichnet man als Radix-Exchange-Sort. Es ist Quicksort recht ähnlich. Nach diesem Verfahren wird also erst nach den Extensions und dann nach den Filenamen sortiert.

An dieser Stelle muß jedoch auch auf die Nachteile einer Radix-Sortierung hingewiesen werden.:

- Wenn im Extremfall ein Teilschlüssel nur einen Wert annimmt, so macht das Verfahren schon einen überflüssigen Sortierdurchgang.
  - Wenn ein Schlüssel aus vielen kleinen Teilschlüsseln besteht, kann dies also schnell zu langen Sortierzeiten führen.

Eine Radix-Sortierung macht nur dann Sinn, wenn die Werte der Schlüssel und Teilschlüssel ein breites Spektrum aufweisen. Wenn jedoch die Teilschlüssel nur wenige Werte annehmen, so ist meist eine Radix-Sortierung nur noch mit einem Sortierverfahren günstig, das eine Vorsortierung ausnutzt.

1.12.3. Algorithmus

Sequentielles Vertauschen. Vergleiche die Elemente zweier sortierter Folgen miteinander und füge sie zu einer sortierten Folge zusammen.

Algorithmuseigenschaften:

- Speicherbedarf Als In-situ-Verfahren programmierbar, aber das rekursive Mergesort für Listen ist schneller.
- Stabilität gewährleistet
- Vorsortierung Nutzung der normalen und inversen Vorsortierung
- Datenstruktur Bevorzugt Listen, aber auch Arrays möglich.
- Anwendung Bevorzugt bei Daten in Listen und auf ext. Datenträgern

Laufzeitverhalten:

Datenanordnung	Vergleiche	Elementbeweg.	Stack-Bedarf
sortiert	O(N * log(N))	O(N * log(N))	O(log(N))
invers sortiert	O(N * log(N))	O(N * log(N))	O(log(N))
zufälligeAnordnung	O(N * log(N))	O(N * log(N))	O(log(N))

```
PROCEDURE mergesort (links, rechts : INTEGER);

PROCEDURE merge (l, m, r : INTEGER);
BEGIN
i := l; j := m + l; k := l;
WHILE (i <= m) AND (j <= R) DO
   BEGIN
   IF a[i].key <= a[j].key THEN
      BEGIN
      b[k] := a[i]; i := i +1;
      END
   ELSE
      BEGIN
      b[k] := a[j]; j := j +1;
      END;
   k := k +1;
   END;
IF i > m THEN
   BEGIN { nimm noch die zweite Teilfolge }
   FOR h := j TO r DO b[ k + h - j] := a[h];
   END
ELSE
   BEGIN
   FOR h := i TO m DO b[ k + h - i] := a[h];
   END;
FOR h := l TO r DO a[h] := b[h];
END;

BEGIN
IF links < rechts THEN
   BEGIN { sonst max. ein Element }
   help := (links + rechts) DIV 2;
   mergesort ( links, help);
   mergesort ( help + 1, rechts);
   merge ( links, help, rechts);
   END;
END;
```

1.13. Einordnung der versch. Algorithmen

1.13.1. Sortieren in Arrays

Als herausragende Eigenschaft bieten Arrays über die Indices einen direkten Zugriff auf jedes Element. Das Shellsort-Verfahren nutzt diese Eigenschaft aus, um im Array sortierte Folgen zwischen weit auseinanderliegenden Elementen herzustellen. Diesem Verfahren liegt als Prinzip das Sortieren durch Einfügen zugrunde. Das elementare Verfahren des Sortierens durch Auswählen realisiert das Heapsort. Durch geschickte Wahl der Haufenstruktur ist das kleinste Element in einem Array bestimmbar.

Heapsort ist das einzige In-situ-Verfahren mit einer worst-case-optimalen Laufzeitordnung. Durch eine geschickte Programmierung läßt sich sogar eine Vorsortierung ausnutzen.

Der Vorteil von Quicksort gegenüber allen anderen bekannten Sortierverfahren ist die enorme Geschwindigkeit. Im Falle zufällig angeordneter Daten hat Quicksort eine Laufzeitordnung für Vergleich und Elementbewegungen von O(N * log(N)). Das Problem bei Quicksort liegt aber in dem nie ausschließbaren worst-case. In diesem Fall degeneriert das Verfahren zu einer Laufzeitordnung von $O(N^2)$.

1.13.2. Fazit

Die Laufzeitordnung beträgt für die drei Verfahren Quicksort, Heapsort und Mergesort O(N * log(N)), wobei Quicksort die wahrscheinlich schnellste Variante ist. Die Laufzeit hängt im wesentlichen von der Zeit ab, die für die Elementbewegungen und die Schlüsselvergleiche nötig ist. Da interner Speicher inzwischen relativ preiswert ist, fällt der Speicherbedarf für den Algorithmus selbst kaum ins Gewicht.

Je mehr Bytes ein Element umfaßt, desto länger dauert die Bewegung eines Elementes im internen Speicher. Eine einfache Methode, diesen Zeitfaktor zu minimieren, besteht in der Idee, nicht die Elemente selbst zu bewegen, sondern nur einen Zeiger auf das Element zu verändern. Dies ist eine komplizierte Umschreibung für eine Liste und liefert einen Grund, bei Elementen mit großem Speicherbedarf zusätzlich einen Pointer vorzusehen.

Die Wahl eines Sortieralgorithmus hängt von vielen Randbedingungen ab. Für eine Entscheidung sollte man sich folgende Fragen beantworten:

- Herrscht eine Vorsortierung vor?
- Muss die Sortierung in einer vorgegebenen Zeit beendet sein?
- Ist eine Listenstruktur verwendbar?
- Soll intern, extern oder parallel sortiert werden?
- Wieviel Platz darf der Algorithmus beanspruchen?
- Wieviel Elemente sind zu sortieren?
- Benötigt man eine stabile Sortierung?
- Kann ein Schlüsselvergleich in mehrere Schlüsselvergleiche aufgeteilt werden?
- Eignet sich die Verteilung der Teilschlüssel für eine Radix-Sortierung?

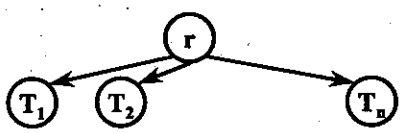
2. Baumstrukturen

Zunächst gilt es zu klären, warum man Elemente, die sowieso alle im Hauptspeicher zu halten sind, nicht einfach in einem Array verwalten soll. Nun, in einem sortierten Array lassen sie sich sicherlich schnell auffinden, doch Einfügen und Löschen ziehen meist sehr umfangreiche Kopiervorgänge nach sich. Zudem stellt die statische Definition der Feldgröße oftmals eine unzumutbare Restriktion dar.

2.1. Definition

Ein Baum T ist eine endliche Menge N von Knoten, die entweder

- leer ist (N={}; In diesem Fall spricht man von einem leeren Baum) oder
- aus einem einzigen Knoten r besteht (N={r}; r wird Wurzel genannt) oder
- aus mehreren Knoten besteht (N = {r, T_1, T_2, ..., T_n}, n bezeichnet den Grad des Baumes).



Ein Binärbaum T (ein Baum vom Grad 2) ist eine endliche Menge N von Knoten, die entweder

- leer ist (N={}; In diesem Fall nennt man den Baum einen leeren Baum) oder
- das Tripel T (TL,r,TR), wobei r ein spezieller Knoten ist, der Wurzel heißt. Hinter TL und TR verbergen sich der linke und der rechte Unterbaum von r (es handelt sich dabei um weitere Binärbäume, die nach dem gleichen Bildungsprinzip aufgebaut sind).

Das Niveau eines Knotens ist gleich der Länge des Weges (d.h. der Anzahl der Knoten) von der Wurzel. Die Wurzel besitzt stets das Niveau 1. Das größte auftretende Niveau nennt man die Höhe des Baumes.

Knoten eines Baumes, die keine auslaufenden Kanten besitzen, heißen Blätter. Die von einem Knoten k erreichbaren Knoten bilden den Teilbaum mit der Wurzel k. Wenn eine Kante vom Knoten u zum Knoten v existiert, dann heißt u Vater von v und v Sohn von u. Gibt es eine weitere Kante von u zu einem Knoten v' dann nennt man v und v' Brüder. Bei binären Bäumen unterscheidet man zwischen dem linken und dem rechten Sohn eines Knotens.