3.4 Mergesort

3.4.1 Mischen

Algorithmus: *Zweiweg-Mischen*

Es liegen zwei sortierte Folgen von Daten vor, die zu einer sortierten Folge kombiniert werden müssen.
Man verwendet dazu ein Feld, das so lang ist, wie die beiden zu mischenden Folgen zusammen und sortiert durch sequentielles Durchlaufen beider Ursprungsfolgen.

Programm:

```
i = 1; j = 1;
a[M+1] = INT_MAX; b[N+1] = INT_MAX;
for (k = 1; k<= M+N; k++)
  c[k] = ( a[i] < b[j] ) ? a[i++] : b[j++];
```

Dabei wird O(M+N) zusätzlicher Speicherplatz benötigt.

3.4.2 Zwei-Weg-Mergesort

Algorithmus:

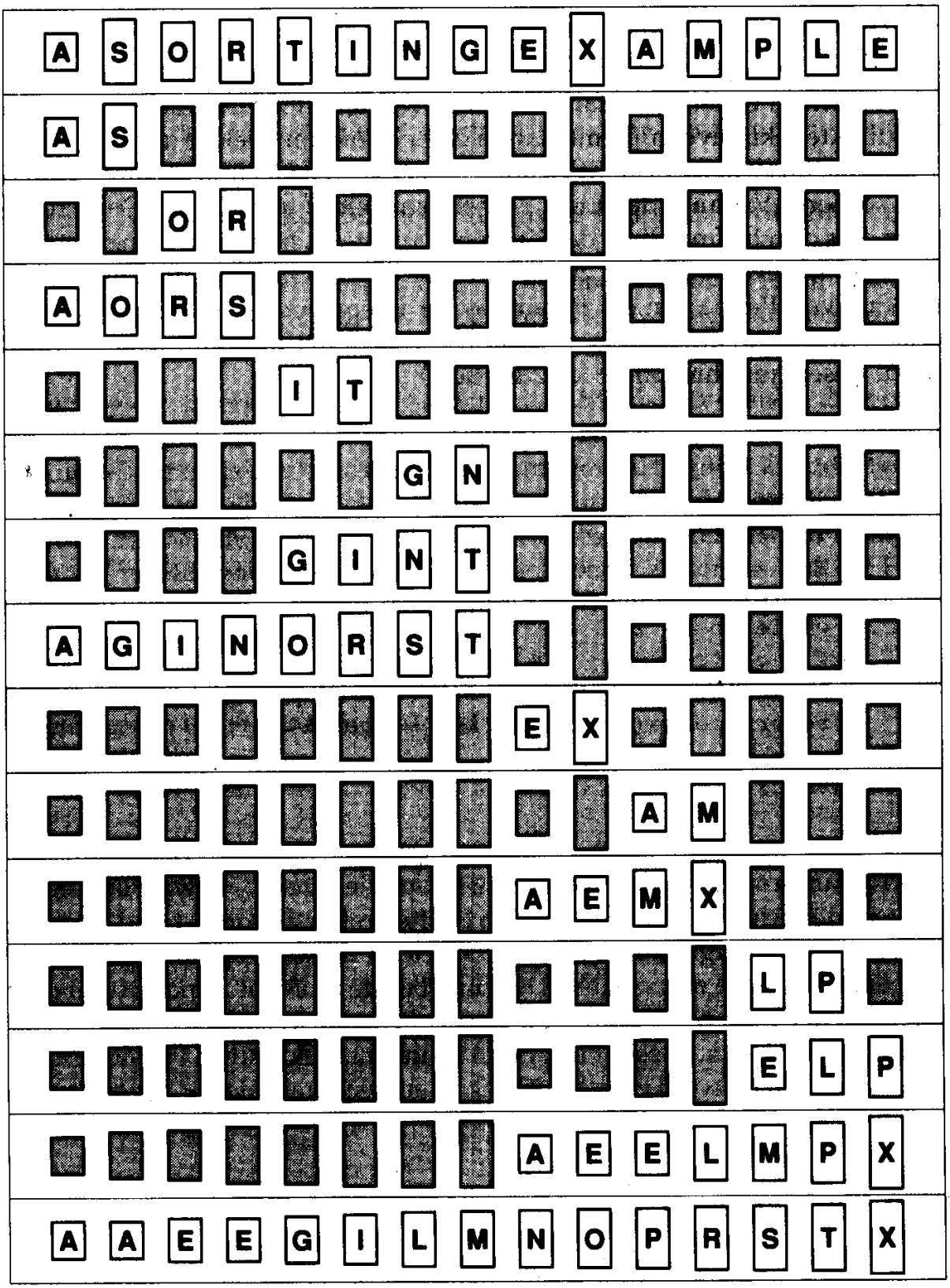
Man teile eine gegebene Elementfolge in zwei Hälften, sortiere diese beiden Hälften (rekursiv) und mische sie dann.
Mergesort folgt also wiederum dem Prinzip **divide and conquer**.

Programm:

```
/* a[l], ..., a[r] wird mit Hilfe von b[l], ..., b[r]
sortiert */

mergesort (int a[], int l, int r)
  {
    int i, j, k, m;
    if (r > l)
      {
        m = (r+l)/2;
        mergesort (a, l, m);
        mergesort (a, m+1, r);
  /* a[l]...a[m] und a[m+1]...a[r] sind sortiert */
        for (i= m+1; i> l; i--) b[i-1] = a[i-1];
        for (j=m; j<r; j++) b[r+m-j] = a[j+1];
        for (k=l; k<= r; k++)
          a[k] = (b[i]<b[j])  ? b[i++] : b[j--];
      }
  }
```

Mergesort, Beispiel



Mergesort, Komplexität

Anzahl Vergleiche:

Beim Mischen der Folge der Länge N werden N Vergleiche durchgeführt.
Nach $\lceil \log N \rceil$ rekursiven Aufrufen ist die Folge sortiert.
Damit gilt:

$$C(N) = \begin{cases} 2 * C(N/2) + N & N > 1 \\ 0 & N = 1 \end{cases}$$

$\Rightarrow C(N) = O(N \log N)$
wobei $C_{min}(N) = C_{max}(N) = C_{avg}(N) = O(N \log N)$

Anzahl Bewegungen:

Es werden mehr Bewegungen ausgeführt, als Vergleiche.
Pro Vergleich wird eine Bewegung ausgeführt.
Zusätzlich werden die Elemente von a nach b kopiert.
Es werden also 2N Bewegungen beim Mischen durchgeführt.
Damit gilt:

$$C(N) = \begin{cases} 2 * M(N/2) + 2N & N > 1 \\ 0 & N = 1 \end{cases}$$

$\Rightarrow M(N) = O(N \log N)$
wobei wiederum $M_{min}(N) = M_{max}(N) = M_{avg}(N) = O(N \log N)$

Mergesort, Kommentare

- Mergesort sortiert eine aus N Elementen bestehende Folge immer in O(N log N), also selbst im ungünstigsten Fall.

- Normalerweise benötigt Mergesort O(N) zusätzlichen Speicherplatz zum Sortieren; es gibt jedoch Implementierungen von Mergesort mit O(1) zusätzlichem Speicherplatzbedarf.

- Mergesort kann so implementiert werden, daß es den Zugriff auf die Daten hauptsächlich sequentiell realisiert, d.h. es ist gut geeignet zum Sortieren von auf Bändern gespeicherten Daten oder von Daten in verketteten Listen.

- Mergesort ist *stabil*, d.h. es behält die relative Reihenfolge gleicher Schlüssel in der Datei bei.

3.4.3 Iterativer Zwei-Wege-Mergesort

Algorithmus

Durchlaufe Elementfolge und sortiere Paare, so daß Teillisten der Größe 2 entstehen.
Durchlaufe Elementfolge und sortiere je 2 Paare, so daß Teillisten der Größe 4 entstehen.

usw.

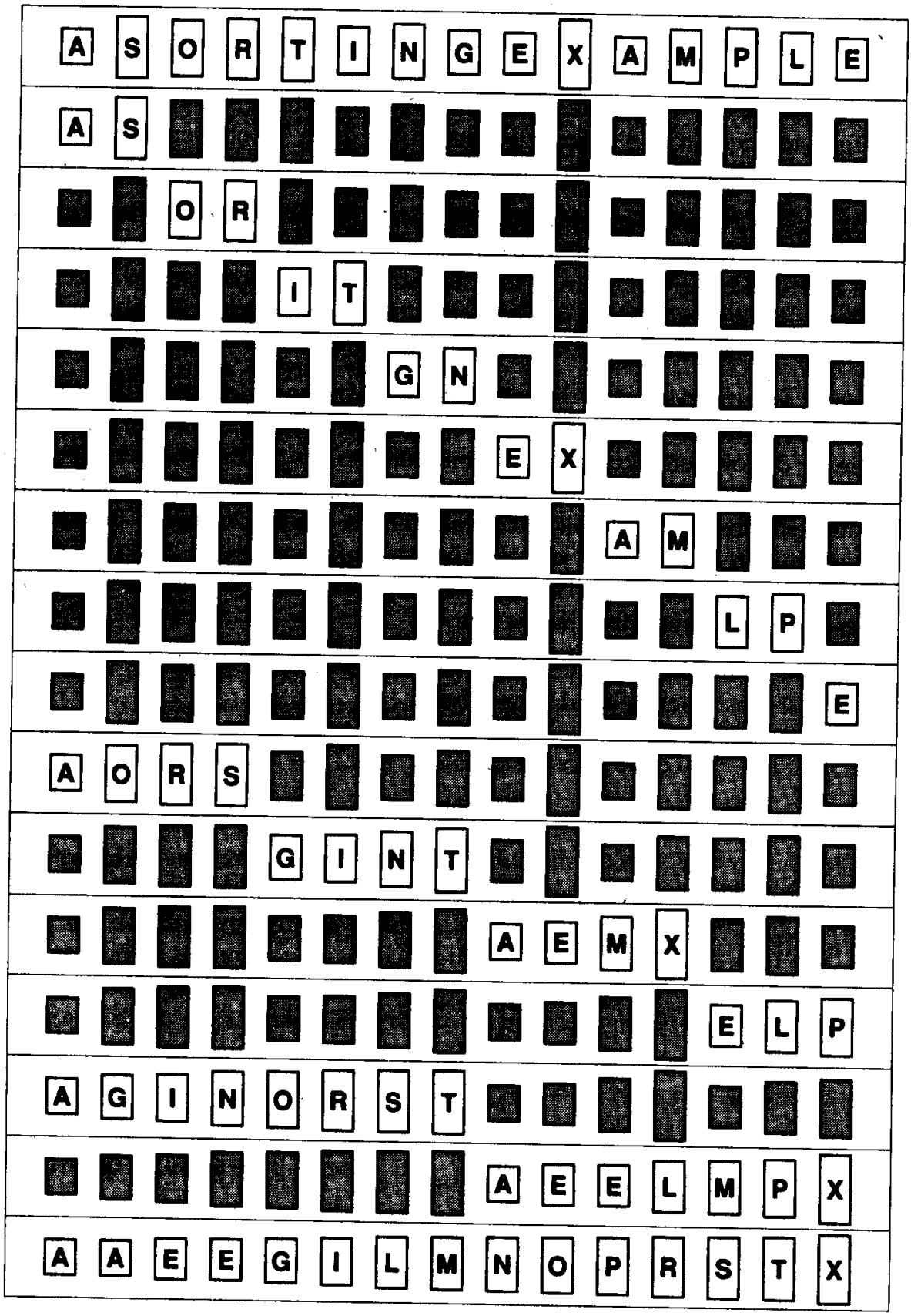
Durchlaufe Elementfolge und sortiere beide Teillisten zu einer sortierten Folge.

Iterativer Zwei-Wege-Mergesort

Programm

```
struct node *mergesort (struct node *c)
  {
    int i, N;
    struct node *a, *b, *head, *todo, *t;
    head = (struct node *) malloc(sizeof *head);
    head->next = c; a=z;
    for (N=1; a != head->next; N=N+N)
      {
        todo = head->next; c=head;
        while (todo != z)
          {
            t=todo; a=t;
            for (i=1; i<N; i++) t=t->next;
            b=t->next; t->next=z; t=b;
            for (i=1; i<N; i++) t=t->next;
            todo = t->next; t->next = z;
            c->next = merge(a, b);
            for (i=1; i<=N+N; i++) c= c->next;
          }
      }
    return head->next;
  }
```

Iterativer Mergesort, Beispiel



Iterativer Mergesort, Kommentare

- Bottom-up-Methode

- im wesentlichen selbe Mischoperationen wie rekursive Form, jedoch in anderer Reihenfolge

- Implementierung mit verketteten Listen sinnvoll, da Umsetzen eines Zeigers schneller geht, als Umkopieren von Daten

- Selbe Komplexität wie rekursiver Mergesort, da anstatt von log N rekursiven Aufrufen log N Durchläufe der Liste durchgeführt werden.

  $C(N) = O(\log N)$

  $M(N) = O(\log N)$

3.4.4 Natürlicher Zwei-Wege-Mergesort

Algorithmus

Nutze die natürliche Ordnung der Daten aus.
Verschmelze also die ersten beiden sortierten Sequenzen (gleichgültig, wie lang diese sind), dann die nächsten beiden, usw., bis die Datei sortiert ist.

Beispiel

F = 2, | 1, 3, 9, | 5, 6, 7, | 4, 8, 10
1, 2, 3, 9, | 4, 5, 6, 7, 8, 10
1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Natürlicher Mergesort, Analyse

Anzahl Vergleiche

Zur Ermittlung der sortierten Sequenzen werden in jedem Durchlauf linear viele zusätzliche Vergleiche benötigt. Bei $\lceil \log N \rceil$ Durchläufen also zusätzlich O(N log N) Vergleiche.
Damit ist

$C_{max}(N) = O(N \log N)$

Im besten Fall ist die gegebene Folge bereits sortiert, so daß nur 1 Durchlauf nötig ist:

$C_{min} = O(N)$

Im Mittel benötigen wir $\frac{\log N}{2}$ Durchläufe, so daß gilt:

$C_{avg}(N) = O(N \log N)$

Anzahl Bewegungen:

$M_{min}(N) = 0$
$M_{max}(N) = M_{avg}(N) = O(N \log N)$

entsprechend

Natürlicher Mergesort, Kommentare

- Kosten zur Identifizierung der bereits sortierten Sequenzen kommen dazu, so daß im Durchschnitt keine Verbesserungen durch diesen Algorithmus erzielbar ist

- ist für entartete Fälle gut (z.B. bereits sortierte Datei)

3.5 Radix - Sort

Idee:

Die Schlüssel werden als Zahlen (Zifferfolgen zu einer bestimmten Basis) betrachtet. Beim Sortieren werden die einzelnen Ziffern nacheinander behandelt: Zuerst wird nach der ersten Ziffer sortiert, dann nach der zweiten usw. Deshalb bezeichnet man das Verfahren auch als digitales Sortieren.

Varianten

Welche Basis wird betrachtet? (Im Rechner typischerweise 2^i)?
In welcher Reihenfolge werden die Ziffern bearbeitet?

Einfachste Möglichkeit: Radix = 2 => Binärzahlen
Sortieren nach Bits

Reihenfolge:

- von links nach rechts: `Radix Exchange Sort'
- von rechts nach links: `Straight Radix Sort'

3.5.1 Algorithmus Radix Exchange Sort

- Beginne mit dem höchstwertigen Bit
- Vertausche Elemente im Datensatz so, daß im linken Teil der Folge alle Bits den Wert = 0 haben, im rechten Teil den Wert 1
- Wende denselben Algorithmus auf beide Teile rekursiv an mit dem nächstniedrigen Bit, usw.

```
unsigned bits (unsigned x, int k, int j)
{ return (x >> k) &  ( 0 << j),  }
```

Programm:

Vorbemerkung: Die Sprache C hat einen mächtigen Satz von Bitoperatoren, so daß Radix Sort in C leicht zu programmieren ist. Wir schreiben eine Prozedur, die uns die rechten k Bits aus einer Ganzzahl ausblendet:
radixexchange (int a[], int l, int r, int b)

```
  {
    int t, i, j;
    if (r>l && b>=0)
      {
        i = l; j = r;
        while (j != i)
          {
while (bits (a[i], b, l)==0 && i<j) i++; while (bits
(a[j], b, 1)!=0 && j>i) j--; t = a[i]; a[i] = a[j]; a[j]
= t;
          }
```

```
if (bits (a[r], b, l) == 0) j++; radixexchange (a, l, j-
l, b-l); radixexchange (a, j, r, b-1);
      }
  }
```

Radix Exchange Sort, Beispiel

Anmerkung: Wir verwenden als Schlüssel wieder Buchstaben, die wir als im Alphabet durchnumeriert ansehen:

Buchstabe	A	B	C	D	E	...
Bitmuster	00001	00010	00011	00100	00101	...

Radix Exchange Sort, Beispiel

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	E	O	L	M	I	N	G	E	A	X	T	P	R	S
A	E	A	E	G	I	N	M	L	O					
A	A	E	E	G										
A	A													
A	A													
		E	E	G										
		E	E											
					I	N	M	L	O					
						L	M	N	O					
						L	M							
								N	O					
										S	T	P	R	X
										S	R	P	T	
										P	R	S		
											R	S		

Straight Radix Sort, Beispiel

A 00001	**R** 10010	**T** 10100	**X** 11000	**P** 10000	**A** 00001
S 10011	**T** 10100	**X** 11000	**P** 10000	**A** 00001	**A** 00001
O 01111	**N** 01110	**P** 10000	**A** 00001	**A** 00001	**E** 00101
R 10010	**X** 11000	**L** 01100	**I** 01001	**R** 10010	**E** 00101
T 10100	**P** 10000	**A** 00001	**A** 00001	**S** 10011	**G** 00111
I 01001	**L** 01100	**I** 01001	**R** 10010	**T** 10100	**I** 01001
N 01110	**A** 00001	**E** 00101	**S** 10011	**E** 00101	**L** 01100
G 00111	**S** 10011	**A** 00001	**T** 10100	**E** 00101	**M** 01101
E 00101	**O** 01111	**M** 01101	**L** 01100	**G** 00111	**N** 01110
X 11000	**I** 01001	**E** 00101	**E** 00101	**X** 11000	**O** 01111
A 00001	**G** 00111	**R** 10010	**M** 01101	**I** 01001	**P** 10000
M 01101	**E** 00101	**N** 01110	**E** 00101	**L** 01100	**R** 10010
P 10000	**A** 00001	**S** 10011	**N** 01110	**M** 01101	**S** 10011
L 01100	**M** 01101	**O** 01111	**O** 01111	**N** 01110	**T** 10100
E 00101	**E** 00101	**G** 00111	**G** 00111	**O** 01111	**X** 11000

Radix Exchange Sort, Komplexität

$C(N) = c_1 * b * N$ für einen Schlüssel von b Bits

Deshalb:
$C_{max}(N) = C_{min} = C_{avg}(N) = O(N * b)$

Im günstigsten Fall sind die Schlüsselwerte gleichverteilt und decken den gesamten Wertebereich ab. Dann ist $b = \log N$, und es gilt
$C_{min}(N) = O(N \log N)$.

Im ungünstigsten Fall liegen viele, gleiche oder ähnliche Schlüssel vor ($b >> \log N$), dann ist Radix Exchange Sort schlecht.

Radix Exchange Sort, Kommentare

Sehr ähnlich zum Quicksort! Unterschied:
Das Teilerelement ist jeweils 2 b. Es muß nicht in der Ausgangsfolge vorkommen, so dass nicht sicher ist, dass bei einer Vertauschungsoperation ein Element auf seinen endgültigen Platz kommt.

- Nachdem ein Schlüssel anhand seiner "linken" Bits von allen anderen Schlüsseln unterschieden worden ist, werden seinen weiteren Bits nicht mehr betrachtet. Das spart Vergleiche. Bei einer zufälligen Verteilung der Schlüssel über den Wertebereich unterscheidet sich ein Schlüssel im Mittel nach log N Bits von den anderen, $N<<2^b$.
-
-

Beispiel: 1032 Elemente a` 32 Bit,
log N = 5, b = 32, im Mittel fünf Vergleiche pro Element.

Daher: Radix Exchange Sort günstig bei zufällig verteilten Schlüsseln, ungünstig bei vielen gleichen oder benachbarten Schlüsseln.

3.5.2 Straight Radix Sort

Algorithmus:

- Beginne mit niedrigstwertigen Bit.
- Entnehme dem Datensatz der Reihe nach zuerst alle Elemente mit 0, dann alle Elemente mit 1 an der betreffenden Bitposition.
- Wende Algorithmus rekursiv auf die beiden Teilfolgen und das nächsthöhere Bit an.
-
-

Achtung! Dies funktioniert nur, wenn der Ein-Bit-Zerlegungsprozess stabil ist.

Zerlegung in "Einser" und "Nuller":

Distribution Counting (ideal dafür geeignet)

Im einfachsten Fall: M = 2

Variante

In der Regel besser: M > 2, Tabelle von M Zählern.

Straight Radix Sort, Programm

```
straightradix (int a[], int b[], int N)
  {
    int i, j, pass, count[M-l];
    for [pass = 0; pass < (w/m)-l; pass++)
      {
        for (j = 0; j < M; j++) count [j] = 0;
        for (i = l; i <= N; i++)
             count[bits(a[i], pass*m, m)]++;
        for (j = l; j < M; j++)
             count[j] = count[j-l]+count[j];
        for (i = N; i>= l; i-)
             b[count[bits[a[i], pass*m, m)]-] = a[i];
        for (i = l; i<= N; i++) a[i] = b[i];
      }
  }
```

Straight Radix Sort, Beispiel

w = 1, ein Bit pro Durchgang, rechts beginnend

(Abb. 10.4, S. 170 Sedgewick)

A 00001	**R** 10010	**T** 10100	**X** 11000	**P** 10000	**A** 00001
S 10011	**T** 10100	**X** 11000	**P** 10000	**A** 00001	**A** 00001
O 01111	**N** 01110	**P** 10000	**A** 00001	**A** 00001	**E** 00101
R 10010	**X** 11000	**L** 01100	**I** 01001	**R** 10010	**E** 00101
T 10100	**P** 10000	**A** 00001	**A** 00001	**S** 10011	**G** 00111
I 01001	**L** 01100	**I** 01001	**R** 10010	**T** 10100	**I** 01001
N 01110	**A** 00001	**E** 00101	**S** 10011	**E** 00101	**L** 01100
G 00111	**S** 10011	**A** 00001	**T** 10100	**E** 00101	**M** 01101
E 00101	**O** 01111	**M** 01101	**L** 01100	**G** 00111	**N** 01110
X 11000	**I** 01001	**E** 00101	**E** 00101	**X** 11000	**O** 01111
A 00001	**G** 00111	**R** 10010	**M** 01101	**I** 01001	**P** 10000
M 01101	**E** 00101	**N** 01110	**E** 00101	**L** 01100	**R** 10010
P 10000	**A** 00001	**S** 10011	**N** 01110	**M** 01101	**S** 10011
L 01100	**M** 01101	**O** 01111	**O** 01111	**N** 01110	**T** 10100
E 00101	**E** 00101	**G** 00111	**G** 00111	**O** 01111	**X** 11000

Straight Radix Sort, Komplexität

Straight Radix Sort kann N Datensätze mit aus b Bits bestehenden Schlüsseln in b/m Durchläufen sortieren, wobei ein zusätzlicher Platz für 2^m Zähler (und ein Puffer für das Umordnen der Datei) benötigt wird.

Für konstantes b ist Straight Radix Sort O (N), also linear.

Empirisch: - Wähle m = b/4.
Schlüssel werden zur Basis M betrachtet. Bei 32-Bit Worten: m = 8, 4 Durchläufe a` 1 Byte, Sortierung der Einzelbytes mit Distribution Counting. Gut in der Praxis, da die Rechner über effiziente Byte-Operationen verfügen.

Straight Radix Sort, Kommentare

- Das Verfahren entspricht dem alten mechanischen Sortieren von Lochkartenstapeln: pro Durchgang von rechts nach links wird eine der 12 Zeilen verarbeitet, jeder sortierte Teilstapel wird im nächsten Durchgang erneut in 12 Teilstapel sortiert, die Sortierung ist stabil
- Sehr ähnlich dem Distribution Counting

3.6 Externes Sortieren

Sortierung unter Verwendung von externen Speichermedien. Vor allem relevant, wenn die zu sortierende Datenfolge nicht in den Haupspeicher paßt.

Warum braucht man andere Algorithmen?

1. Die Kosten des Zugriffs auf ein Element sind um Größenordnungen höher.
2. Die externene Speichermedien sind häufig nur sequentiell zugreifbar.

Wir beschränken uns im folgenden

- auf zwei Hierarchie- Ebenen
- auf Magnetband als externes Gerät

Sort - Merge - Verfahren

Unter der Annahme, daß das externe Medium nur sequentiell arbeitet, nehmen alle Algorithmen eine Sort-Merge-Struktur an:
Es wird zunächst der Hauptspeicher "vollgelesen", dann intern sortiert, dann die sortierten Teilfolgen sequentiell auf Hilfsspeicher- Dateien geschrieben, dann wird wieder eingelesen usw. Gerade das "Merging" ist für externe Verfahren besonders geeignet, weil sowohl die Eingabefolgen sequentiell gelesen, als auch die Ausgabefolgen sequentiell geschrieben werden.
Auf dem Ausgabemedium entstehen sortierte Sequenzen ("runs").

3.6.1 Natürliches Mischen

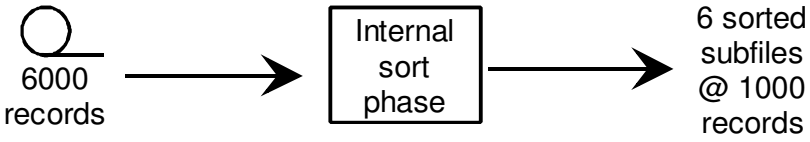
Eingabe: k Bänder

Sortierung: im Hauptspeicher der Größe M

Ausgabe: ein Band, alle Sequenzen nacheinander

Umkopieren auf k Bänder als Zwischenschritt

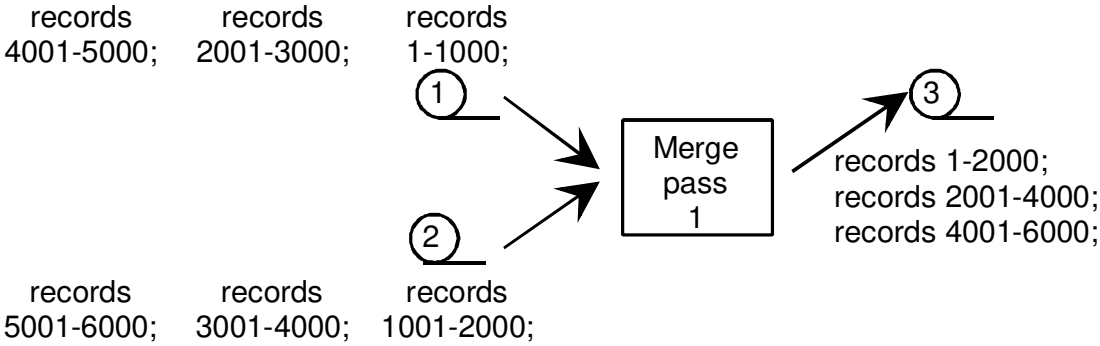
Beispiel 1: Natürliches Zweiwege- Mischen



2-way merge: Assume 2 input files and 1 output file

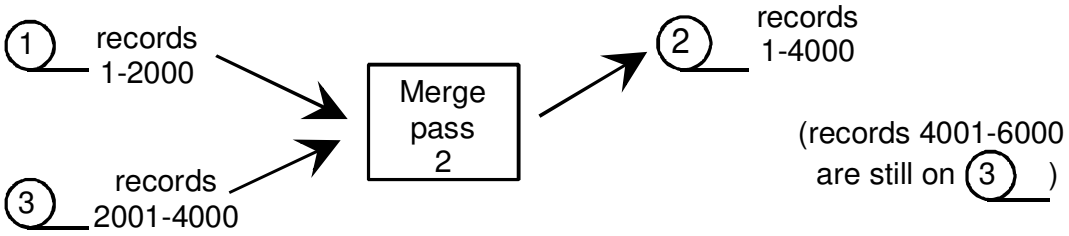
(1) Distribute sorted runs onto 2 files (may be done in conjunction with internal sort phase)

(2) Merge pass 1:

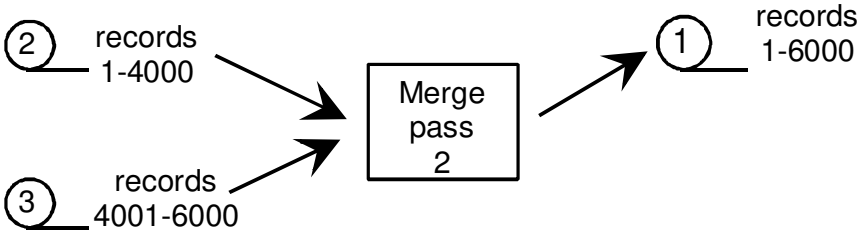


(3) Distribute sorted runs from (3) onto 2 files)

(4) Merge pass 2:

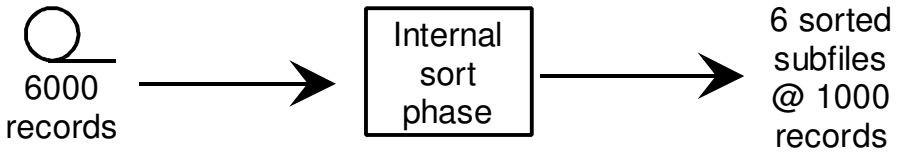


(5) Merge pass 3:



Beispiel 2: natürliches Dreiwege-Mischen

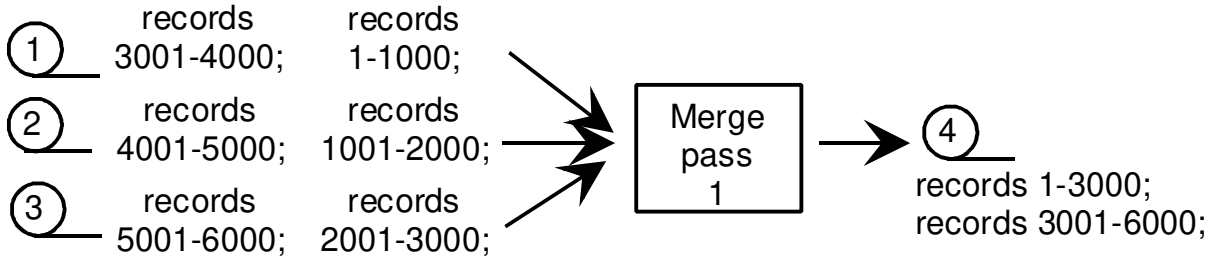
Sorting and Merging Files



3-way merge: Assume 3 input files and 1 output file

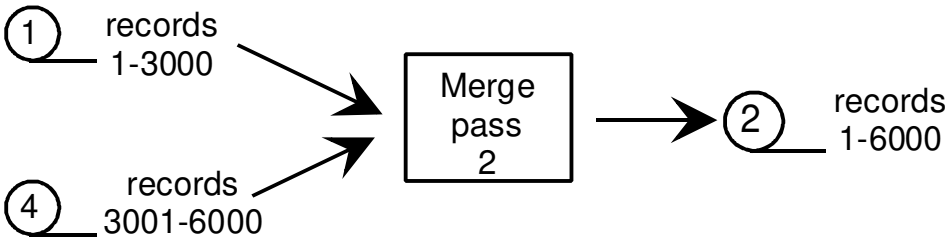
(1) Distribute sorted runs onto 3 files (may be done in conjunction with internal sort phase)

(2) Merge pass 1:



(3) Distribute sorted runs from (4) onto 3 files (here only 2 are used)

(4) Merge pass 2:



Entscheidender Nachteil:

ca. die Hälfte der Lese/-Schreibvorgänge dient nur dem Umkopieren zur Vorbereitung des nächsten Durchlaufs!

3.6.2 Balanciertes Mehrwege-Mischen

Idee: sofortige Verteilung der intern sortierten Sequenzen (runs) auf k Ausgabebänder

Algorithmus: "balanced merging"

Die zu sortierenden N Datensätze stehen auf einem Magnetband, dem Eingabe-Band t_1. Die Anzahl N der Datensätze ist so groß, daß nicht alle Datensätze gleichzeitig im Hauptspeicher Platz finden. Den externen Mergesort- Varianten liegt nun folgende naheliegende Idee zugrunde.
Man teilt die N Datensätze gedanklich in Teilfolgen von jeweils höchstens *l* Datensätzen, wobei *l* die Anzahl der Datensätze ist, die höchstens im Internspeicher Platz finden. Dann betrachtet man der Reihe nach jede dieser Teilfolgen separat. Man liest die Teilfolge ganz in den Internspeicher ein, sortiert sie mit einem der bereits bekannten Verfahren und schreibt die sortierte Teilfolge auf den Externspeicher.
Schließlich verschmilzt man die sortierten Teilfolgen.

Beispiel 1 für Zweiwege-Mischen

Zwei Eingabebänder, zwei Ausgabebänder

Der Hauptspeicher des Rechners fasse drei Datensätze ($I = 3$), und die Folge der zu sortierenden Schlüssel sei

$F = 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8.$

Anfangs stehen die Datensätze, hier repräsentiert durch ihre Schlüssel, auf Band t_1; alle Bänder sind leer:

t_1 : 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8.
t_2 :
t_3 :
t_4 :

Zunächst werden jeweils I Datensätze von t_1 gelesen, sortiert, und abwechselnd auf t_3 und t_4 aufgeteilt. Nach den ersten drei gelesenen Datensätzen ergibt sich folgendes Bild:

t_1 : 12, 5, 2, <u>15</u>, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8.
t_2 : _
t_3 : 2, 5, 12_
t_4 : _

Die Stelle des nächsten Datensatzes jedes Bandes ist unterstrichen. Nachdem alle Datensätze auf t_3 und t_4 verteilt sind, haben wir folgende Situation (Runs sind durch **;** getrennt):

t_1 : 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8._
t_2 : _
t_3 : 2, 5, 12; 1, 4, 14; 7, 8, 11._
t_4 : 6, 13, 15; 3, 9, 10._

Alle Bänder werden zurückgespult, t_3 und t_4 werden gelesen, t_1 und t_2 beschrieben. Der ursprüngliche Inhalt von t_1 ist damit verloren. Wir zeigen die Situationen, die sich jeweils nach einer Verschmelzungs- und Verteilungsphase ergeben:

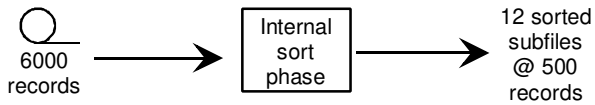
t_1 : 2, 5, 6, 12, 13, 15; 7, 8, 11._
t_2 : 1, 3, 4, 9, 10, 14._
t_3 : _
t_4 : _

t_1 : _
t_2 : _
t_3 : 1, 2, 3, 4, 5, 6, 9, 10, 12, 13, 14, 15._
t_4 : 7, 8, 11._

t_1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15._
t_2 : _
t_3 : _
t_4 : _

Beispiel 2 für balanciertes Zweiwege-Mischen

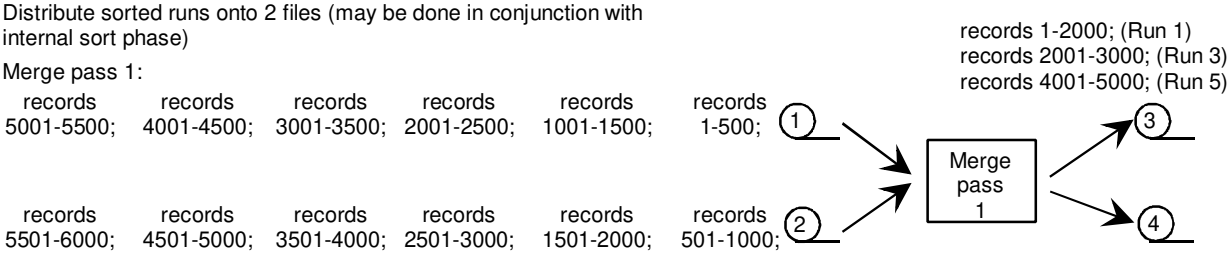
Sorting and Merging Files



2-way balanced merge: Assume 4 files

(1) Distribute sorted runs onto 2 files (may be done in conjunction with internal sort phase)

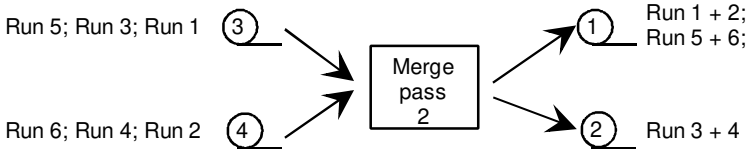
(2) Merge pass 1:



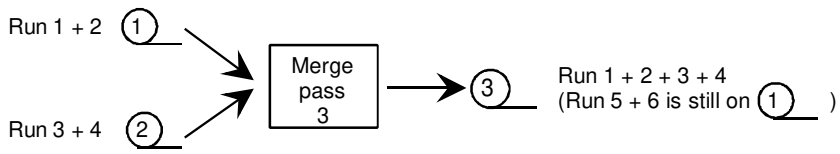
records 1-2000; (Run 1)
records 2001-3000; (Run 3)
records 4001-5000; (Run 5)

records 1001-2000; (Run 2)
records 3001-4000; (Run 4)
records 5001-6000; (Run 6)

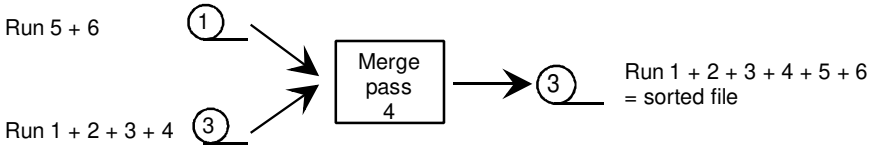
(3) Merge pass 2:



(4) Merge pass 3:



(5) Merge pass 4:



Note that Record 1 is read and written 5 times.

Verallgemeinerung auf k Eingabebänder

"balanced multiway merge"

Beispiel 3: Dreiwege-Mischen

Drei Eingabebänder, drei Ausgabebänder

Eingabe: ASORTINGANDMERGEINGEXAMPLE

Erste Phase:

Band 1	A	O	S	D	M	N	A	E	X
Band 2	I	R	T	E	G	R	L	M	P
Band 3	A	G	n	G	I	N	E		
Band 4									
Band 5									
Band 6									

Zweite Phase:

Band 1								
Band 2								
Band 3								
Band 4	A	A	G	I	N	O	R	S	T
Band 5	D	E	G	G	I	M	N	N	R
Band 6	A	E	E	L	M	P	X		

Das Programm dazu ist trivial.

Komplexität des Mehrwege-Mischens

Annahme: Die Kosten für die Ein- und Ausgabe dominieren, die internen Sortierkosten sind vernachlässigbar.

Anzahl Elemente in der Eingabefolge: N
Größe des Hauptspeichers [Elemente]: M
Wege beim Mischen: P

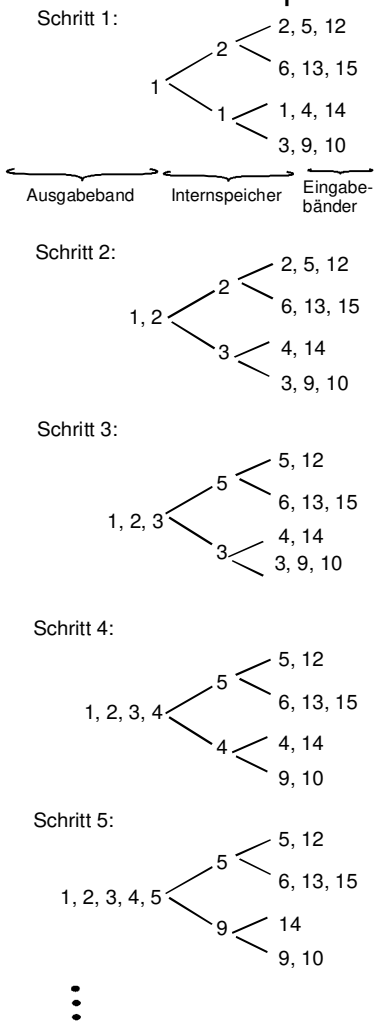
Jeder Durchlauf erzeugt N/M sortierte Blöcke.

Jeder Durchlauf verringert die Anzahl der zu sortierenden Blöcke um den Faktor P. Die Anzahl der erforderlichen Durchläufe ist demnach $\log_p (N/M)$.

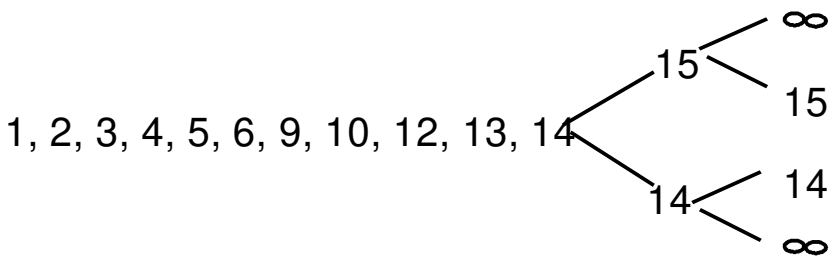
Optimierung für große Hauptspeicher = Heap

Damit das Verschmelzen von *k* Datensätzen im Internspeicher geschehen kann, muß **k** ≤ *l* gelten. Für kleine Werte von *k* mag es vernünftig sein, unter den gerade betrachteten *k* Datensätzen denjenigen mit minimalem Schlüssel durch lineares Durchsehen aller *k* Schlüssel zu bestimmen; für größere Werte von *k* ist es vorteilhaft, das *k*-Wege-Verschmelzen mittels einer Halde (Heap) oder eines Auswahlbaumes (selection tree) zu unterstützen. In einer solchen Datenstruktur kostet das Entfernen des Minimums und Hinzufügen eines neuen Schlüssels nur O(log*k*) Schritte (interne Vergleichs- und Rechenoperationen), wenn *k* Schlüssel gespeichert sind (vgl. Abschnitt 2.3 und Kapitel 6).

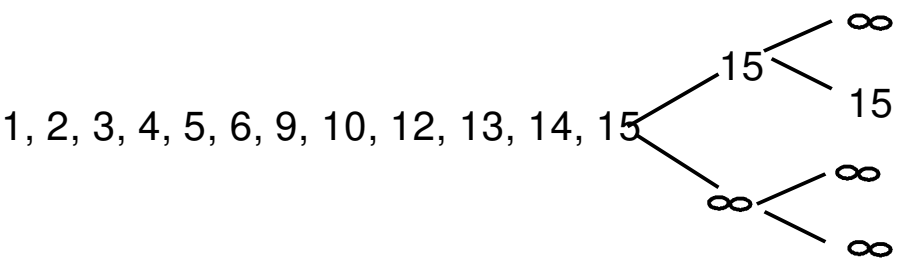
Beispiel: Zweiweg-Mischen mit Heap unf "replacement selection"



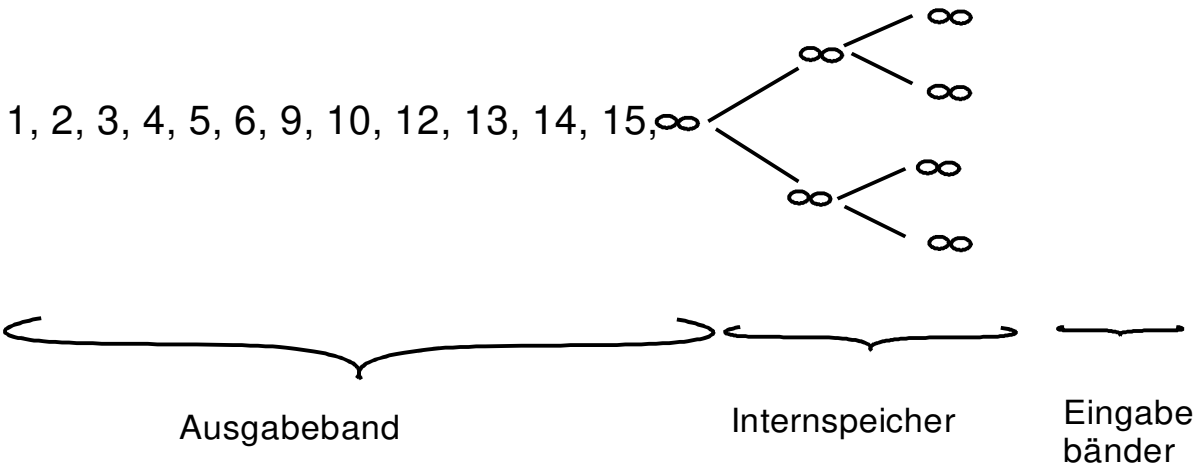
Schritt 11:



Schritt 12:



Schritt 13:



Anmerkung: Im Unterschied zu unserer ursprünglichen Definition bildet hier stets das kleinste Element die Wurzel des Heaps. Diese Methode heißt *Auswahl und Ersetzen* (replacement selection).

Turnier mit k.o.-System

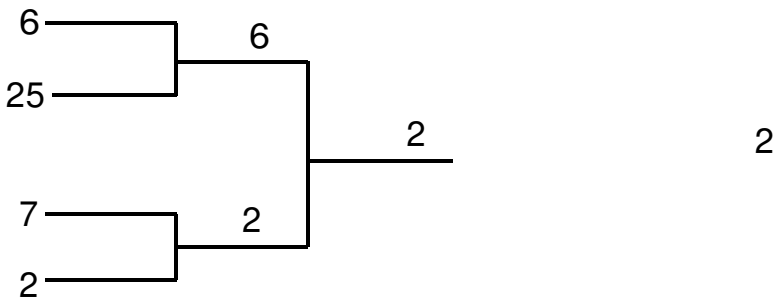
Zwar sieht in jedem Schritt der internen Auswahl und nach jeder Ersetzung die Hauptspeicher-Datenstruktur aus wie ein Heap (mit kleinstem Element an der Wurzel) und wird auch in vielen Lehrbüchern so eingeführt, doch erfolgt das "Nachfüllen" von Elementen in der Praxis ganz anders als beim Heapsort. Man stelle sich vielmehr einen Binärbaum wie bei einem Ping-Pong-Turnier mit k.o.-System vor, den nachrückende Elemente nach der Turnierregel durchlaufen. Das Verfahren wird auch als "tournament sort" bezeichnet.

Turniersystem zum internen "Merging"

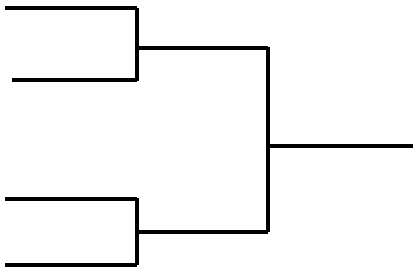
Beispiel

Ausgangsfolge: 6, 25, 7, 2, 14, 10, 9, 22,
3, 14, 8, 12, 1, 30, 13

Schritt 1 **Ausgabe**



Die weiteren Schritte



Regel:
Wenn ein neu eintreffendes Element kleiner ist als das zuletzt in die Ausgabe geschriebene Element, wird es markiert und blockiert den Eingang bis zum Beginn der nächsten Segmente. Sind alle Eingänge blockiert, so werden alle Markierungen gelöscht, und es beginnt eine neue Sequenz.

Dieselbe Methode läßt sich in einem Vorlauf zu Verbesserung des ersten Durchlaufs (für jeweils *ein* Band) verwenden.

Beispiel:

Betrachten wir die Folge F = 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8.
für l = 3:

Anfangs: | 12, 5, 2 | 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8

Schritt 1: 2 | 12, 5, 15 | 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8
Schritt 2: 2, 5 | 12, 15, 13 | 6, 14, 1, 4, 9, 10, 3, 11, 7, 8
Schritt 3: 2, 5, 12 | 15, 13, 6 | 14, 1, 4, 9, 10, 3, 11, 7, 8
Schritt 4: 2, 5, 12, 13 | 15, 6, 14 | 1, 4, 9, 10, 3, 11, 7, 8
Schritt 5: 2, 5, 12, 13, 14.| 15, 6, 1 | 4, 9, 10, 3, 11, 7, 8
Schritt 6: 2, 5, 12, 13, 14, 15 | 6, 1, 4 | 9, 10, 3, 11, 7, 8

Alle Schlüssel im Internspeicher sind kleiner als der zuletzt ausgegebene; daher wird ein neuer Run angefangen.

Schritt 7: 2, 5, 12, 13, 14, 15; 1 | 6, 4, 9 | 10, 3, 11; 7, 8
Schritt 8: 2, 5, 12, 13, 14, 15; 1, 4, | 6, 9, 10 | 3, 11; 7, 8
.
.
.
Schritt 9: 2, 5, 12, 13, 14, 15; 1, 4, 6, 9, 10, 11; 3; 7, 8

Am Beispiel erkennt man, daß die Länge von Runs *l* übersteigen kann. Man kann zeigen, daß beim Verfahren Auswahl und Ersetzen die durchschnittliche Länge von Runs 2 * *l* ist, Runs also im Mittel doppelt so lang sind wie beim internen Sortieren. Außerdem werden Runs, die schon in der Eingabefolge vorhanden sind, noch verlängert, also eine Vorsortierung berücksichtigt.

Komplexität

Für zufällige Schlüssel besitzen die Sequenzen, die aus einem einzigen Band durch Replacement Selection erzeugt werden, ungefähr die doppelte Größe des verwendeten Heaps.

Wenn die Schlüssel etwas sortiert sind, können die Sequenzen nocht wesentlich länger sein.

Fazit:
Durch Replacement Selection sowohl im vorlauf als auch in der Mischphase kann die Anzahl der Durchläufe halbiert werden.

Mit P + 1 Bändern kann in

$$O(\log_p(N/2M))$$

sortiert werden.