

Sortieren sequentieller Files

Wenn die zu sortierende Daten nicht in den Hauptspeicher des Rechners passen (z.B. Massen-datenverarbeitung in einer Bank), sondern auf externen Medien (z.B. Bändern) abgelegt sind, dann sind die bisherigen Sortierverfahren nicht mehr verwendbar.

Hier werden Verfahren benötigt, bei denen nicht auf jede Komponente zugreifbar ist, sondern bei denen das Medium sequentiell durchlaufen wird.

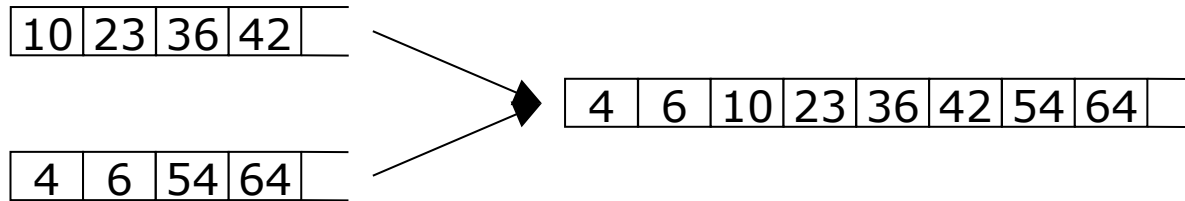
Inhalt

1. Direktes Mischen.....	2
2. Natürliches Mischen.....	12

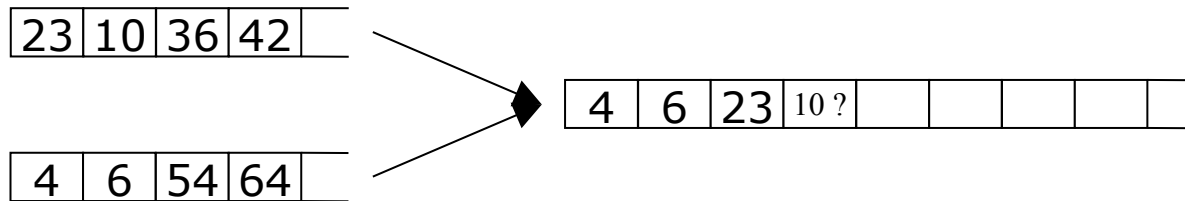
1. Direktes Mischen

Sortieren durch Mischen (engl. merge sort) bedeutet das Zusammenfließen von zwei (oder mehreren) **geordneten** Sequenzen zu einer einzigen sortierten Sequenz durch fortgesetzte Auswahl aus den gerade zugreifbaren Komponenten.

Mischen (Zugriff immer nur auf das erste Element):



Was passiert, wenn die beiden Sequenzen nicht bereits sortiert sind:



Mischen wird als Elementaroperation zur Realisierung von Sortieren verwendet.

Eine einfache Art des Sortierens durch Mischen ist **direktes Mischen**:

1. Zerlege die (unsortierte) Sequenz a in zwei Hälften b und c.
2. Mische b und c durch Kombination einzelner Elemente zu geordneten Paaren.
3. Zerlege die neue, wieder mit a bezeichnete Sequenz in zwei Hälften b und c.
4. Mische b und c durch Kombination von geordneten Paaren zu geordneten Quadrupeln.
5. Zerlege die neue, wieder mit a bezeichnete Sequenz in zwei Hälften b und c.
6. Mische b und c durch Kombination von geordneten Quadrupeln zu geordneten Octupeln
7. Wiederhole den Zerlegungs- und Mischschritt bis die gesamte Sequenz geordnet ist.

Beispiel:

1. Durchlauf:

44	55	12	42	94	18	6	67
----	----	----	----	----	----	---	----

Zerlegen:

44	55	12	42
----	----	----	----

94	18	6	67
----	----	---	----

Mischen:

44	94	18	55	6	12	42	67
----	----	----	----	---	----	----	----

2. Durchlauf:

44	94	18	55	6	12	42	67
----	----	----	----	---	----	----	----

Zerlegen:

44	94	18	55
----	----	----	----

6	12	42	67
---	----	----	----

Mischen:

6	12	44	94	18	42	55	67
---	----	----	----	----	----	----	----

3. Durchlauf:

6	12	44	94	18	42	55	67
---	----	----	----	----	----	----	----

Zerlegen:

6	12	44	94
---	----	----	----

18	42	55	67
----	----	----	----

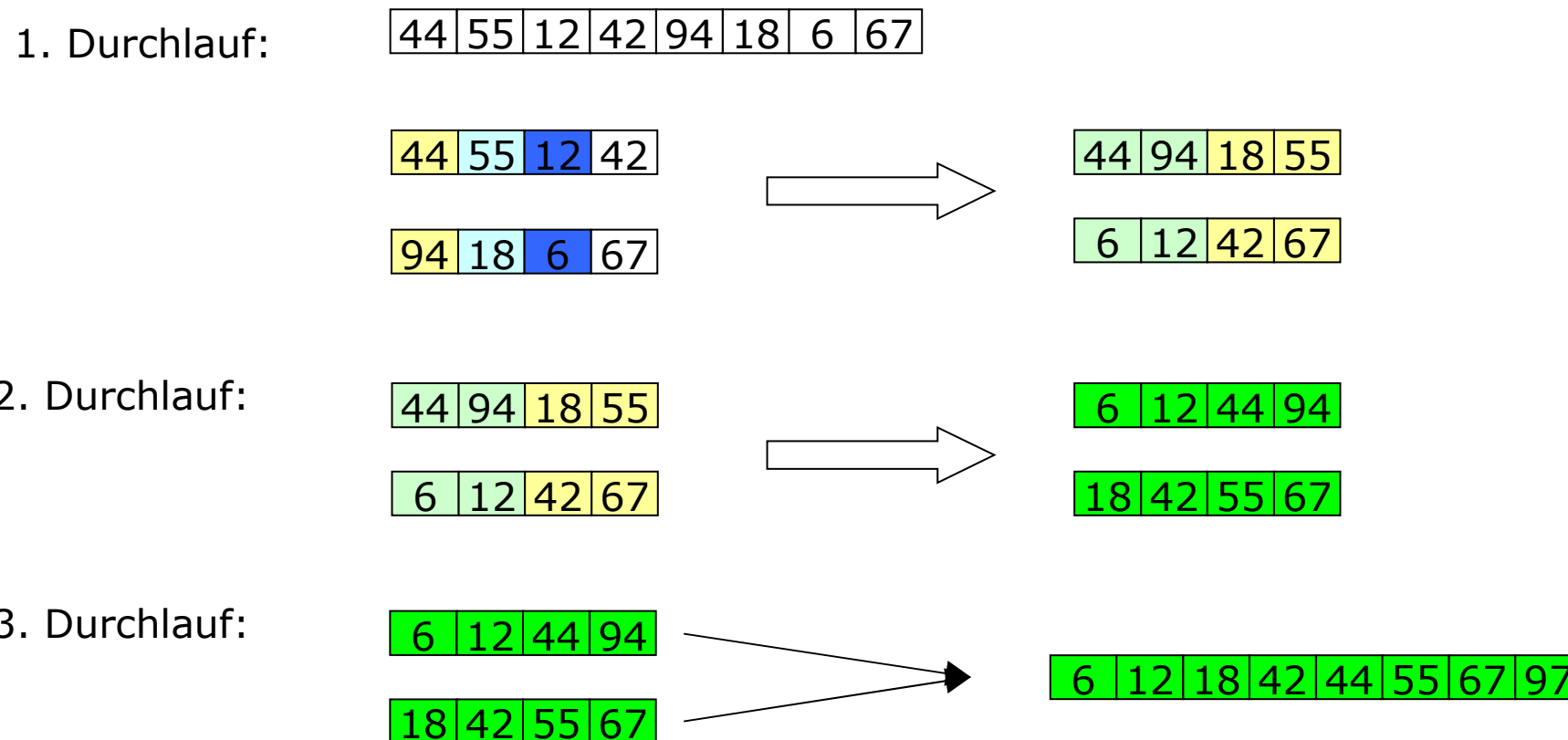
Mischen:

6	12	18	42	44	55	67	97
---	----	----	----	----	----	----	----

Im Beispiel sind 3 Durchläufe und jeweils eine Zerlegungs- und eine Mischphase erforderlich. Erforderlich sind dazu drei Bänder (a, b, c); man nennt das Verfahren deshalb auch **3-Band-Mischen** (engl. 3-tape merge).

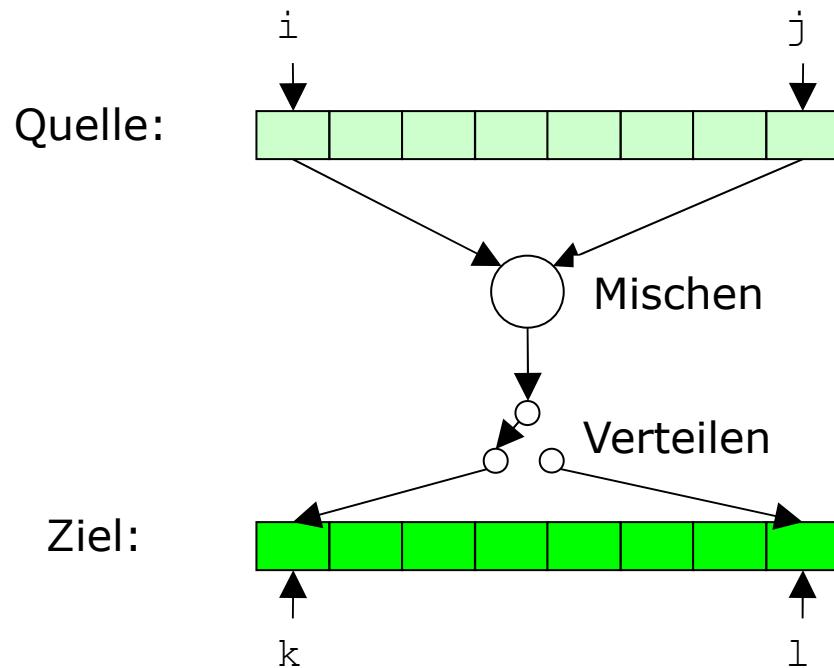
Die **Zerlegungsphase** ist **unproduktiv**, sie trägt nichts zur Sortierung bei. Man kann sie verhindern, indem man sofort beim Mischen auf zwei Bänder verteilt. Dieses Verfahren heißt dann **1-Phasen-Mischen**, im Gegensatz zum ersten Verfahren, dem 2-Phasen-Mischen.

Dabei sind dann nur **halb so viele Kopiervorgänge** erforderlich, dafür aber ein **viertes Band**:



Zur Implementierung des ersten Verfahrens wird ein **Array** verwendet, das aber streng sequentiell durchlaufen wird. Anstelle von zwei Dateien kann man ein Array verwenden, wenn man es als zweiseitige Sequenz betrachtet.

Damit sieht die kombinierte **Misch-Aufteilungsphase** wie folgt aus:



Das Ziel der gemischtem Elemente wird nach jedem geordneten Paar im ersten Lauf, nach jedem geordneten Quadrupel im zweiten Lauf usw. geändert, so dass die beiden Zielsequenzen (Enden eines einzelnen Array) gleichmäßig aufgefüllt werden. Nach jedem Lauf ändern die beiden Arrayenden ihre Rolle: das Ziel wird Quelle und umgekehrt.

Zur weiteren Vereinfachung stellen wir die Quelle und das Ziel als ein Array doppelter Größe dar, wobei i und j jeweils die Quellelemente und k und l die Zielelemente darstellen; Ausgangswerte sind $a[0], \dots, a[n-1]$.

Die Richtung des Datenflusses wird durch eine boolesche Variable up verwaltet: $up==true$ bedeutet, dass $a[0], \dots, a[n-1]$ zu den Positionen $a[n], \dots, a[2*n-1]$ hinauf bewegt werden. up ändert sich in jedem Durchlauf.

Somit sieht das Verfahren wie folgt aus:

```

void directMerge(int data[], int n) {
    bool up = true;           // markiert Richtung
    int p = 1;                // gibt Größe des sortierten Teilbereichs an: 1,2,4,8,16, ...
    int i, j;                 // Quellindeizes
    int k,l;                  // Zielindeizes
    do {
        if (up) {
            i = 0; j = n-1; k = n; l = 2*n-1;
        } else {
            k = 0; l = n-1; i = n; j = 2*n-1;
        }
        // mische die p-Tupel der i- und j-Sequenzen zu k- und l-Sequenzen
        .....                // Das muss noch programmiert werden
        up = !up;             // wechsele Richtung
        p = 2*p;              // verdopple Mischbreite
    } while (p<n);
}

```

Das Mischen der p-Tupel kann wie folgt programmiert werden:

```
do { // mische die p-Tupel der i- und j-Sequenzen zu k- und l-Sequenzen
    if (m>=p) q = p; // Länge des i-ten Laufs
    else q = m;
    m= m-q;
    if (m>=p) r = p; // Länge des j-ten Laufs
    else r = m;
    m= m-r;
    // mische
    ....
    // kopiere Rest des j-ten Laufs
    ....
    // kopiere Rest des i-ten Laufs
    ....
    // vetausche k und l
    ...
} while (m > 0);
```

Somit hat das vollständige Programm das folgende Aussehen:

```
$ cat directMerge.cpp
void directMerge(int data[], int n) {
    bool up = true;           // markiert Richtung
    int p = 1;                // gibt Größe des sortierten Teilbereichs an: 1,2,4,8,16, ...
    int i, j;                 // Quellindezees
    int k,l;                  // Zielindezees
    do {
        int h = 1;
        int q, r;           // Länge der zu mischenden Sequenzen
        int m = n;         // Anzahl der zu mischenden Elemente
        if (up) {
            i = 0; j = n-1; k = n; l = 2*n-1;
        } else {
            k = 0; l = n-1; i = n; j = 2*n-1;
        }
        do {               // mische die p-Tupel der i- und j-Sequenzen zu k- und l-Sequenzen
            if (m>=p) q = p;           // Länge des i-ten Laufs
            else q = m;
            m= m-q;
            if (m>=p) r = p;           // Länge des j-ten Laufs
            else r = m;
            m= m-r;
        }
    }
}
```

```

while (q>0 && r>0) { // mische
    if (data[i] < data[j]) { // bring Element i nach k
        data[k] = data[i]; k = k+h; i++; q--;
    } else { // bring Element j nach k
        data[k] = data[j]; k = k+h; j--; r--;
    }
}
if (q==0) { // kopiere Rest des j-ten Laufs
    while (r>0) {
        data[k] = data[j]; k = k+h; j--; r--;
    }
} else { // kopiere Rest des i-ten Laufs
    while (q>0) {
        data[k] = data[i]; k = k+h; i++; q--;
    }
}
{ int t; // vertausche k und l
  h = -h; t = k; k = l; l = t;
}
} while (m > 0);
up = !up;
p = 2*p;
} while (p<n);

```

```

    if (! up)
        for (int i=0; i<n; i++)
            data[i] = data[i+n];
}
$

```

Verdeutlichen Sie sich die Ausgabe an einem Beispiel.

Analyse des Verfahrens:

Jeder Durchlauf verdoppelt den Wert von p ($p=2*p$). Die Sortierung ist abgeschlossen, sobald $p \geq n$ ist. Somit existieren **$\log(n)$ Durchläufe**.

Da jeder Durchlauf alle n Elemente genau einmal kopiert ist die Gesamtzahl der **Bewegungen = $n * \log(n)$** .

```

$ directMerge
Integer 4
Integer 3
Integer 1
Integer 2
Integer 5
Integer 8
Integer 7
Integer 6
i:0 j:7 k:8 l:15 data: 4 3 1 2 5 8 7 6 0 0 0 0 0 0 0 0
i:1 j:6 k:15 l:10 data: 4 3 1 2 5 8 7 6 4 6 0 0 0 0 0 0
i:2 j:5 k:10 l:13 data: 4 3 1 2 5 8 7 6 4 6 0 0 0 0 7 3
i:3 j:4 k:13 l:12 data: 4 3 1 2 5 8 7 6 4 6 1 8 0 0 7 3
i:8 j:15 k:0 l:7 data: 4 3 1 2 5 8 7 6 4 6 1 8 5 2 7 3
i:10 j:13 k:7 l:4 data: 3 4 6 7 5 8 7 6 4 6 1 8 5 2 7 3
i:0 j:7 k:8 l:15 data: 3 4 6 7 8 5 2 1 4 6 1 8 5 2 7 3
1 2 3 4 5 6 7 8
$

```

Die Anzahl der Vergleiche ist geringer als n , da die Kopieroperation für den Rest keine Vergleiche braucht.

Da der Algorithmus „Mischsortieren“ i.A. für externe Speichermedien verwendet wird, ist die Anzahl exakte Vergleiche uninteressant, das sie gegenüber Zugriffen auf die externen Speicher (Bewegungen) nicht ins Gewicht fällt.

2. Natürliches Mischen

Direktes Mischen nutzt es **nicht** aus, dass eine **Teilsequenz** bereits **sortiert** ist.

Beliebige bereits sortierte Teilsequenzen der Längen n und m können aber in einem Lauf zu einer Sequenz aus $n+m$ Elementen gemischt werden.

Natürliches Mischen mischt die maximal bereits sortierte Teilsequenzen zu einer sortierten Sequenz.

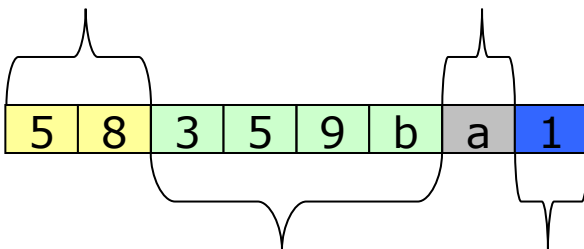
Eine geordnete Teilsequenz a_i, a_{i+1}, \dots, a_j heißt *maximaler Lauf* oder kurz Lauf, wenn folgende Bedingung erfüllt ist:

$$1 : a_k \leq a_{k+1} \text{ für } k = i, \dots, j-1$$

$$2 : a_{i-1} > a_i$$

$$3 : a_j > a_{j+1}$$

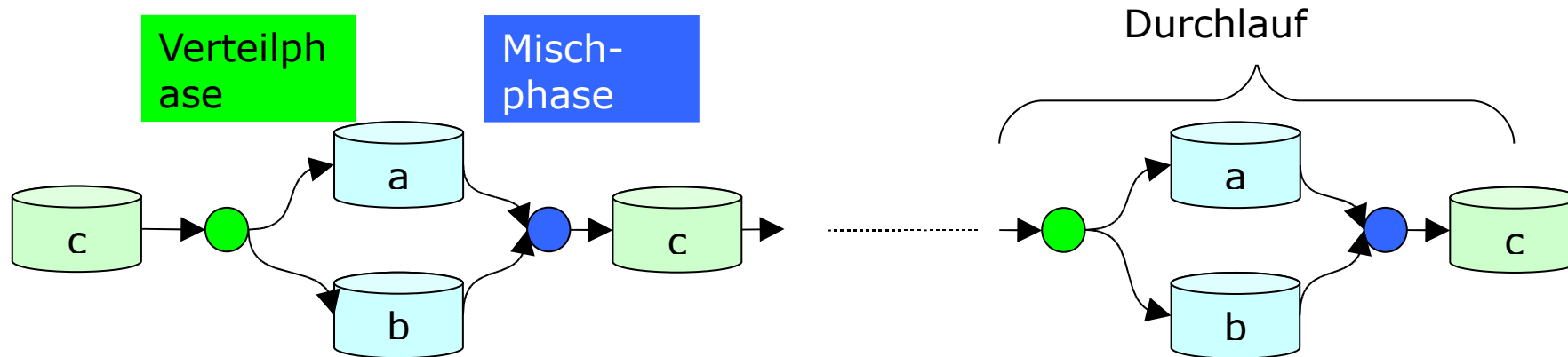
Also, informal sind die Elemente innerhalb des Laufs aufsteigend, der linke Nachbar des kleinsten Elements des Laufs ist größer und der rechte Nachbar des größten Elements des Laufs ist kleiner.



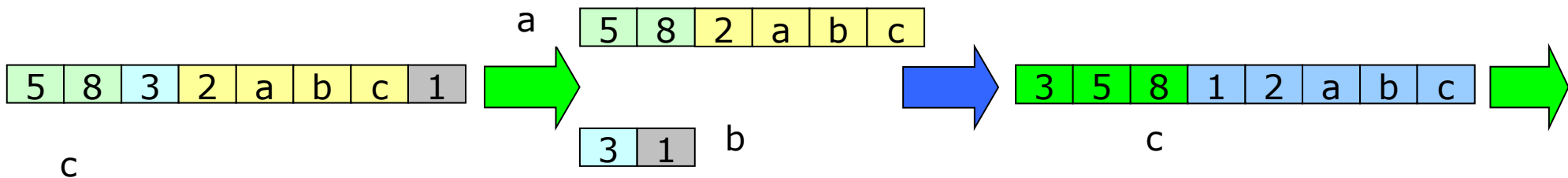
Eine natürliche Mischsortierung mischt maximale Läufe statt fester Sequenzen.

Wir realisieren ein Verfahren für das natürliche Mischen von Dateien als 2-Phasen 3-Band Mischsortierer.

Das Originalband mit den zu sortierenden Werten sei die Datei c , die Hilfsbänder die Dateien a und b . Jeder **Durchlauf** umfasst eine **Verteilphase**, die die Daten von c auf die beiden Hilfsbänder aufteilt und eine **Mischphase**, die die Daten der Hilfsbänder auf c mischt:



Beispiel:



Das Verfahren ist **beendet**, wenn auf dem Band c nur noch **ein Lauf** vorhanden ist.

Nun wird das C++ Programm hergeleitet. Wir verwenden die Variable l , um die Läufe auf dem Band c zu zählen und ein Flag eor , um das Ende eines Laufs zu markieren.

```

$ cat naturalMerge.cpp
...
int l; // Zahl gemischter Läufe
bool eor=false; // end of run Anzeige

void naturalMerge(string c /*Originalband */) {
    string a="A.dat", b="B.dat"; // Hilfsbänder zum Verteilen der Läufe

    do {
        distribute(c, a, b); // Verteilen c auf a und b
        l = 0;
        merge(a, b, c); // Mischen von a und b auf c
    } while (l != 1);
} // naturalMerge
...

```

Die beiden Phasen sind unabhängige Prozeduren:

```

void distribute(string c, string a, string b) {// Verteilen von c auf a und b
    ifstream cS;
    ofstream aS, bS;
    cS.open(c.c_str(), ios::in); // zum Lesen öffnen
    if (!cS) { // Datei kann nicht geöffnet werden
        cerr << c << " kann nicht geöffnet werden!\n"; exit (-1);
    }
    aS.open(a.c_str(), ios::out); // zum Schreiben öffnen
    if (!aS) { // Datei kann nicht geöffnet werden
        cerr << a << " kann nicht geöffnet werden!\n"; exit (-1);
    }
    bS.open(b.c_str(), ios::out); // zum Schreiben öffnen
    if (!bS) { // Datei kann nicht geöffnet werden
        cerr << b << " kann nicht geöffnet werden!\n"; exit (-1);
    }

    do { // verteil Läufe von c auf
        copyrun(cS, aS); // ... a
        if (!cS.eof())
            copyrun(cS, bS); // ... b
    } while (!cS.eof());

    cS.close(); aS.close(); bS.close(); // schliesse Dateien
} // distribute

```

```

void merge(string a, string b, string c) { // Mischen von a und b auf c
    ofstream cS;
    ifstream aS, bS;
    cS.open(c.c_str(), ios::out); // zum Schreiben öffnen
    if (!cS) { // Datei kann nicht geöffnet werden
        cerr << c << " kann nicht geöffnet werden!\n"; exit (-1);
    }
    aS.open(a.c_str(), ios::in); // zum Lesen öffnen
    if (!aS) { // Datei kann nicht geöffnet werden
        cerr << a << " kann nicht geöffnet werden!\n"; exit (-1);
    }
    bS.open(b.c_str(), ios::in); // zum Lesen öffnen
    if (!bS) { // Datei kann nicht geöffnet werden
        cerr << b << " kann nicht geöffnet werden!\n"; exit (-1);
    }
    while (!aS.eof() && !bS.eof()) { // Mischen
        mergerun(aS, bS, cS); l++;
    }
    while (!aS.eof()) { // Restlauf anhängen
        copyrun(aS, cS); l++;
    }
    while (!bS.eof()) {
        copyrun(bS, cS); l++;
    }
} // merge

```

Die beiden Prozeduren verwenden jeweils eine eigene Prozedur, die für die Bearbeitung eines Laufs zuständig ist.

Verwendet wird jedes Mal das Flag `eor`, das anzeigt, dass ein Lauf beendet ist (gemäß der o.a. Definition).

```
void copyrun(ifstream &quelle, ofstream &ziel) { // kopiert einen Lauf von quelle zum
ziel
    do {
        copy(quelle, ziel);
    } while (!eor);
} // copyrun

void mergerun(ifstream &aS, ifstream &bS, ofstream &cS) { // Mergen eines Lauf
do {
    if (aS.peek() < bS.peek()) {
        copy(aS, cS); // Schlüssel kopieren
        if (eor) copyrun(bS, cS); // Rest des Laufs nehmen
    } else {
        copy(bS, cS);
        if (eor) copyrun(aS, cS);
    }
} while (!eor);
} // mergerun
```

Beide Prozeduren verwenden die Prozedur `copy`, die die eigentliche Kopieroperation durchführt. Dabei wird immer ein Element `quelle` zum `ziel` kopiert und geprüft, ob das Ende des Laufs schon erreicht ist.

```
void copy(istream &quelle, ostream &ziel) { // kopiert einen Wert
    int buf;
    buf = quelle.get(); // Lesen eines Elementes
    if (!quelle.eof())
        ziel.put(buf); // Schreiben des eines Elementes
    eor = quelle.eof() || (buf > quelle.peek());
} // copy
```

nur auf die
Datei schauen,
das Element
nicht
konsumieren!

Das gesamte Programm inklusive einer Prozedur zur Ausgabe einer Datei hat dann folgendes Aussehen. Wir behandeln hier Textdateien. Die Änderungen, um Dateien mit Strukturen verarbeiten zu können, bei denen lediglich die Schlüsselkomponente zur Sortierung betrachtet wird, sollte als Heimarbeit versucht werden.

```

// 3-Band 2-Phasen natürliches Mischsortieren
#include <iostream.h>
#include <fstream.h>           // wg. Dateioperationen
#include <string>              // wg. Dateiname

void list(string filename) {   // Ausgabe einer Datei mit char Werten (ohne Fehlerbehandlung)
    ifstream in;
    char buf;

    in.open(filename.c_str(), ios::in);           // zum Lesen öffnen
    if (!in) {                                   // Datei kann nicht geöffnet werden
        cerr << filename << " kann nicht geöffnet werden!\n";
        exit (-1);
    }
    while (!in.eof()) {                          // lesen bis EOF
        in >> buf;
        if (in.eof()) break;
        cout << buf;                             // ausgeben
    }
    in.close();                                  // Datei schliessen
}

int l;                                          // Zahl gemischter Läufe
bool eor=false;                               // end of run Anzeige

void copy(ifstream &quelle, ofstream &ziel) { // kopiert einen Wert
...
}
void copyrun(ifstream &quelle, ofstream &ziel) { //kopiert einen Lauf
...
}
void distribute(string c, string a, string b) { // Verteilen von c auf a und b
...
}

void mergerun(ifstream &aS, ifstream &bS, ofstream &cS) { //Mergen eines Lauf

```

```

...
}
void merge(string a, string b, string c) {           // Miischen von a und b auf c
    ...
}
void naturalMerge(string c /*Originalband */) {
    string a="A.dat", b="B.dat";                   // Hilfsbänder zum Verteilen der Läufe

    do {
        distribute(c, a, b);                       // Verteilen c auf a und b
        l = 0;
        merge(a, b, c);                           // Mischen von a und b auf c
    } while (l != 1);
}

int main(int argc, char *argv[]) {
    string dateiname = "dat";                       // default
    if (argc == 2) {
        dateiname = argv[1];
    }

    cout << "Datei vor Sortierung:" << endl;
    list(dateiname);
    naturalMerge(dateiname);
    cout << endl << "Datei nach Sortierung:" << endl;
    list(dateiname);
    cout << endl;
}

```

```

$ naturalMerge dat
Datei vor Sortierung:
bcd123678123zyx
Datei nach Sortierung:
112233678bcdxyz
$

```

Analyse des Verfahrens:

Der **Aufwand** ist **proportional zur Zahl** der benötigten **Durchläufe**, da nach Definition bei jedem Durchlauf die ganze Menge der Daten kopiert wird. Ein Weg zur **Verbesserung** der Anzahl Durchläufe ist die **Verteilung** auf **mehr** als zwei **Dateien**.

In einem Durchlauf gilt: das **Mischen** von **N Läufen**, die auf **n Bänder** verteilt sind, ergibt eine Sequenz von N/n Läufen. Ein zweiter Durchlauf vermindert ihre Zahl auf N/n^2 , ein dritter Durchlauf auf N/n^3 , und nach k Durchläufen bleiben N/n^k Läufe übrig.

Zum Sortieren von N Elementen werden also insgesamt $\log_n(N)$ Durchläufe benötigt. Da jeder Durchlauf N Kopieroperationen erfordert, ist die **Gesamtzahl** der **Kopieroperationen** im schlimmsten Fall $N * \log_n(N)$.

Weitere Verfahren werden im 2. Semester behandelt:

- Ausgeglichenes n -Wege Mischen
- Mehrphasen Sortieren