Mergesort, Coding Tipps

KW 21, Zentralübung Informatik II

2006-05-23

Externes Sortieren

- Bisher: Wir sind beim Sortieren davon ausgegangen, dass alle Daten in den Hauptspeicher passen.

- Was macht man in Situationen, wo das nicht geht?
  - Bei großen Datenmengen: Oft nur sequentielles Lesen und sequentielles Schreiben möglich.
  - Beispiel: Die Kapazität von Festplatten wächst stärker als die Zugriffszeit oder Übertragungsgeschwindigkeit. Sie werden also in Zukunft klassischen Magnetbändern immer ähnlicher. Selbiges gilt für RAM.

- Lösung: *Externe Sortierverfahren* nehmen auf die erwähnten Beschränkungen Rücksicht.

Ströme (Streams)

Ein Strom ist eine Datenstruktur, die nur sequentiellen Zugriff auf die Daten erlaubt. In Java ist Strömen das Package `java.io` gewidmet. Übliche Operationen für Ströme sind:

- t.reset() : Band t zurückspulen, Lesemodus verwenden.
- d=t.read(): Nächsten Datensatz d von Band t lesen.
- t.eof(): Beim Lesen am Ende von Band t angelangt?
- t.rewrite(): Rückspulen, Schreibmodus verwenden.
- t.write(t): Schreibe Datensatz d auf Band t.

Vergleichbare Operationen bei Arrays? a[i]=x; x=a[i]; a.length

Ausgeglichenes 2-Wege-Mergesort

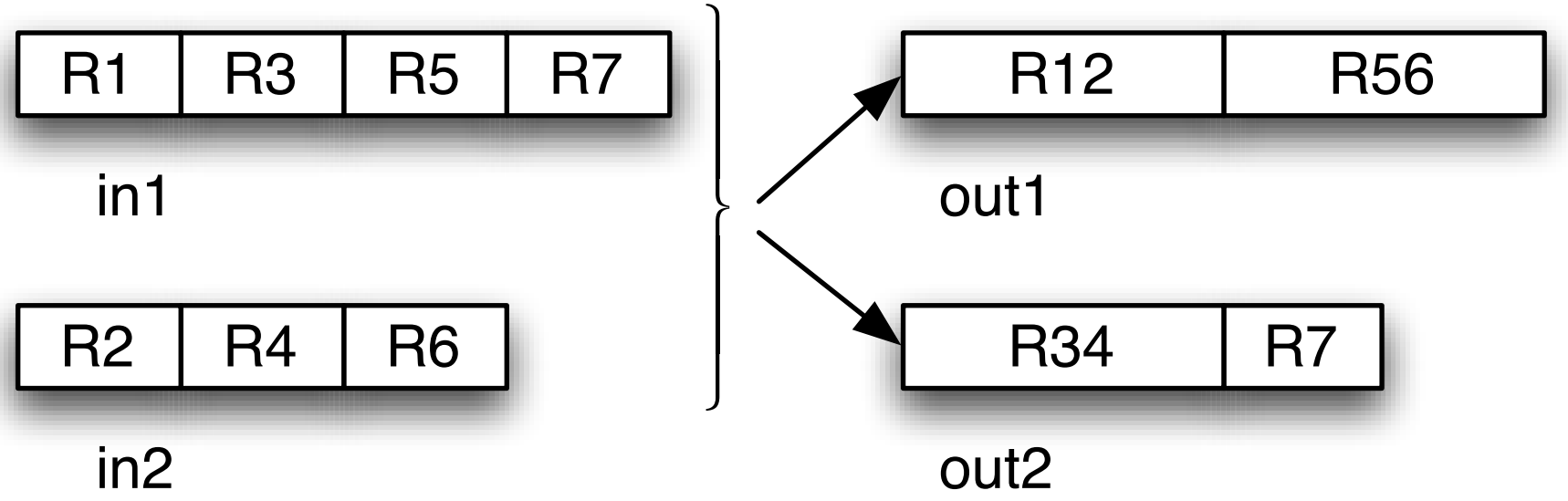
Algorithmus für externes Sortieren.

Vorgaben:

- Im Hauptspeicher ist Platz für I Datensätze,
- es gibt 4 Bänder t_0, t_1, t_2, t_3.

Grundprinzip: Verschmelzen von Runs

Idee: $Runs$ (geordnete Teilsequenzen) der Länge n können zu Runs der Länge $2n$ verschmolzen werden.

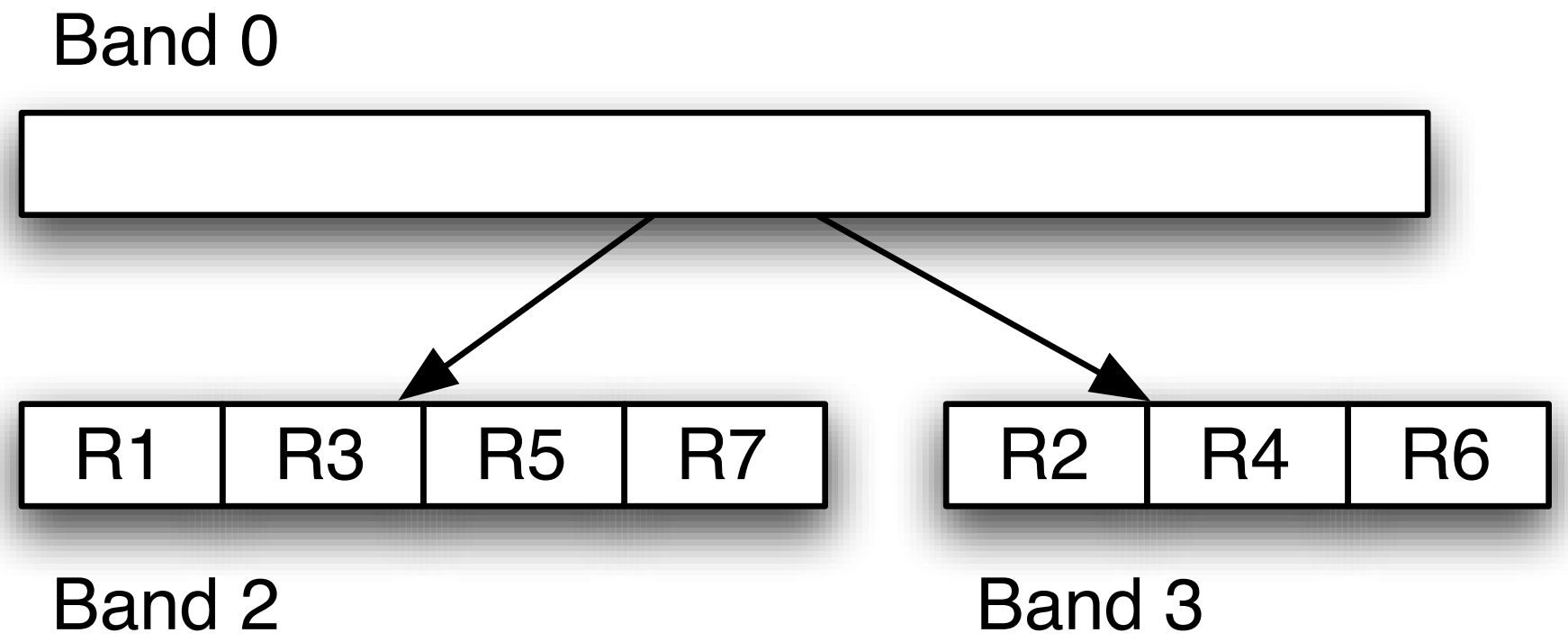


Algorithmus (1/2): Erzeuge Runs

Anfang: Schaffe Runs der Länge I, hierzu wird der Hauptspeicher voll ausgeschöpft.

- Teile Eingabeband t_0 auf Eingabebänder t_2 und t_3 auf.
  - Lies I Datensätze von t_0, sortiere diese und schreibe diese Anfangsruns wechselweise auf t_2 und t_3.
  - Danach: t_2 und t_3 haben Runs der Länge I von sortierten Datensätzen.

Algorithmus (1/2): Erzeuge Runs



Algorithmus (2/2): Verschmelze Runs

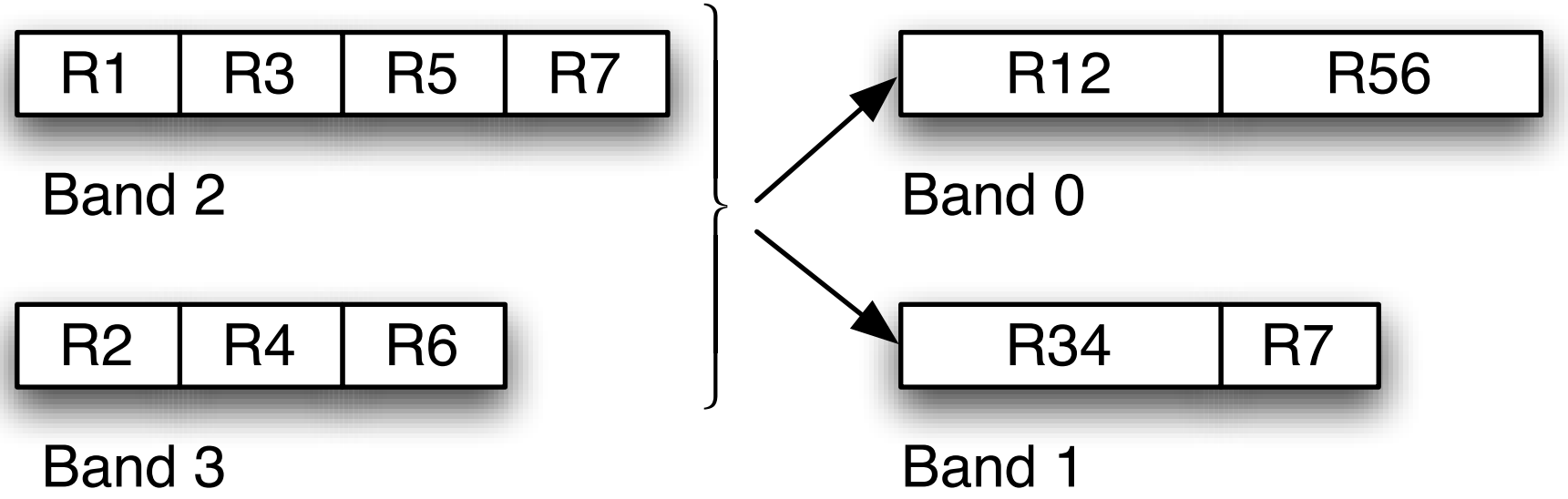
Wenn auf zwei Bändern Runs der Länge n stehen, kann man diese zu einem Run der Länge $2n$ verschmelzen. Man muß nur das Anfangs-Element der Bänder nehmen, das aktuell kleiner ist und auf ein weiteres Band schreiben.
Speicherplatzbedarf: 2 Datensätze (nicht I Datensätze!).

- Verschmelze Runs von t_2, t_3 zu Runs der Länge $2 \cdot I$, die kontinuierlich (wechselweise) auf t_0, t_1 geschrieben werden.

- Wiederhole den letzten Schritt bis die Runlänge so groß ist, dass alle Elemente in einen Run passen.

  Dabei wird bei jedem Mal die Schreib-Lese-Richtung umgekehrt.

Algorithmus (2/2): Verschmelze Runs



Beispiel: l=3 (1/3)

Anfang:

$t_0 : 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8$
$t_1 :$
$t_2 :$
$t_3 :$

Nach den ersten 3 Datensätzen

$t_0 : 12, 5, 2, \underline{15}, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8$
$t_1 : \_$
$t_2 : 2, 5, 12; \_$
$t_3 : \_$

Beispiel (2/3)

Nach 1. Durchlauf: Runlänge 3

$t_0 : 12, 5, 2, 15, 13, 6, 14, 1, 4, 9, 10, 3, 11, 7, 8\_$
$t_1 : \_$
$t_2 : 2, 5, 12 \mid 1, 4, 14 \mid 7, 8, 11 \mid \_$
$t_3 : 6, 13, 15 \mid 3, 9, 10 \mid \_$

Runlänge 6

$t_0 : 2, 5, 6, 12, 13, 15 \mid 7, 8, 11\_$
$t_1 : 1, 3, 4, 9, 10, 14\_$
$t_2 : \_$
$t_3 : \_$

Beispiel (3/3)

Runlänge 12

$t_0 : \_$
$t_1 : \_$
$t_2 : 1, 2, 3, 4, 5, 6, 9, 10, 12, 13, 14, 15 \mid \_$
$t_3 : 7, 8, 11\_$

Fertig:

$t_0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\_$
$t_1 : \_$
$t_2 : \_$
$t_3 : \_$

Quellcode (1/2)

```
public static int mergesort(Stream[] t) {
    createInitialRuns(t[0], t[2], t[3]); // von 0 nach 2,3
    int in1 = 2, in2 = 3, out1 = 0, out2 = 1;
    int runLength = I;
    while(true) {
        rewindEverything(t, in1, in2, out1, out2);
        if (t[in2].eof()) {     // t[in1] ist immer länger!
            // Alles hat auf ein Band, in1, gepasst: fertig
            return in1;
        }
        mergeAllRuns(t, in1, in2, out1, out2, runLength);
        // Nicht gezeigt: Swap: in1 <-> out1, in2 <-> out2
        runLength = runLength * 2;
    }
}
```

Quellcode (2/2)

```
private static void mergeAllRuns(Stream[] t, int in1, int in2,
        int out1, int out2, int runLength) {
    int out = out1;
    // t[in1] ist das längere von beiden Bändern!
    while(! t[in2].eof()) {
        mergeTwoRuns(t[in1], t[in2], t[out], runLength);
        if (out == out1) {
            out = out2;
        } else {
            out = out1;
        }
    }
    copyRest(t[in1], t[out]); // kopiere Rest des längeren Bands
}
```

Analyse

- Vorteil: Schneller als Heapsort für große Datenmengen.
- Nachteil: Hat, verglichen mit anderen Sortierverfahren, ungefähr den doppelten Speicherbedarf.

Komplexität: $O(n \log(\frac{n}{I})) = O(n \log(n))$.

- $\log(\frac{n}{I})$ Durchläufe: Am Anfang gibt es $\frac{n}{I}$ Runs, in jedem Durchlauf wird diese Zahl halbiert.
- Pro Durchlauf: n Zugriffe.

Mergsort-Varianten

- Augeglichenes Mehr-Wege-Mergesort: $2k$-Bänder; statt bisher 2, gibt es dann k Eingabe- und k Ausgabebänder. In jedem Durchgang werden die Runlängen um Faktor k größer.

- Mehrphasen-Mergesort: Nur ein Band dient der Ausgabe, Rest der Eingabe. Sobald ein Eingabeband leer ist, wird dieses zum nächsten Ausgabeband (Ende und Beginn einer „Phase“).

Tipps für lesbaren Code

- Generelle Idee: Code + Doku muss wie ein Handbuch („selbsterklärend“) sein.
- An jeder Stelle sollte alle Information zum Verständnis vorhanden sein.
- Im Team: Auf gemeinsame Formatierungskonventionen (Sprache, Einrückungen, geschweifte Klammern etc.) einigen.

  Tipp: Source → Format in Eclipse ist ein guter, emotionsloser Ratgeber in solchen Fragen.

Coding: JavaDoc schreiben!

```
public static void sort(int[] a, int fromIndex, int toIndex) {
    // ...
}
```

Coding: JavaDoc schreiben!

```
/**
 * Sorts the specified range of the specified array of ints into
 * ascending numerical order.  The range to be sorted extends from ind
 * <tt>fromIndex</tt>, inclusive, to index <tt>toIndex</tt>, exclusive
 * (If <tt>fromIndex==toIndex</tt>, the range to be sorted is empty.)<
 *
 * @param a the array to be sorted.
 * @param fromIndex the index of the first element (inclusive) to be
 *        sorted.
 * @param toIndex the index of the last element (exclusive) to be sort
 */
public static void sort(int[] a, int fromIndex, int toIndex) {
    // ...
}
```

Coding: Methodengröße – vorher

```
public static void algorithm() {
    // step1
    do1a();
    do1b();
    int x=do1c();

    // step2
    do2a(x);
    do2b();
    do2c();
}
```

Coding: Methodengröße – nachher

```
public static void algorithm() {
    int x = step1();
    step2(x);
}
private static int step1() {
    do1a();
    do1b();
    return do1c();
}
private static void step2(int x) {
    do2a(x);
    do2b();
    do2c();
}
```

Tipp: Refactoring „Extract Method“ in Eclipse.

Coding: Sprechende Bezeichnernamen

- openFileAndSortEntries()
- writeAccessCounter
- ensureWriteMode()
- isEmpty(): Methode mit booleschem Rückgabewert.

Coding: Keine überflüssigen Kommentare

Negativbeispiele (wie macht man's besser?):

```
i++; // i um 1 erhoehen

// number auf System.out ausgeben
System.out.println(number);

// Sortiere myArray und gib es aus
doit(myArray);

// hier darf increment auf keinen Fall 0 sein:
result = result + increment;
```

Coding: Namenskonventionen

- Konstanten (`public static final`): Substantive, nur in Großbuchstaben geschrieben, Unterstrich als Worttrenner. Beispiel: `MAX_ITEMS`

  Ansonsten werden Worte ausschliesslich durch Großbuchstaben getrennt: `StringBuffer`

- Klassen: Substantive, groß geschrieben. Beispiele: `String`, `System`

- Interfaces: Adjektive, groß geschrieben. Beispiele: `Clonable`, `Appendable`.

- Methoden: Verben oder Sätze, klein geschrieben. Beispiel: `println()`

- Variablen: Substantive, klein geschrieben. Beispiel: `itemCounter`

- Etc.: Im Zweifelsfall bei der Java-API abschauen.