Radixsort

Finn Jannsen, Philipp Schwarz

14. Mai 2019

Inhaltsverzeichnis

1 Einführung

Diese Dokumentation beschreibt eine Sortieralgorithmus-Implementation für Arrays mit vordefinierten Integer-Keys, basierend auf Radixsort. In Abschnitt 2 wird darauf eingegangen, wie der Algorithmus realisiert wurde. Anschließend wird in Abschnitt 3.1 geprüft, ob die Implementation korrekt funktioniert und in Abschnitt 3.2 die Performance mit dem zu Grunde liegenden Quicksort verglichen.

2 Implementation

Der Algorithmus wurde als Klasse realisiert, der ein Interface implementiert, welches es ermöglicht, einfach weitere Implementationen, sofern dies gewünscht wird, zu schreiben.

2.1 Sortieren

Der bisherige Quicksort-Algorithmus besteht im wesentlichen aus einer Schleife, in der zuerst das erste Element von links, welches gleich oder größer als Pivot ist, gesucht wird. Danach wird das erste Element von rechts, welches kleiner als das Pivot ist, gesucht. Diese werden dann getauscht. Die Schleife wird verlassen, wenn die jeweiligen Such-Schleifen einander kreuzen. Danach wird das Pivot in diese Mitte an Stelle i zurückgetauscht und der Algorithmus wird zwei mal rekursiv gestartet, einmal um die Liste von ganz links bis i-1 zu sortieren und noch einmal um die Liste von i+1 bis rechts zu sortieren.

Radixsort besteht aus mehreren Schritten, und jeder Schritt aus zwei Phasen. Die Partitionierungsphase dient dazu, die Daten auf Fächer aufzuteilen, während in der Sammelphase die Daten aus diesen Fächern wieder aufgesammelt werden. Beide Phasen werden für jede Stelle des zu sortierenden Schlüssels einmal durchgeführt. Die Anzahl der Schritte ist gleich der maximalen Stellenanzahl.

Der Code hierfür ist in 3.2 dargestellt.

3 Testen und Verifikation

3.1 Verifizieren

Der Algorithmus wurde auf seine korrekte Funktionalität getestet. Hierzu zählt natürlich das Vorliegen des Ergebnisses in der korrekten Reihenfolge. Alle Tests wurden erfolgreich mit unterschiedlichen Eingabewerten absolviert.

3.2 Aufwandsanalyse

Für die Aufwandsanalyse wurde der Klasse ein Counter eingeführt, der die Tauschoperationen zählt, als auch die Ausführungszeit beobachtet.

Bisherige Untersuchungen von Quicksort haben bereits einen Asymptotischen Aufwand wie folgt ergeben:

Rechenoperationen bei Best- und Average Case:

$$T(n) = \mathcal{O}(n * ln(n))$$

Rechenoperationen bei Worst Case:

$$T(n) = \mathcal{O}(n^2)$$

Bei der Radixsort-Implementation ist der Asymptotische Aufwand konstant wie folgt (l ist die Länge des Keys):

$$T(n) = \mathcal{O}(l * n)$$

Außerdem wurden dieses mal statt komplett zufälligen Key-Werten welche mit folgender Beschränkung ausgewählt:

$$700 * N \leq key \leq 800 * N$$

Es wurden 10 Durchläufe des Tests mit den Größen $N = 10^k, k = 1, ..., 6$ durchgeführt, um einen angemessenen Mittelwert zu ermitteln.

Die Ergebnisse für die Rechenoperationen sind in Abbildung 3.1 zu sehen. Es ist ersichtlich, dass das Radixsort weniger Rechenoperationen verursacht.

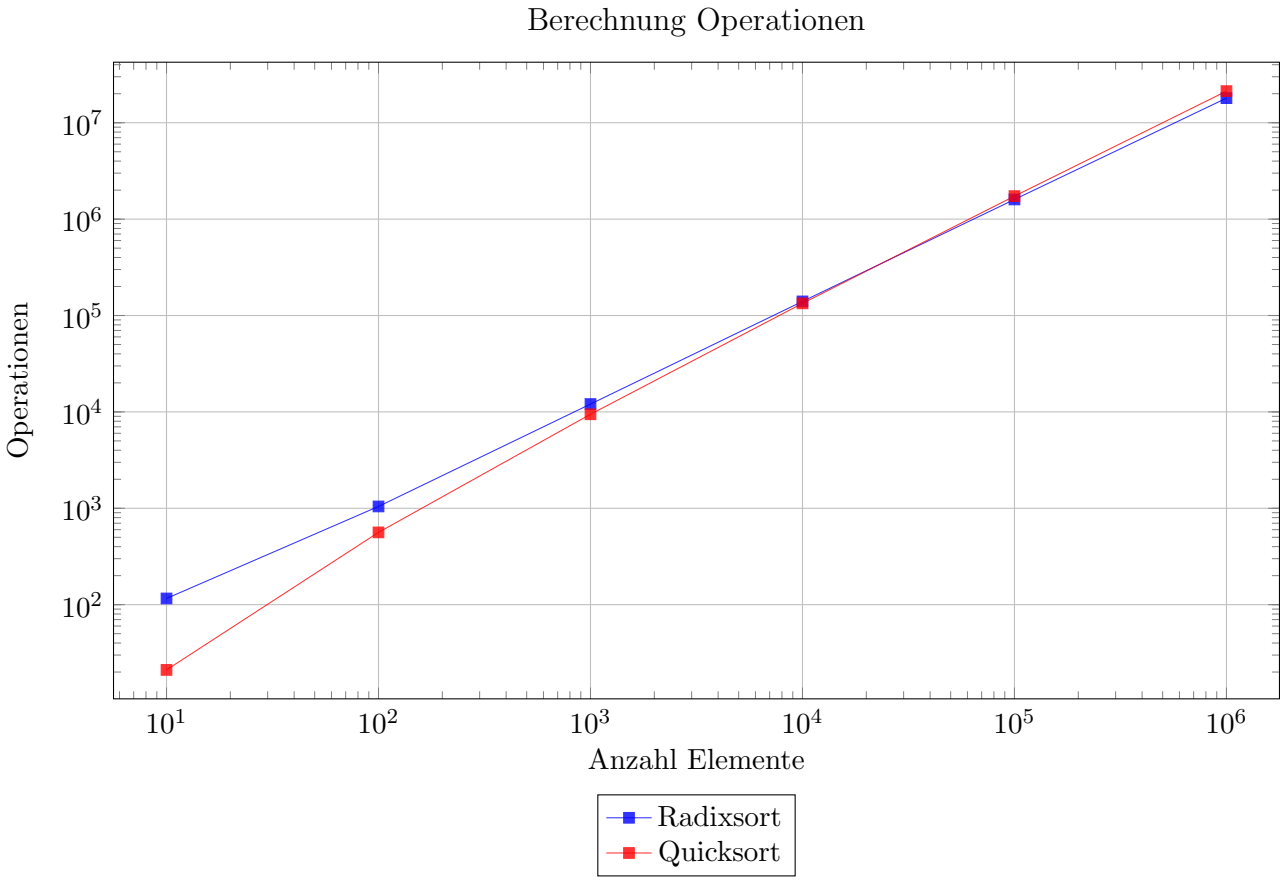

Abbildung 3.1: Quantitativer Vergleich zu Quicksort anhand Rechenoperationen

```
Node[] out = new Node[list.length];

int[] count = new int[10];

for (int i = 0; i < list.length; i++) {
    int digit = getDigit(list[i].getKey(), exp);
    count[digit] += 1;
}

for (int i = 1; i < count.length; i++) {
    count[i] += count[i - 1];
}

for (int i = list.length - 1; i >= 0; i--) {
    int digit = getDigit(list[i].getKey(), exp);

    out[count[digit] - 1] = list[i];
    count[digit]--;
}

return out;
```

Abbildung 3.2: Code-Ausschnitt Sortieren

Viele Duplikate

Kein Insert			Insert 10			Insert 100		
N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs
10	29.64	0	10	23.8	0	10	23.25	0.01
100	571.3	0.02	100	513.23	0	100	2447.28	0
1000	6587.53	0.02	1000	6419.15	0.07	1000	16357.26	0.11
10000	66579.74	0.44	10000	66548.55	0.5	10000	64931.09	0.66
100000	665092.01	8.15	100000	668592.79	7.81	100000	662118.18	8.03
1000000	6649592.4	161.02	1000000	6611998.45	147.22	1000000	6695958.91	159.14

Wenig Duplikate

Kein Insert			Insert 10			Insert 100		
N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs	N	Counter fqs	T in ms fqs
10	30.74	0	10	21.87	0	10	22.04	0
100	657.5	0.02	100	577.57	0.02	100	2464.69	0
1000	10433.3	0.11	1000	9568.93	0.09	1000	21261.65	0.15
10000	143825.56	1.3	10000	135667.7	1.23	10000	253308.34	1.41
100000	1827608.13	19.72	100000	1746850.13	19.47	100000	2922396.04	21.43
1000000	22316105.8	343.26	1000000	21383728.4	338.14	1000000	33143857.1	356.4

Abbildung 3.3: Durchschnittswerte unterschiedlicher Insertionsorts von 100 Durchführungen bei vielen und wenigen Duplikaten