Binärer Suchbaum

Finn Jannsen, Philipp Schwarz

1. Juni 2019

Inhaltsverzeichnis

1 Einführung

Diese Dokumentation beschreibt Implementationen für binäre Suchbäume. 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 Aufbau

Die zu Grunde liegende Aufgabe bestand aus 2 Teilen. Im ersten ging es darum, einen binären Suchbaum zu implementieren, welcher Als Elemente Generics entgegen nimmt. Unter anderem sollte der Baum einmal aus Nodes (dt. Knoten) bestehen und einmal in einen Array eingebettet sein. Letzteres unterstützt nur geringe Baumgrößen, da die Indexe von Childs (dt. Kinder) $2 * i$ aufwärts sind, wobei i der Index von dem Father-Node ist. Das sorgt relativ schnell für interne Integer-Overflows, wobei das Java-Framework auch nur Integer-Größen als Array-Größen akzeptiert. Diese Implementation wurde erfolgreich durchgeführt mit Generics, die Comparableïmplementieren, denn um den Baum sortiert zu erstellen müssen die Elemente miteinander verglichen werden. Der Baum sollte im zweiten Teil so umgebaut werden, dass er Integer als Elemente akzeptiert. Außerdem sollte folgende Funktionalität hinzugefügt werden:

Eingabe: Zwei natürliche Zahlen m und M
Ausgabe: $\sum_i a_i \; m \leq a_i \leq M$

Da die Array-Einbettung erhebliche Performance-Nachteile hat, was den effizienten Speichergebrauch angeht, wird im nachfolgenden das Augenmerk auf die verkettete Node-Implementation gelegt. Die Nodes haben jeweils 3 Verweise auf Nodes: Father-, Left- und Right-Node. Die letzten beiden sind die Childs. Um die gefordete Funktionalität zu erfüllen wurden im zweiten Aufgabenteil außerdem 2 zusätzliche Integer beigefügt:
bSum, die Summe des Branch (dt. Zweig) inkl. aller Child-Nodes und **sum**, die Summe aller kleineren Elemente einschließlich des Nodes selbst. **bSum** wird beim hinzufügen eines Nodes geupdatet. Dafür ruft der hinzugefügte Node eine Methode auf, die rekursiv über die Father-Nodes bishin zum Root-Node die Werte aktualisiert. Der Code ist unter 3.2 zu sehen. **sum** wird nach dem Hinzufügen einer oder mehrerer Nodes geupdated. Die dazugehörige Methode läuft rekursiv vom Root-Node über alle Child-Nodes und aktualisiert sum. Dabei wird wie in 3.3 zu sehen bSum zur Hilfe genommen.

2.2 Algorithmus

Der eigentliche Algorithmus zum Berechnen der Summe aller Nodes zwischen m und M ist durch die in 2.1 erklärten Zusatzinformationen sehr simpel. Im Grunde genommen besteht das Ergebnis aus den Werten von den 2 nächsten Nodes zu m und M und berechnet sich wie folgt:

$$\sum_i a_i = right.sum - (left.sum - left.key)$$

Der dazugehörige Code ist unter 3.4 zu sehen. Hier wird auch eine weitere Methode ersichtlich, die zum finden der richtigen Knoten für die Berechnung anhand m und M benötigt wird. Sie sucht vom Root-Node aus Werte, die möglichst nahe dran sind und gibt die dazugehörigen Nodes zurück, der Code ist unter 3.5 zu sehen.

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:

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

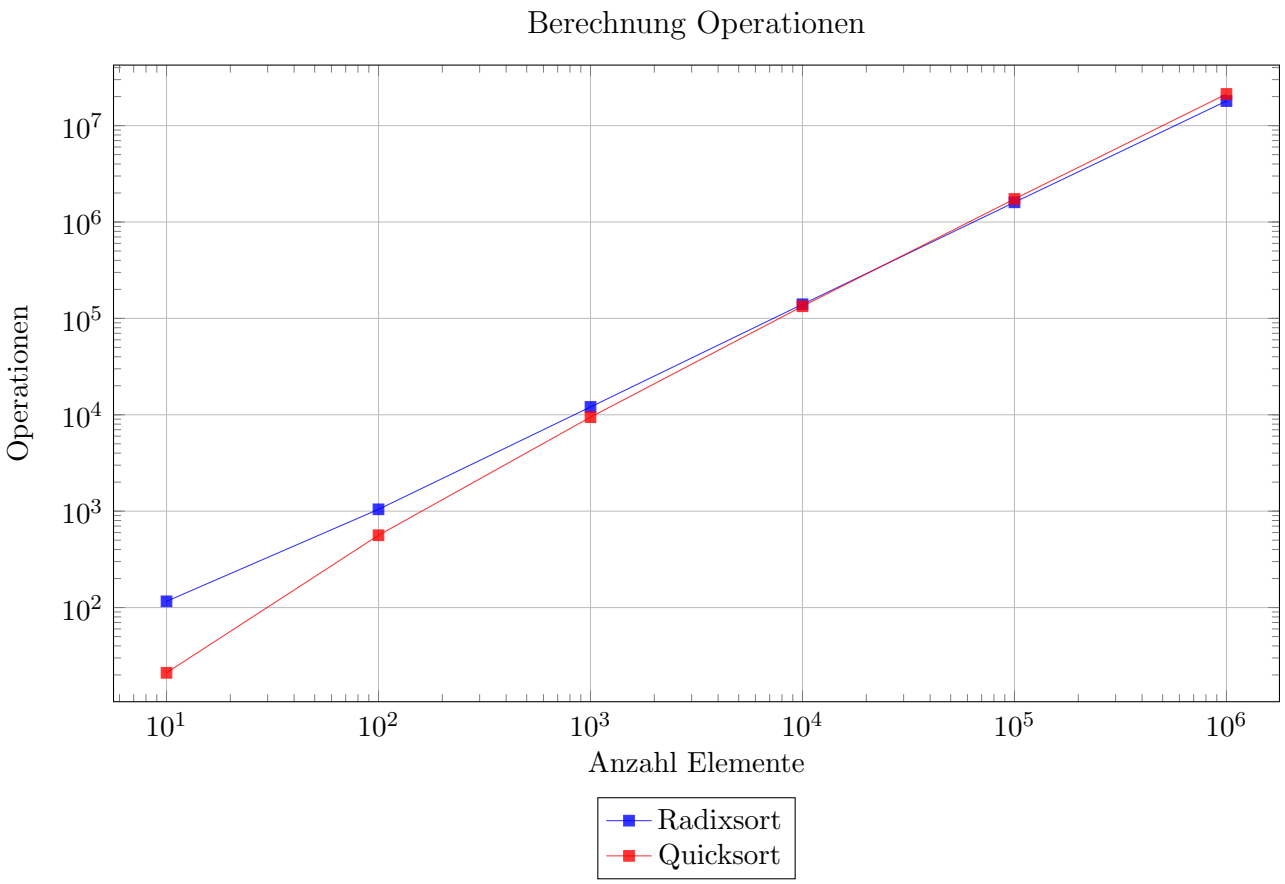

Abbildung 3.1: Quantitativer Vergleich zu Quicksort anhand Rechenoperationen

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.

```
public void updatebSum(int addition) {
    NodeNodeLinking father = getFather();

    setbSum(getbSum() + addition);
    if (father != null) {
        father.updatebSum(addition);
    }
}
```

Abbildung 3.2: Code-Ausschnitt bSum

```
public void updateSum() {
    NodeNodeLinking left = getLeft();
    NodeNodeLinking right = getRight();
    NodeNodeLinking father = getFather();
    if (left != null && father != null) {
        left.updateSum();
        setSum(left.getbSum() + getSmallerFatherSum(getKey()) + getKey());
    } else if (left == null && father != null) {
        setSum(getSmallerFatherSum(getKey()) + getKey());
    } else if (left != null && father == null) {
        left.updateSum();
        setSum(left.getbSum() + getKey());
    } else {
        setSum(getKey());
    }

    // update right branch
    if (right != null) {
        getRight().updateSum();
    }
}
```

Abbildung 3.3: Code-Ausschnitt sum

```
public long getSum(int m, int M) {
    NodeNodeLinking leftnode = root.findClosest(true, m);
    NodeNodeLinking rightnode = root.findClosest(false, M);
    long leftval = leftnode.getSum() - leftnode.getKey();
    long rightval = rightnode.getSum();
    return rightval - leftval;
}
```

Abbildung 3.4: Code-Ausschnitt Kalkulierung der Summe zwischen m und M

```
public NodeNodeLinking findClosest(Boolean highlow, int startValue) {
    NodeNodeLinking result = null;
    if (highlow) {
        // Get higher or equal to startValue's closest node
        if (getKey() >= startValue) {
            // Try and find a closer value
            if (getLeft() != null && getLeft().getKey() >= startValue) {
                // There is a closer value, recurse
                result = getLeft().findClosest(highlow, startValue);
            } else {
                // There is either no left (smaller) child or it's smaller than startValu
                result = this;
            }
        } else {
            if (getRight() != null) {
                result = getRight().findClosest(highlow, startValue);
            }
        }
    } else {
        // Get smaller or equal to startValue's closest node
        if (getKey() <= startValue) {
            // Try and find a closer value
            if (getRight() != null && getRight().getKey() <= startValue) {
                // There is a closer value, recurse
                result = getRight().findClosest(highlow, startValue);
            } else {
                // There is either no right (bigger) child or it's higher than startValue
                result = this;
            }
        } else {
            if (getLeft() != null) {
                result = getLeft().findClosest(highlow, startValue);
            }
        }
    }
    return result;
}
```

Abbildung 3.5: Code-Ausschnitt findClosest

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