Binärer Suchbaum

Finn Jannsen, Philipp Schwarz

3. 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 veranschaulicht.

2 Implementation

Der Algorithmus wurde als Klasse realisiert, die 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 verlinkten 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 das Hinzufügen mittels randomisierter Sequenzen, mit Überprüfung auf richtige Sortierung und stichprobenweise Berechnung von Summen zwischen zwei Zahlen. Alle Tests wurden erfolgreich mit unterschiedlichen Eingabewerten absolviert.

3.2 Aufwandsanalyse

Für die Aufwandsanalyse wurden die beiden Bäume mit 10^n Werten gefüllt, wobei die Reihenfolge randomisiert und $n = 1, ..., 7$ war. Für Durchschnittswerte wurde dies 10 mal durchgeführt. Beobachtet wurde hierbei die Ausführungszeit von updateSum(), zu sehen unter 3.3, die Methode hat einen Aufwand von $T(n) = \mathcal{O}(n)$. Es wird diese Methode betrachtet, da sie vor der Berechnung der Summe stattfinden muss und letzteres alleine mit den Zusatzinformationen zu schnell für aussagekräftige Messwerte stattfindet. Das gleiche galt für die Array-Einbettung, welche aufgrund mangelndem Arbeitsspeicher nur bis zu einer Größe von 10^3 funktionierte und nicht in messbare Bereiche gekommen ist. Die verlinkten Nodes brauchen lediglich einen großen Stack, um zuverlässig zu funktionieren. Die Ergebnisse der verlinkten Nodes sind unter 3.1 zu sehen.

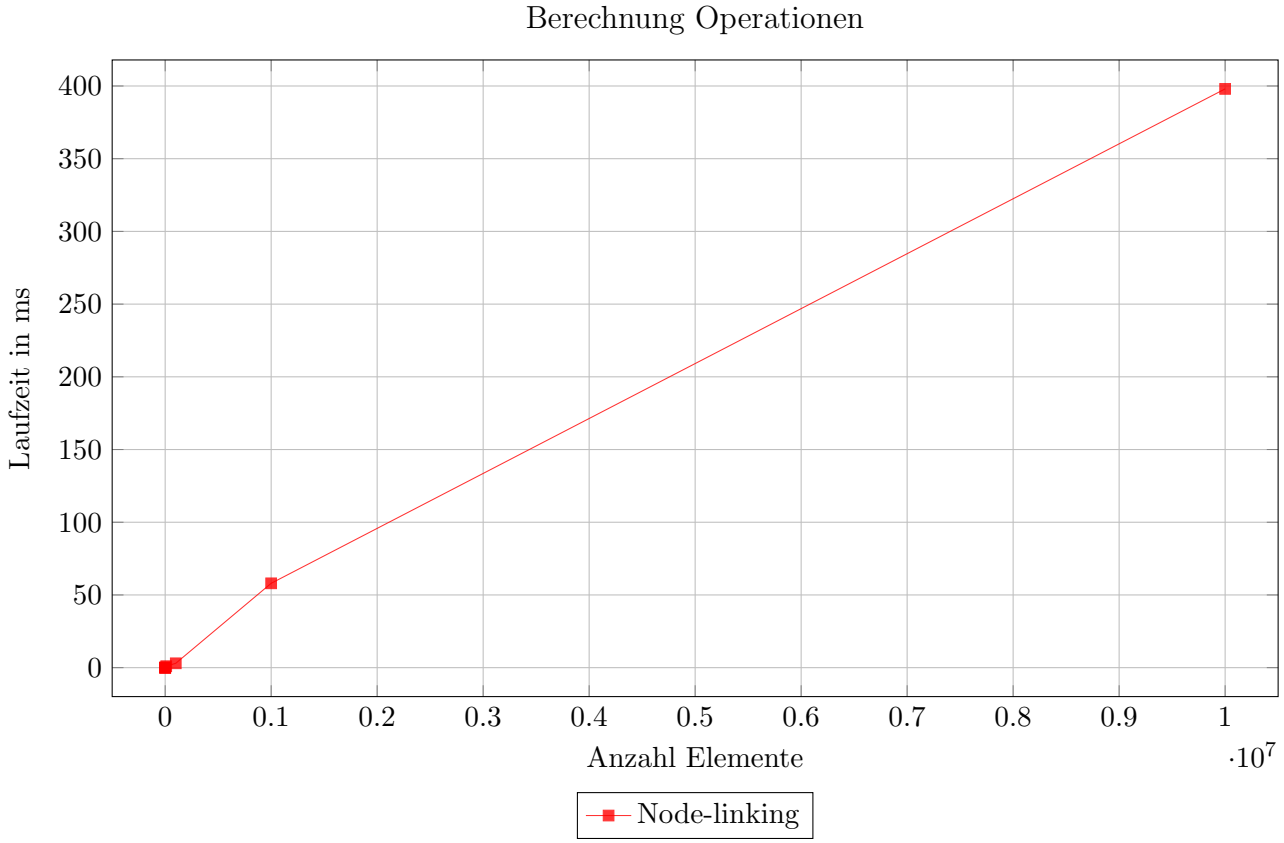

Abbildung 3.1: Quantitativer Vergleich zu Quicksort anhand Rechenoperationen

```
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