Binärer Suchbaum Teil 2

Aufgabenblatt 8

Niels Gandrass (2285656)

25. Mai 2017

Abstract

Es wird gezeigt wie die Summe der Elemente eines binären Suchbaums auf einem gegebenen Intervall möglichst effizient berechnet werden kann. Hierzu wird die Zusatzinformation SSumme der kleineren Knotenïn jedem Knoten des Baum hinterlegt. So kann durch finden der beiden Elemente, welche durch das gegebene Intervall eingeschlossen werden, direkt die Summe aller Knoten innerhalb des Intervalls bestimmt werden.

I. Vorwort

Im Rahmen des achten Aufgabenblatts des Moduls BTI3-ADP wurde ein Verfahren entwickelt, welches es ermöglicht in binären Suchbäumen schnellstmöglich die Summe der Elemente, innerhalb eines gegebenen Intervalls, zu berechnen.

Enthält ein binärer Suchbaum die Elemente $F = a_1, a_2, \ldots, a_{n-1}, a_n$, so erfolgt die Berechnung der Summe durch:

$$\sum_i a_i \quad \text{mit} \quad m \leq a_i \leq M$$

Wobei m der unteren und M der oberen Intervall-Grenze entspricht.

Zur Umsetzung wurde in dieser Ausarbeitung die Programmiersprache Java[1] genutzt. Die, in ihrer Art, durchgeführten Untersuchungen können jedoch unabhängig der Programmiersprache nachvollzogen werden.

II. Kurz und Knackig

Durch das verwendete Verfahren lies sich die Berechnung der Summe auf die Komplexität für das Finden von Knoten im binären Suchbaum reduzieren. Somit wurde $\mathcal{O}(log(n))$ erreicht.

[1] https://en.wikipedia.org/wiki/Java_(programming_language)

III. Implementiertes Verfahren

Es wurde ein Verfahren implementiert, welches in jedem Knoten des binären Suchbaums eine Zusatzinformation hinterlegt. Es wird die Summe der kleineren Knoten gespeichert. Somit kann durch finden der beiden Elemente, welche durch das gegebene Intervall eingeschlossen werden, direkt die Summe aller Knoten innerhalb des Intervalls bestimmt werden.

I. Berechnen der Zusatzinformation

Diese Summe wird mit Aufruf von `updateChildSums()` aktualisiert. Hierbei wird der Baum in Inorder-Reihenfolge traversiert und jeweils die Werte der Knoten aufaddiert und in den nächsten Knoten geschrieben.

```java
/**
 * Wrapper for updateChildSums(LinkedBinaryTreeNode, Long)
 */
public void updateChildSums() {
  updateChildSums(_rootNode, 0L);
}

/**
 * Iterates inOrder trough the tree and sums up all values that are smalle than curNode
 *
 * @param curNode Current node to process
 * @param curSum  Current sum
 *
 * @return Current sum after calculation to destruct call stack
 */
private Long updateChildSums(LinkedBinaryTreeNode curNode, Long curSum) {
  // Cancelation condition
  if (curNode == null) {
    return 0L;
  }

  // Go down do smallest (leftest) node
  if (curNode.getLeftChild() != null) {
    curSum = updateChildSums(curNode.getLeftChild(), curSum);
  }

  // Add curNodes value to sum
  curSum += curNode.getValue();
  curNode.setLowerValueSum(curSum);
  _counterUpdateChildSum++;

  // Go down to biggest (rightest) node
  if (curNode.getRightChild() != null) {
    curSum = updateChildSums(curNode.getRightChild(), curSum);
  }

  return curSum;
}
```

II. Berechnen der Knotensumme

Zur Berechnung der Summe der Knoten des binären Suchbaums auf deinem gegebenen Intervall müssen nun nur noch die umschließenden Knoten gefunden werden. Somit wird für den Knoten, welcher die obere Grenze bildet der nächst größere Knoten gesucht. Für den Knoten, welcher die untere Grenze bildet, der nächst kleinere.

Abschließend werden von Unterknotensumme des oberen Kontens die Unterknotensumme des unteren Knotens und gegebenenfalls ein Offset abgezogen. Dieser Offset berücksichtigt den Sonderfall, dass beispielsweise die Grenze genau auf einem vorhandenen Knoten liegt.

```java
/**
 * Determines the sum of node values between the given bounds
 *
 * @param lowerBound Lower bound
 * @param upperBound Upper bound
 *
 * @return Sum of the node values in between these bounds
 */
public Long limitSum(long lowerBound, long upperBound) {
  // Preconditions
  if (lowerBound > upperBound)
  throw new IllegalArgumentException();

  // Find nodes
  LinkedBinaryTreeNode upperNode = findClosestBiggerNode(upperBound);
  LinkedBinaryTreeNode lowerNode = findClosestSmallerNode(lowerBound);

  // Assure that nodes got found otherwise select smallest/biggest node
  if (upperNode == MIN_VALUE_NODE) {
    upperNode = getBiggestNode();
    upperBound = upperNode.getValue();
    _counterLimitSum++;
  }
  if (lowerNode == MAX_VALUE_NODE) {
    lowerNode = getSmallestNode();
    lowerBound = lowerNode.getValue();
    _counterLimitSum++;
  }

  // Calculate offset for cases where nodes match directly
  long offset = 0;
  if (lowerNode.getValue() == lowerBound)
  offset += lowerNode.getValue();
  if (upperNode.getValue() > upperBound)
  offset -= upperNode.getValue();
  _counterLimitSum+=2;

  // Calculate difference
  _counterLimitSum++;
  return findClosestBiggerNode(upperBound).getLowerValueSum() -
   ↪  findClosestSmallerNode(lowerBound).getLowerValueSum() + offset;
}
```

III. UML-Klassendiagram

Folgendes UML-Klassendiagram zeigt die komplette Implementierung.

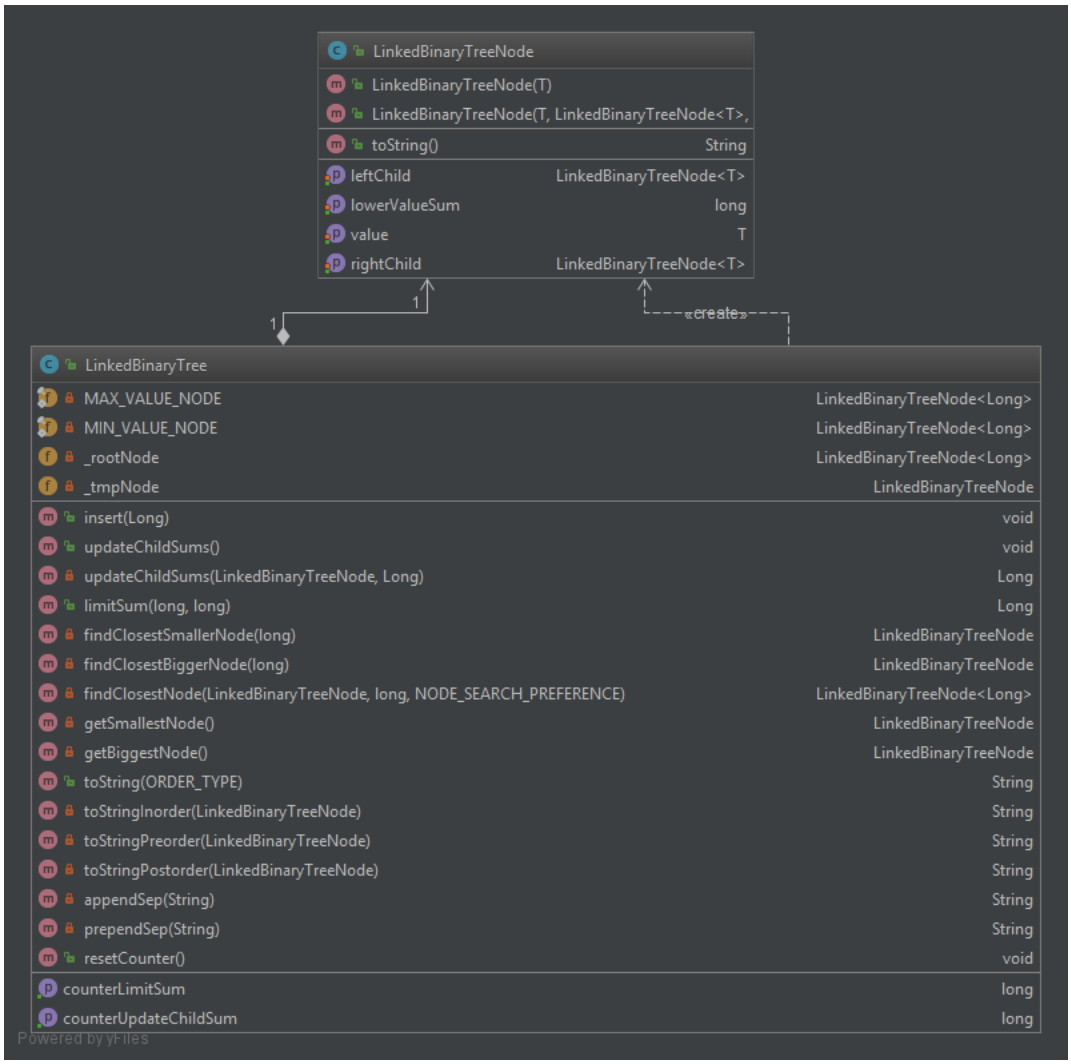

Abbildung 1: UML-Klassendiagramm

IV. Komplexitätsuntersuchung

Es wurde eine empirische Komplexitätsuntersuchung durchgeführt, bei welcher binäre Suchbäume aus N Elementen mit $N = 10^k$ mit $k = 1, \ldots, 6$ erzeugt wurden. Für jede Größe N wurden 100 Durchläufe durchgeführt. Es wurden die benötigten Instruktionen der Methoden `updateChildSums()` sowie `limitSum()` aufgezeichnet.

Hierbei ergaben sich folgende Ergebnisse:

I. Ergebnisse `updateChildSums()`

Es zeigt sich, dass die benötigten Instruktionen für die Berechnung der Anzahl der im Baum vorhandenen Elemente entspricht. Die Diskrepanz zur Anzahl der Eingangselemente N ergibt sich dadurch, dass durch die zufällige Generierung der Zahlen auch doppelte Elemente vorkommen, welche aber nur jeweils einmal in den Baum eingefügt werden. Somit ergibt sich für `updateChildSums()` eine Komplexität von $\mathcal{O}(N)$.

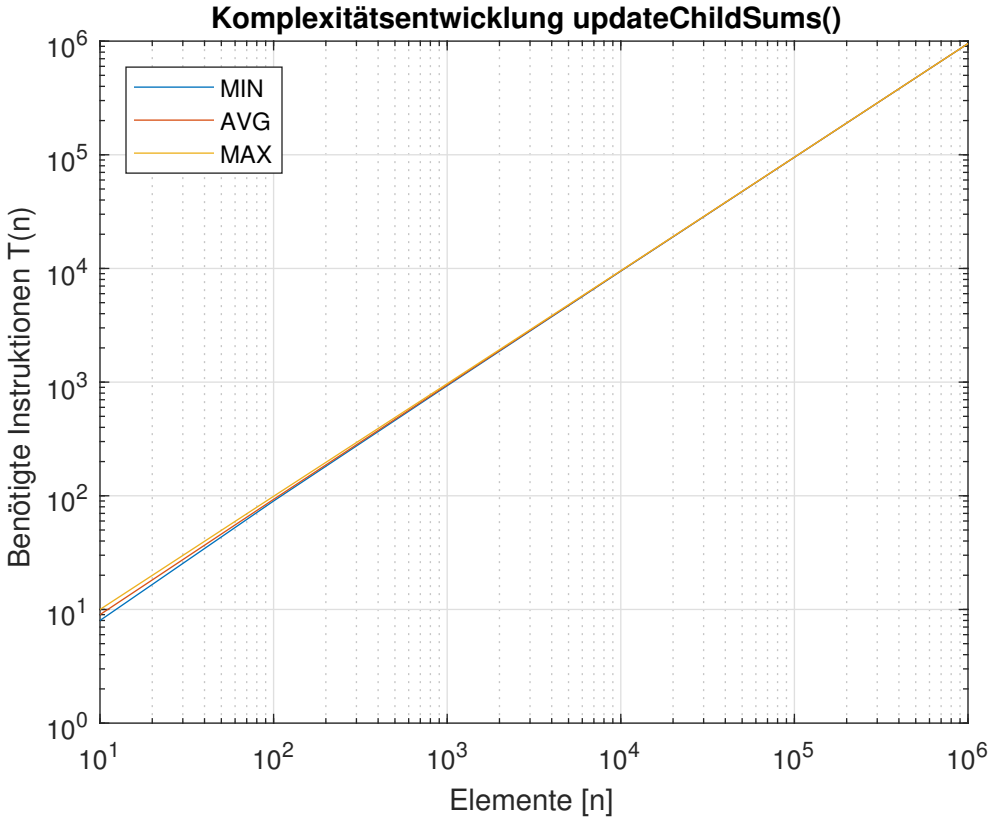

Abbildung 2: Instruktionen `updateChildSums()`

N	MIN	AVG	MAX
10^1	8	9	10
10^2	90	93	99
10^3	930	945	969
10^4	9469	9514	9555
10^5	95001	95135	95284
10^6	951149	951694	952130

Tabelle 1: Instruktionen `updateChildSums()`

II. Ergebnisse `limitSum()`

Zum Bestimmen der Summe müssen lediglich die zwei Knoten gefunden werden, welche das Intervall einschließen. Somit wird die Komplexität auf das Suchen von Knoten in einem binären Suchbaum reduziert. Hiermit ergibt sich für `limitSum()` eine Komplexität von $\mathcal{O}(log(n))$.

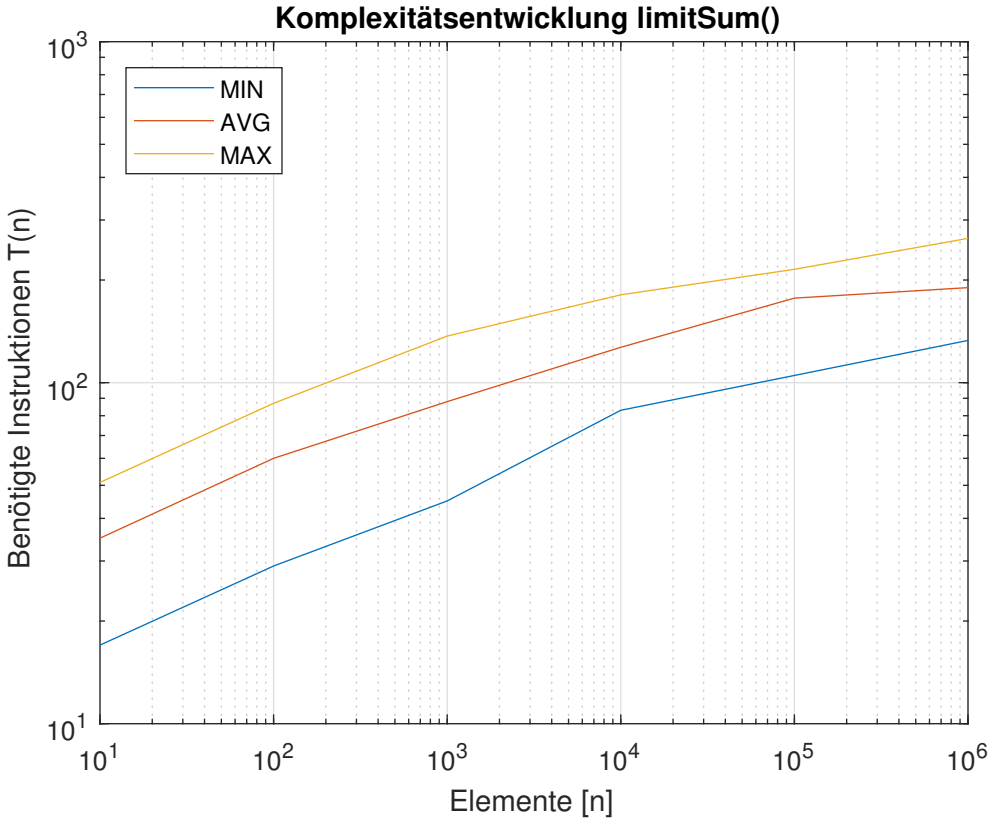

Abbildung 3: Instruktionen `limitSum()`

N	MIN	AVG	MAX
10^1	17	35	51
10^2	29	60	87
10^3	45	88	137
10^4	83	127	181
10^5	105	177	215
10^6	133	190	265

Tabelle 2: Instruktionen `limitSum()`

III. Anmerkungen

Die zu analysierenden binären Suchbäume wurden wie folgt erzeugt:

```
private static LinkedBinaryTree generateTree(long size) {
  LinkedBinaryTree tree = new LinkedBinaryTree();

  // Fill random values
  for (long i = 0; i < size; i++) {
    tree.insert((long) (Math.random()*size*10));
  }

  return tree;
}
```