Ludwig-Maximilians-Universität München
Institut für Informatik
PD Dr. Matthias Renz
Dr. Erich Schubert, Daniel Kaltenthaler

München, 06.05.2015

Algorithmen und Datenstrukturen

SS 2015

Übungsblatt 4: O-Notation und Bäume

Besprechung: 18.05.–22.05.

Abgabe aller mit **Punkten** versehenen Aufgaben bis 17.05.2015 13:25

Aufgabe 4-1 O-Notation III — 5 Punkte

Geben Sie an, für welche $x \in \mathbb{R}^+$ die folgende Gleichung gilt:

$$\sum_{i=1}^{n} x^i = O(n)$$

Beweisen Sie Ihr Ergebnis.

Aufgabe 4-2 Maximum-Subarray — Tutorenaufgabe

Das Maximum-Subarray Problem besteht darin, in einem Array aus ganzen Zahlen der Länge n die **nicht leere** Teilfolge von Zahlen zu finden, deren Summe maximal ist.

Beispiel: Für die Eingabefolge $[x_0, ..., x_9]$

$$[12, -32, 38, 22, -27, 32, 17, -13, -10, 10]$$

ist die maximale Summe einer Teilfolge 82 für die Teilfolge $[x_2, ..., x_6]$.

Ein naiver Algorithmus zur Lösung dieses Problems ist der folgende:
Berechne für jedes mögliche Start- und Endelement die dazwischenliegende Summe und merke Dir die größte Summe und die entsprechenden Start- und Endelemente. Als Java-Algorithmus ausgedrückt:

```
public void naiveSolution(Integer[] x) {
  int maxSum = 0;
  for (int u = 0; u < x.length; u++) {
    for (int o = u; o < x.length; o++) {
      // bestimme die Summe der Elemente der Teilfolge von [x_u, ..., x_o]
      int sum = 0;
      for (int i = u; i <= o; i++) {
        sum = sum + x[i];
      }
      // vergleiche die gefundene Summe mit der bis jetzt maximalen
      maxSum = Math.max(maxSum, sum);
    }
  }
}
```

(a) Schätzen Sie die Zeitkomplexität des obigen Algorithmus ab und begründen Sie Ihre Angabe kurz.

(b) Geben Sie die Idee für einen Algorithmus an, der obiges Problem in linearer Komplexität löst und demonstrieren Sie diese am obigen Zahlenbeispiel.

(c) Implementieren Sie Ihre Lösungsidee aus Aufgabe (b) in Java. Das Programm soll dabei die Summe, das erste und das letzte Element der maximalen Teilfolge als Ergebnis ausgeben. Die Eingabe des Inputarrays als Konstante ist ausreichend.

Aufgabe 4-3 Binärbaum Tutorenaufgabe

In der Vorlesung wurde die Datenstruktur eines Baumes besprochen.

Wir implementieren nun die einfachste Version eines binären Baumes.

- Ergänzen Sie die Methode `hoehe()`, mit der die Höhe des Baumes berechnet werden kann.
  Ein leerer Baum soll dabei die Höhe 0 haben, ein Baum der nur aus der Wurzel besteht die Höhe 1, usw.

```
/** Binaerer Baum fuer Daten vom Typ INHALT */
public class BinaerBaum<INHALT> {
  /** Wurzel des Baumes */
  Knoten<INHALT> wurzel;

  /** Konstruktor fuer den Baum */
  public BinaerBaum(Knoten<INHALT> wurzel) {
    this.wurzel = wurzel;
  };

  /** Hoehe berechnen */
  public int hoehe() {
    // TODO: Implementieren
  }

  /** Klasse, die einen Knoten des Baumes speichert */
  protected static class Knoten<INHALT> {
    /** Gespeichterte Daten */
    protected INHALT daten;

    /** Linker und rechter Sohn */
    protected Knoten<INHALT> links, rechts;

    /** Konstruktor fuer Knoten */
    public Knoten(INHALT daten, Knoten<INHALT> links, Knoten<INHALT> rechts) {
      this.daten = daten;
      this.links = links;
      this.rechts = rechts;
    }
  }
}
```

- Welche Komplexität hat die eben implementierte Methode `hoehe()`?
- Testen Sie auch diesmal Ihr Programm, mindestens mit folgendem Programm:

```
/** Binaerer Baum fuer Daten vom Typ INHALT */
public class BinaerBaum<INHALT> {
  // Wie zuvor

  /** Methode zum Testen der Implementierung - besser: JUnit Test */
  public static void main(String[] args) {
    Knoten<Integer> links = new Knoten<Integer>(1, null,
        new Knoten<Integer>(2, null, null));
    Knoten<Integer> rechts = new Knoten<Integer>(5,
        new Knoten<Integer>(4, null, null),
        new Knoten<Integer>(6, null,
          new Knoten<Integer>(7, null, null)));
    Knoten<Integer> wurzel = new Knoten<Integer>(3, links, rechts);
    BinaerBaum<Integer> baum = new BinaerBaum<Integer>(wurzel);
    System.out.println("Hat der Baum die Hoehe 4? " + baum.hoehe());
    BinaerBaum<Integer> leer = new BinaerBaum<Integer>(null);
    System.out.println("Hat der leere Baum die Hoehe 0? " + leer.hoehe());
  }
}
```

- Eine verkettete Liste kann als degenerierter Baum (mit nur einem Ast) verstanden werden. Vergleichen Sie den Knoten des Binärbaumes mit dem Knoten einer verketteten Liste. Warum sind Methoden wie `prepend` und `append` bei einem Binärbaum nicht sinnvoll?
- Bei verketteten Listen unterscheidet man zwischen einfach- und doppelt-verketteten Listen. Wie sieht das äquivalent zur Doppeltverkettung bei einem Baum aus?

Aufgabe 4-4 Binärbaum II 4 Punkte

Wir wollen den Binärbaum nun um zusätzliche Funktionalität ergänzen.

- Ergänzen Sie die Methode `anzahlKnoten()`, die berechnet aus wie vielen `Knoten<INHALT>` der Baum besteht. Testen Sie ihre Implementierung!

```
/** Binaerer Baum fuer Daten vom Typ INHALT */
public class BinaerBaum<INHALT> {
  // ... Wie zuvor ...

  /** Anzahl der Knoten zaehlen */
  public int anzahlKnoten() {
    // TODO: Implementieren
  }

  /** Methode zum Testen der Implementierung - besser: JUnit Test */
  public static void main(String[] args) {
    // ... Wie zuvor ...
    System.out.println("Hat der Baum 7 Knoten? " + baum.anzahlKnoten());
    System.out.println("Hat der leere Baum 0 Knoten? " + leer.anzahlKnoten());
  }
}
```

- Welche Komplexität hat die eben implementierte Methode `anzahlKnoten()`?

Aufgabe 4-5 Einfache Sortierverfahren **Tutorenaufgabe**

Wir wollen die Sortierverfahren aus der Vorlesung selbst implementieren. Wir vereinfachen die Algorithmen aber derart, dass wir ein `int[]` sortieren statt einem `Entry[]`.

- Implementieren Sie eine Hilfsfunktion um zu testen, ob ein `int[]` bereits sortiert ist, sowie zum vertauschen von zwei Elementen:

```
public class SortUtil {
  /** Testet, ob Daten sortiert sind. */
  public static boolean isSorted(int[] daten) {
    int prev = daten[0];
    for (int i = 1; i < daten.length; i++) {
      if (daten[i] < prev) {
        return false;
      }
      prev = daten[i];
    }
    return true;
  }

  /** Vertauscht zwei Elemente */
  public static void vertausche(int[] daten, int x, int y) {
    final int tmp = daten[x];
    daten[x] = daten[y];
    daten[y] = tmp;
  }
}
```

- Als Vorlage verwenden Sie folgende Klasse und testen Sie Ihr Programm:

```
public class AbstractSort {
  /** Daten sortieren */
  public static void sort(int[] daten) {
    // TODO: Implementieren
  }

  /** Funktionstest - eleganter waere: JUnit-Test */
  public static void main(String[] args) {
    int[] test = new int[]{23, 12, 45, 31, 56, 71, 07};
    sort(test); // Hier wird sortiert
    System.out.print("Ergebnis: ");
    for (int val : test) {
      System.out.print(" " + val);
    }
    System.out.println();
    System.out.println("Sortiert? " + SortUtil.isSorted(test));
  }
}
```

- Implementieren Sie die vier Verfahren (Pseudocode siehe Vorlesung)
  `CountingSort`, `SelectionSort`, `BubbleSort`, `InsertionSort`.

- Vergleichen Sie diese Sortieralgorithmen.
  Welcher Algorithmus hat einen deutlichen Nachteil, welcher ist am elegantesten?