TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen SS 2016
Grundlagen: Algorithmen und Datenstrukturen **Übungsblatt 1**
Prof. Dr. Helmut Seidl, J. Kranz, A. Reuss, R. Vogler 17.04.2016

Abgabe: 24.04.2016 (bis 23:59 Uhr)

Aufgabe 1.1 (P) Tiefe Bäume

Gegeben sei folgende induktive Definition der Datenstruktur *Baum*: Ein Baum ist

- entweder ein als *Blatt* bezeichnetes Objekt
- oder ein innerer *Knoten* zusammen mit einer Liste von (Unter-)Bäumen.

Sei $k \geq 1$. Ein Baum heißt k-verzweigt, wenn

- er ein Blatt ist oder
- er k viele k-verzweigte Unterbäume hat.

In k-verzweigten Bäumen hat also jeder innere Knoten gleich viele *Kinder*.

Wir bezeichnen einen Baum als *vollständig*, wenn

- er ein Blatt ist oder
- die Unterbäume alle gleich und ihrerseits vollständig sind.

Die *Tiefe* eines Baumes definieren wir als

- 1, wenn er ein Blatt ist, und
- 1+ die maximale Tiefe eines Unterbaumes sonst.

Die Anzahl Knoten in einem Baum ist

- 1, wenn er ein Blatt ist, und
- 1+ die Summe der Knoten der Unterbäume sonst.

Beweisen Sie folgende Behauptung mit Hilfe von Induktion **nach der Anzahl Knoten** n: Die Tiefe eines vollständigen 2-verzweigten Baumes mit n Knoten ist höchstens $log_2(n+1)$.

Hinweis zur Definition der Vollständigkeit: Wir bezeichnen zwei Bäume als *gleich*, wenn

- beide Blätter sind oder
- beide die gleichen Unterbäume haben (in der selben Reihenfolge).

Lösungsvorschlag 1.1

Induktionsanfang, -voraussetzung und -schritt werden mit IA, IV, IS bezeichnet.

IA: $n = 1$: Laut induktiver Definition enthält jeder Unterbaum mindestens einen (weiteren) Knoten. $n = 1$ bedeutet also, dass der Baum ein Blatt ist. Die Tiefe ist 1. Zu zeigen ist also: $1 \leq log_2(n+1)$ für $n = 1$. Es gilt $log_2 2 = 1 \geq 1$.

IV: Für alle $k \leq n$ gilt, dass die Tiefe eines vollständigen 2-verzweigten Baumes mit k Knoten höchstens $log_2(k+1)$ ist.

IS: Betrachte Bäume mit $n+1$ Knoten. Fallunterscheidung:

a) Es existiert ein $k \in \{1, 2, 3, \ldots\}$ mit $n + 1 = 2^k - 1$. Weil der Baum $n + 1$, also mindestens zwei Knoten besitzt, kann er kein Blatt sein. Es gibt also 2 Unterbäume, die jeweils höchstens n Knoten besitzen, da ein innerer Knoten schon *vergeben* ist.

Laut IV ist die Tiefe der Unterbäume jeweils höchstens $log_2(n+1) = log_2(2^k - 1) < k$. Da die Tiefe kleiner als k und eine ganze Zahl ist, kann sie höchstens $k-1$ sein. Damit ist die Tiefe des gesamten Baumes höchstens $k = log_2(n+1+1)$, was zu zeigen war.

b) Es existiert kein $k \in \{1, 2, 3, \ldots\}$ mit $n + 1 = 2^k - 1$. Dann gibt es auch keinen vollständigen 2-verzweigten Baum mit $n+1$ Knoten, und es ist nichts zu zeigen.
Dass es zu allen vollständigen 2-verzweigten Bäumen ein k gibt, so dass die Zahl der Knoten $2^k - 1$ ist, kann man mittels Induktion nach der Tiefe t des Baumes zeigen. Für $t = 1$ gilt $k = 1$, für einen vollständigen 2-verzweigten Baum der Tiefe $t+1$ gibt es dann nach IV ein entsprechendes k für die Unterbäume der Tiefe t, und weil die Unterbäume gemäß Definition gleich sind, ist das k ebenfalls gleich, d.h. die Gesamtzahl der Knoten ist $1 + 2(2^k - 1) = 1 + 2^{k+1} - 2 = 2^{k+1} - 1$.

Aufgabe 1.2 (P) Bubblesort mit Listen

Gegeben sei folgender Java-Code, der eine einfach verkettete Liste implementiert:

```
public class List {
  private static class Node {
    private int data;

    public int getData () {
      return data;
    }

    public void setData (int data) {
      this.data = data;
    }

    private Node next;

    public Node getNext () {
      return next;
    }

    public Node (int data, Node next) {
      this.data = data;
      this.next = next;
    }
  }

  private Node head;

  private int size;

  public int getSize () {
    return size;
  }

  public List () {
  }

  void prepend (int data) {
    head = new Node(data, head);
    size++;
  }

  int get (int index) {
    Node it = head;
    while (index != 0) {
      index--;
```

```
      it = it.getNext();
      if (it == null)
        throw new RuntimeException("Out of bounds");
    }
    return it.getData();
  }

  void swap (int indexFirst, int indexSecond) {
    if (indexFirst > indexSecond) {
      swap(indexSecond, indexFirst);
      return;
    }
    int distance = indexSecond - indexFirst;
    if (head == null)
      throw new RuntimeException("Out of bounds");
    Node it_first = head;
    while (indexFirst != 0) {
      indexFirst--;
      it_first = it_first.getNext();
      if (it_first == null)
        throw new RuntimeException("Out of bounds");
    }
    Node it_second = it_first;
    while (distance != 0) {
      distance--;
      it_second = it_second.getNext();
      if (it_second == null)
        throw new RuntimeException("Out of bounds");
    }
    int temp = it_second.getData();
    it_second.setData(it_first.getData());
    it_first.setData(temp);
  }

  @Override public String toString () {
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    boolean first = true;
    Node it = head;
    while (it != null) {
      if(first)
        first = false;
      else
        sb.append(", ");
      sb.append(it.getData());
      it = it.getNext();
    }
    sb.append("]");
    return sb.toString();
  }
}
```

a) Betrachten Sie die Implementierung und beantworten Sie dabei folgende Fragen:

- Was hat es mit der Schachtelung der Klassen auf sich?
- Was bedeutet es, dass die innere Klasse statisch ist? Wieso ist dies hier sinnvoll?
- Wozu wurde der Wrapper `List` implementiert, statt den Benutzer direkt auf Knoten arbeiten zu lassen? Was hat dies mit *abstrakten Datentypen* zu tun?
- Welchen Zweck erfüllt `throw`? Wieso wird eine `RuntimeException` statt einer normalen `Exception` geworfen? Wo liegt der Unterschied?
- Welche der Operationen sind langsam, welche schnell? Wieso?

- Die `swap`-Methode ruft sich selbst auf. Wie nennt man Methoden, die sich so verhalten? Welche Motivationen gibt es, derart zu programmieren?
- Wieso wird in `toString()` ein `StringBuilder` verwendet? Welche Operation implementiert dieser effizient?

b) Die Liste soll nun genutzt werden, um Daten sortiert zu speichern. Um die Liste zu sortieren, wird folgende Methode verwendet, die den *Bubblesort*-Algorithmus implementiert:

```
void bubblesort(List l) {
  for(int i = 0; i < l.getSize(); i++)
    for(int j = 1; j < l.getSize() - i; j++)
      if(l.get(j - 1) > l.get(j))
        l.swap(j - 1, j);
}
```

Beantworten Sie folgende Fragen:

- Die Methode hat keinen Rückgabewert; funktioniert sie dennoch? Wenn ja, wieso? Rufen Sie sich allgemein das Konzept von *Referenzen* ins Gedächtnis.
- Wie funktioniert der Algorithmus, warum liefert er stets ein sortiertes Ergebnis?
- Wie oft *ungefähr* wird eine Liste der Größe n durchlaufen? Beantworten Sie diese Frage zunächst, ohne dabei an die Laufzeit der aufgerufenen Listenmethoden zu denken.
- Eignet sich unsere Implementierung einer Liste hier besonders gut oder besonders schlecht? Ziehen Sie nun die Implementierung der Listenmethoden in Ihre Laufzeitüberlegung mit ein. Zu welch verheerendem Ergebnis kommen Sie?

Lösungsvorschlag 1.2

a)
- Die Schachtelung der Klasse `Node` macht hier Sinn, da diese nur als *Hilfsklasse* der Klasse `List` verwendet wird. Die Klasse selbst ist daher auch `private`, also von außen nicht sichtbar. Besonders praktisch an eingebetteten Klassen ist außerdem, dass sie auf Daten der umgebenden Klasse direkt zugreifen können; hiervon wird allerdings im Beispiel kein Gebrauch gemacht.
- Das Schlüsselwort `static` hat hier im Prinzip die gleiche Bedeutung wie bei anderen Membern auch. Das heißt, dass es die Klassendefinition nur einmal global und nicht speziell für jedes Objekt gibt. Dies macht hier Sinn, da die eingebettete Klasse nicht auf nicht-statische Member der umgebenden Klasse zugreifen möchte.
- Die Wrapperklasse `List` wurde implementiert, um dem Benutzer eine *sichere* Schnittstelle der Liste anbieten zu können, die Details der Implementierung verbirgt. Würde man dem Benutzer direkt die Referenz auf den Kopfknoten geben, so hätte dieser u.U. die Möglichkeit, die Liste zu zerstören. Außerdem würde er durch die Handhabung der Knoten den Aufbau der Liste für seine Implementierung als gegeben ansehen, welches spätere Änderungen an der Listenimplementierung erschweren würde. Die Klasse `List` bietet hier also die Schnittstelle für den abstrakten Datentypen einer Liste an, während die Klasse `Node` Teil der konkreten Implementierung der Datenstruktur ist.
- Das Schlüsselwort `throw` wirft eine Ausnahme und bricht somit die Operation mit einem Fehler ab. Beim Werfen von Ausnahmen muss man in Java beachten, dass mögliche Ausnahmen im Kopf der Methode explizit genannt werden müssen, es sei

denn diese leiten von der Klasse `RuntimeException` ab. Klare Regeln, wann eine `RuntimeException` zu bevorzugen ist, gibt es nicht; da der Ersteller der Aufgabe dem expliziten Nennen von Ausnahmen skeptisch gegenübersteht, wurde hier die einfacher nutzbare Variante einer `RuntimeException` gewählt.

- Während die `prepend`-Methode schnell ist, sind alle übrigen Operationen aufwendig. Dies kommt daher, dass sie alle im schlimmsten Fall die ganze Liste durchlaufen müssen, also eine lineare Laufzeit in der Länge der Liste aufweisen. Bei der `prepend`-Metode dagegen müssen nur Referenzen verändert werden; dies geschieht in konstanter Zeit.
- Man nennt derartige Methoden rekursiv. Rekursive Programmierung kann einfacher verständlich als iterative Lösungen sein, insbesondere da mathematische Zusammenhänge oft rekursiv definiert werden und sich daher sehr gut durch rekursive Programme abbilden lassen.
- Ein `StringBuilder` ist praktisch, wenn man große Strings aus vielen Einzelteilen aufbauen möchte. Kettet man stattdessen die Strings mit dem +-Operator aneinander, so kann sich das schnell negativ auf die Laufzeit des Programms auswirken, da in diesem Fall bei jeder Anhängung der komplette String kopiert werden muss.

b)
- Die Methode funktioniert, da Objekte in Java generell durch Referenzen repräsentiert werden. Das heißt, dass die Variable, die die Liste identifiziert, bei dem Aufruf zwar kopiert wird, das allerdings nur dazu führt, dass die Referenz, und nicht das eigentliche Objekt kopiert wird. Daher verändert die Methode `bubblesort` anschließend dasselbe Listeobjekt, welches in der aufrufenden Methode angelegt wurde. Da also das referenzierte, übergebene Objekt verändert wird, muss es nicht explizit zurückgegeben werden. Wichtig ist zu verstehen, dass dies z.B. bei einer Integer-Variablen nicht funktionieren würde, da Integer ein Basistyp ist und daher Integer-Variablen den Zahlenwert direkt und keine Referenz auf einen solchen beinhalten.
- Der Programmcode implementiert den bekannten *Bubblesort*-Algorithmus. Für eine gute Erklärung kann z.B. die Wikipedia-Seite[1] zu Rate gezogen werden.
- Da der Algorithmus jeweils pro Durchlauf durch die Liste ein Element an die richtige Position schiebt, verhält sich die Laufzeit quadratisch zur Anzahl der Elemente in der Liste.
- Die Verwendung einer Liste, insbesondere einer Liste mit den hier vorgestellten Operationen, eignet sich besonders schlecht für den *Bubblesort*-Algorithmus, da die verwendeten Methoden `get` und `swap` jeweils wieder im schlimmsten Fall die gesamte Liste durchlaufen müssen. Das führt dazu, dass die Laufzeit nicht nur quadratisch, sondern sogar kubisch in der Länge der zu sortierenden Liste wird.

[1] `(http://de.wikipedia.org/wiki/Bubblesort`