Bonuspunkte zur Klausur "Algorithmen und Datenstrukturen für Medizintechnik"
Sommersemester 2021
René Raab, Robert Richer

Lehrstuhl für Maschinelles Lernen und Datenanalytik
(`http://www.mad.tf.fau.de`)

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT
MACHINE LEARNING & DATA ANALYTICS

Aufgabenblatt BONUS vom 27.06.2021, Abgabe am 18.07.2021, 22:00 Uhr

Hinweis: Dieses Blatt zählt **ausschließlich** für das Sammeln von Bonuspunkten für die Klausur im Sommersemester 2021. Punkte für den Übungsschein können in diesem Blatt **nicht** gesammelt werden! Für das Sammeln von Bonuspunkten ist eine **Einzelabgabe** erforderlich!
In diesem Semester gibt es nur ein Bonusblatt. Es können maximal 45 Bonuspunkte erreicht werden.

Aufgabe BONUS.1: n-ärer Baum

21 Punkte

Bäume, Breiten- und Tiefensuche, Rekursion

Aus der Vorlesung kennen Sie Binärbäume, bei denen jeder innere Knoten (= Knoten mit mindestens einem Kindknoten) im Baum maximal zwei Nachfolger hat. Knoten ohne Nachfolger heißen Blattknoten. In dieser Aufgabe soll eine Baum-Klasse implementiert werden, in der jeder innere Knoten bis zu n Nachfolger haben kann (siehe Abbildung 1). Jeder Knoten speichert eine Ganzzahl.

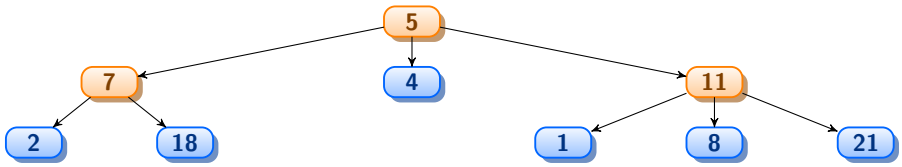

Abbildung 1: Beispiel für einen ternären Baum ($n = 3$).

1. Erstellen Sie ein neues Eclipse-Projekt `Bonus-01-NaryTree`.
2. Erstellen Sie eine Klasse `NaryTree`, die das Interface `NaryTreeInterface` implementiert. Das Interface finden Sie bei den Materialien zu diesem Blatt.
3. Dem einzigen Konstruktor `NaryTree(int capacity, int value)` wird die maximale Kapazität jedes Knotens `capacity` (n) sowie der zu speichernde Wert für den aktuellen Knoten `value` übergeben. Jeder Knoten soll seine Kinder in einer `java.util.ArrayList` verwalten.

   Achtung: Eine `ArrayList` hat keine maximale Kapazität, sie kann dynamisch wachsen. Sie müssen selbst sicherstellen, dass jeder Knoten nicht mehr als n Kinder haben kann.
4. Implementieren Sie alle im Interface vorgegebenen Methoden anhand der JavaDoc-Kommentare. Beachten Sie folgende Richtlinien:
   - Die Höhe eines Teilbaums ist definiert als längster Weg von der Wurzel zu einem Blattknoten. In Abbildung 1 hat der Teilbaum ab Knoten 5 die Höhe 2, ab Knoten 7 die Höhe 1 und ab Knoten 2 die Höhe 0.
   - Wenn in `addChild` kein Kind mehr hinzugefügt werden kann, weil die Kapazität erreicht ist, soll eine `TreeAtCapacityException` geworfen werden. Diese gibt es noch nicht in der Java-API, Sie müssen sie selbst umsetzen.

- `containsDFS` soll in Tiefensuch-Reihenfolge (***d**epth-**f**irst **s**earch*) nach der übergebenen Zahl suchen, `containsBFS` in Breitensuch-Reihenfolge (***b**readth-**f**irst **s**earch*). Für `containsBFS` könnten das Interface `Queue<E>` und die Klasse `LinkedList<E>` (die `Queue<E>` implementiert) aus der Java-API hilfreich sein.
- Implementieren Sie `treeHeight`, `numInnerNodes` und `numLeaves` **rekursiv**!

5. Testen Sie in der `main`-Methode alle implementierten Funktionen mit ausreichend komplexen Beispielbäumen, um alle Spezialfälle abzudecken.

   Wir werden diese Tests bewerten! Achten Sie daher darauf, dass Sie wirklich alle Methoden und mögliche Spezialfälle berücksichtigen.

6. Geben Sie die Dateien `NaryTree.java` und `TreeAtCapacityException` im EST ab.

Aufgabe BONUS.2: Huffman-Kodierung

24 Punkte

Prioritätswarteschlangen, Datenkompression

In dieser Aufgabe wollen wir uns mit der Komprimierung von Textdateien beschäftigen. In einer normalen Textdatei ist jedes Zeichen z. B. als *8-Bit-ASCII-Wert* kodiert. Unser Ansatz wird es sein, eine neue Kodierung variabler Länge zu finden, so dass besonders häufig vorkommende Zeichen mit nur wenigen Bits und selten vorkommende Zeichen mit mehr Bits kodiert werden.

Die Huffman-Kodierung ist ein Algorithmus, der sogar die *optimale Kodierung* im Sinne der maximalen Speicherplatzeinsparung berechnen kann. Abbildung 2 zeigt ein Beispiel. Dabei werden immer die beiden Zeichen mit der geringsten Frequenz (Auftretenshäufigkeit) zu einem kombinierten Zeichen zusammengefügt, dessen Frequenz die Summe der Frequenzen der ursprünglichen Zeichen ist.

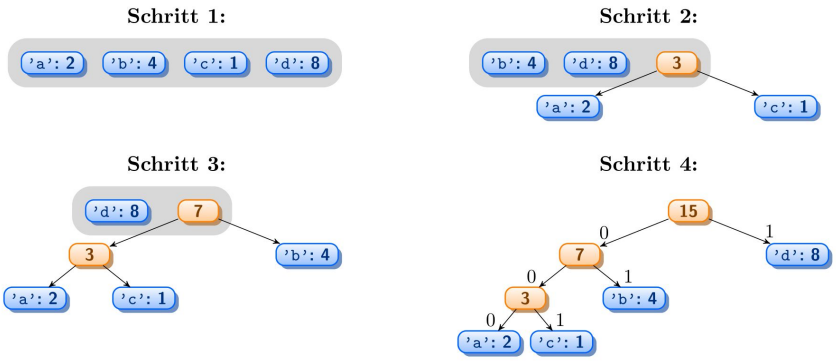

Abbildung 2: Huffman-Kodierung für die Zeichenkette “abddcddbddbabdd”.

Durch wiederholte Anwendung dieses Vorgangs entsteht ein Binärbaum, wobei häufiger vorkommende Zeichen, die mit weniger Bit kodiert sind, näher an der Wurzel sind. Die Kodierung lässt sich am Baum ablesen, indem man an jeden linken Zweig eine 0 und an jeden rechten eine 1 schreibt. Der Weg von der Wurzel zu einem Zeichen ergibt dann seine Kodierung.

Für unser Beispiel: `'a'` $\mapsto$ 000, `'b'` $\mapsto$ 01, `'c'` $\mapsto$ 001, `'d'` $\mapsto$ 1

1. Um die Entnahme des Zeichens mit der niedrigsten Frequenz effizient zu gestalten, soll zuerst eine Prioritätswarteschlange `PriorityQueue<E>` geschrieben werden, die das Interface `PQInterface<E>` implementiert. Vorlagen für beide Dateien finden Sie im StudOn.

   Achtung: Bitte verändern Sie keine der vorgegebenen Signaturen oder Attribute und verwenden Sie unbedingt die vorgegebenen Schnittstellen!

Als zugrunde liegende Datenstruktur für die Prioritätswarteschlange soll eine Max-Halde (*max-heap*) dienen, also ein Binärbaum, in dem jeder Vaterknoten eine *höhere Priorität* als seine Kinderknoten hat. Die Baumstruktur soll dabei **nicht** über Referenzzeiger (`left/right`) verwaltet werden, sondern mittels einer `ArrayList <elements>`, in welcher die Knoten liegen (vgl. VL 12-51 ff.). In Abbildung 3 wird das Prinzip graphisch verdeutlicht.

Achtung: Die Verwendung von `java.util.PriorityQueue<E>` ist in dieser Aufgabe verboten!

Abbildung 3: Lineare Speicherung eines linksbalancierten Binärbaums.

a) Überlegen Sie sich zunächst, wie man aus dem Array-Index eines Vaterknotens die Indizes seiner Kinderknoten berechnen kann. Setzen Sie dies in den Methoden `leftChild(int idx)` bzw. `rightChild(int idx)` um. `parent()` soll aus dem Index eines Kindknotens den seines Vaterknotens berechnen.

b) Die Methode `bubbleUp(int idx)` soll das Element an der übergebenen Position im Array so lange (**rekursiv!**) im Baum nach oben wandern lassen, bis die Heap-Eigenschaft wieder erfüllt ist (vgl. VL 12-52 ff.).

c) Die Methode `trickleDown(int idx)` soll **rekursiv** die "Versickern"-Operation (vgl. VL 12-49 ff.) ab dem gegebenen Index durchführen, um die Heap-Eigenschaft wiederherzustellen.

d) Implementieren Sie nun die im Interface `PQInterface` angegebenen Methoden entsprechend den Kommentaren. Stellen Sie sicher, dass nach *jeder* Einfüge- bzw. Entnahmeoperation die Heap-Eigenschaft wiederhergestellt wird.

2. Komprimiert werden soll der Inhalt des String <s>, welcher in der main-Methode bereits angelegt ist. Hierfür wird wie folgt vorgegangen:

   Blattknoten im durch die Huffman-Kodierung entstehenden Baum werden durch die vorgegebene Klasse `LeafTreeNode` repräsentiert. Jeder Knoten speichert Zeichen und Frequenz (Auftretenshäufigkeit).

   Die Klasse `InnerTreeNode` hingegen stellt **innere** Knoten dar. Beide Klassen erben vom ebenfalls vorgegebenen Interface `TreeNode` und können somit gemeinsam in der Klasse `PriorityQueue` verwaltet werden, die – mit korrekt gewählten Prioritäten – automatisch die jeweils als nächstes zu verarbeitenden Knoten liefert.

   Ihre nächste Aufgabe ist es, die Methode `Huffman.huffmanCode(String s)` zu implementieren:

   a) Verwenden Sie die – bereits implementierte – Methode `characterHistogram(String s)` und erzeugen Sie für jedes in `s` vorkommende Zeichen einen **Blatt**knoten.

   b) Setzen Sie dann den oben geschilderten Algorithmus um, der immer die zwei Zeichen mit der *niedrigsten* Frequenz zu einem kombinierten Zeichen zusammenfügt.

Hinweis: Die `PriorityQueue` gibt immer das Element mit der *höchsten* Priorität zurück, für die Huffman-Kodierung benötigen Sie aber immer die beide Zeichen mit den jeweils *niedrigsten* Frequenzen. Beim Einfügen in die `PriorityQueue` müssen den Zeichen also eine entsprechende Priorität zugewiesen werden. Überlegen Sie sich einen Weg!

c) Sobald der komplette Baum erzeugt wurde, können Sie den Wurzelknoten die Kodierungstabelle erzeugen lassen und diese zurückgeben. Sehen Sie sich an, wie die Methode in `InnerTreeNode` und `LeafTreeNode` implementiert ist und übergeben Sie dem Wurzelknoten das passende Präfix. Als zweiten Parameter verwenden Sie eine leere `HashMap`.

3. Zuletzt soll in `Huffman.computeCompressionRatio(String s, Map<Character, String> map )` die Kompressionsrate berechnet werden. Vergleichen Sie dafür die Gesamtlänge der Huffman-Kodierung (⇝ `Huffman.lengthHuffmanCoding()`) mit der Kodierung als 8-Bit-ASCII-Zeichen (⇝ `Huffman.length8BitCoding`).

   Ein Rückgabewert von `0.75f` soll beispielsweise eine Einsparung von 75% ausdrücken.

4. Zum Testen Ihrer Implementierung können Sie den Sting s in der main-Methode auf einen beliebigen Wert setzen. Hierbei bietet es sich an Strings zu testen, in welchen einige Buchstaben sehr häufig andere dagegen nur selten vorkommen.

5. Geben Sie die Dateien `Huffman.java` und `PriorityQueue.java` im EST ab!

Sollte Ihr Programm nicht übersetz- bzw. ausführbar sein, wird die Lösung mit 0 Punkten bewertet. Stellen Sie also sicher, dass IntelliJ IDEA keine Fehler in Ihrem Programm anzeigt, Ihr Programm übersetz- und ausführbar ist sowie die in der Aufgabenstellung vorgegebenen Namen und Schnittstellen *exakt* eingehalten werden. Geben Sie am Schluss die Dateien `NaryTree.java`, `TreeAtCapacityException.java`, `PriorityQueue.java` und `Huffman.java` über die EST-Webseite ab. Wenn Sie die Aufgabe zusammen mit einem Übungspartner bearbeitet haben, geben Sie im EST unbedingt dessen Gruppenabgabe-Code an! Kontrollieren Sie, ob Ihre Namen am Anfang aller Dateien angegeben sind – schreiben Sie im Quellcode Ihre Angaben in einen Kommentar. Im EST-Abgabesystem können Sie modifizierte Dateien mehrfach abgeben. Nur die zuletzt hochgeladene Version wird bewertet.