4. Praktikumsaufgabe | 8. Januar 2021

Huffman-Kodierung

Implementieren Sie eine Klasse Huffman, welche einen minimalen Präfix-Code erzeugen kann sowie Text mithilfe dieses Codes kodieren und dekodieren kann.

Die Java-Klasse Huffman soll dabei folgende öffentlichen Konstruktoren und Methoden haben:

```
// Huffman-Klasse erzeugen.
Huffman()

// Häufigkeiten aller Zeichen aus einer Zeichenkette errechnen
Integer[] calculateFrequencies(String text)

// Präfix-Code erzeugen
HNode constructPrefixCode(Integer[] frequencies)

// Kann die Zeichenkette mit dem aktuellen Code dekodiert
werden?
boolean canEncode(String text)

//Kodieren der Zeichenkette
String encode(String text, boolean newPrefixCode)

//Dekodieren eines Huffman-kodierten Textes
String decode(String encodedText)

//Dekodieren mithilfe des übergebenen Präfix-Codebaums
String decode(String encodedText, HNode rootNode)

//Präfix-Codebaum ausgeben
void dumpPrefixCodes()
```

Weiterhin ist es sinnvoll, für die Arbeit mit dem Huffman-Code-Baum private rekursive Hilfs-Funktionen zu verwenden, um im Baum zu navigieren. Für die Baumstruktur gibt es eine Knoten-Klasse HNode.

Folgende Fehler sind abzufangen:
Ein in eine Methode übergebenes Objekt ist null. Bei einem Fehler soll das Programm eine entsprechende (selbst gewählte) Meldung ausgeben.
Außerdem sollte vor einer Kodierung mit newPrefixCode=false überprüft werden, ob der Text überhaupt kodiert werden kann.

Als Vorgabe finden Sie zwei Dateien im Canvas-Kurs:

- binheap.zip: Vorübersetzte Minimum-Vorrangwarteschlangen-Klasse
- huffman.java: Klassenvorlage mit zu implementierenden Methoden sowie kleines Testprogramm.

Beispiel für Test-Ein- und -Ausgaben:

```
>>> enc1 Abracadabra
enc1 Abracadabra
Kodierter Text: 10101111100101101010001111100
>>> dec 10101111100101101010001111100
dec 10101111100101101010001111100
Dekodierter Text: Abracadabra
>>> prefixes

>>> enc0 Abracadabra
enc0 Abracadabra
Kodierter Text:
01010101100101100100111101111111111010010111100101100100111110
>>> dec 01010101100101100100111101111111111010010111100101100100111110
dec 01010101100101100100111101111111111010010111100101100100111110
Dekodierter Text: Abracadabra
```

Abzugeben ist entweder eine einzige Java-Datei oder eine Zip-Datei mit mehreren Java-Dateien auf oberster Ebene (d. h. keine Unterordner). Die vorgegebene Datei binheap.zip, um die benötigte Minimum-Vorrangwarteschlange zu erstellen, ist nicht mit abzugeben.

```
// Knoten für den Huffman-Trie
class HNode{
	// chars enthält bei Blattknoten ein Zeichen, ansonsten alle
	Zeichen der darunterliegenden Knoten
	public String chars;
	// Linkes Kind
	public HNode leftChild;
	// Rechtes Kind
	public HNode rightChild;
}
class Huffman {
	// Feld mit Huffman-Codes zu den einzelnen Zeichen.
	// Wenn char c = 'a', dann ist codes[c] ein Code, der aus Nullen
	und Einsen besteht, mit dem das Zeichen a kodiert werden soll.
	private String[] codes;

	// Wurzelknoten des Präfix-Codebaums
```

```
private HNode root;

// Konstruktor
public Huffman() {
	// TODO
}

// Prüfen, ob ein Text mit dem aktuell erstellten Huffman-Code
kodiert werden kann, ob also alle Zeichen einen Präfix-Code
besitzen. Wenn ja, return true, wenn nein, return false.
// Prüfen, ob ein Text mit dem aktuell erstellten Huffman-Code
kodiert werden kann, ob also alle Zeichen einen Präfix-Code
besitzen. Wenn ja, return true, wenn nein, return false.
public boolean canEncode(String text){
	// TODO
	return true;
}

// Vor dem eigentlichen Algorithmus kann mit dieser Funktion die
Häufigkeit der einzelnen Zeichen aus dem übergebenen Text
errechnet werden.
// Hierzu kann die Anzahl des Vorkommens eines Zeichens
berechnet werden und in einem Array gespeichert werden.
// Für jedes Zeichen c enthält das Array f an Stelle c die
Häufigkeit (also f['a'] ist die Häufigkeit von a im Text. Kommt
das Zeichen nicht vor, ist die Häufigkeit 0.)
// Zur Erinnerung: ein char kann wie eine Ganzzahl verwendet
werden, daher funktioniert f[c] für jedes char c.
public Integer[] calculateFrequencies(String text){
	Integer[] f = new Integer[256];
	// TODO
	return f;
}

// Iterativer Algorithmus zur Erstellung des Präfix-Codes
(Skript S.115) mithilfe von BinHeap.
// frequencies enthält die Häufigkeiten (siehe
calculateFrequencies). Häufigkeit von 0 bedeutet, das
entsprechende Zeichen ist nicht im Text vorhanden und wir
brauchen keinen Präfixcode dafür.
// Die Funktion setzt den Knoten root auf den Wurzelknoten des
PräfixCode-Baums und gibt diesen Wurzelknoten außerdem zurück
public HNode constructPrefixCode(Integer[] frequencies){
	// TODO
	return root;
}

// Kodierung einer Zeichenkette text (Skript S.108)
// Die Ergebnis-Zeichenkette enthält nur Nullen und Einsen
```

```
	// (der Einfachheit halber wird dennoch ein String-Objekt
	verwendet)
	// Kodierung: linker Teilbaum -> 0, rechter Teilbaum -> 1
	// Erster Parameter: Zu kodierender Text
	// Zweiter Parameter zeigt an, ob ein neuer Präfixcode erzeugt
	werden soll (true) oder mit dem aktuellen Präfixcode gearbeitet
	werden soll (false)
	public String encode(String text, boolean newPrefixCode){
		String result = "";
		// TODO
		return result;
	}

	// Dekodierung eines Huffman-Kodierten Textes. (Skipt S.107)
	// Die Ergebnis-Zeichenkette ist der ursprüngliche Text vor der
	Huffman-Kodierung
	public String decode(String huffmanEncoded){
		// TODO
		return "";
	}

	// Dekodierung eines Huffman-Kodierten Textes mithilfe des
	übergebenen Präfix-Codebaums. (Skipt S.107) Der aktuelle Baum
	soll dabei nicht überschrieben werden.
	// Die Ergebnis-Zeichenkette ist der ursprüngliche Text vor der
	Huffman-Kodierung
	public String decode(String huffmanEncoded, HNode rootNode){
		String result = "";
		// TODO
		return result;
	}

	// Präfixcodes ausgeben
	// Reihenfolge: preOrder, also WLR, zuerst Wurzel, dann linker
	Teilbaum, dann rechter Teilbaum
	public void dumpPrefixCodes(){
		// TODO
	}
}
```