OPERATIONS RESEARCH
KACKERTSTR. 7
52072 AACHEN

Prof. Dr. Marco Lübbecke
Dipl.-Math. oec. Michael Bastubbe
Christian Puchert, M.Sc.
Annika Thome, M.Sc.

Johannes Gocke
Dustin Hütter
Sascha Kurowski

Programmieren, Algorithmen, Datenstrukturen
8. Programmieraufgabe

Abnahme bis spätestens 17.01.2014

Huffman Algorithmus

In dieser Programmieraufgabe werdet Ihr den Huffman Algorithmus, inklusive der notwendigen Datenstrukturen und Dekompressionsverfahren implementieren. Die Aufgabe ist sehr umfangreich und erfordert, dass Ihr euch die Bearbeitungszeit wohlüberlegt einteilt. Beginnt also frühestmöglich mit ihrer Bearbeitung. Die Aufgabe besteht aus den folgenden vier Teilen:

(A) Implementation einer generischen Priority Queue,

(B) Implementation der Huffman Bäume,

(C) Implementation des Huffman Algorithmus und

(D) Implementation des Dekompressionsverfahrens.

Entpackt die Vorgabedatei und verschafft euch einen Überblick über die enthaltenen Packages.

(A) Implementation einer generischen Priority Queue

(i) Vervollständigt die Klasse `PriorityQueue<T>` im Package `padTree`, die einen aufsteigenden Heap darstellt, welcher durch ein Array realisiert wird. Die Objekte, die im Heap gespeichert werden, müssen das Interface `Comparable<T>` implementieren, welches zum Package `java.lang` gehört.

Wie Ihr am Typparameter `T` erkennen könnt, soll Eure Klasse typsicher sein, d.h. der Typ der Objekte, die in einer Instanz von `PriorityQueue` gespeichert werden können, wird bei ihrer Deklaration festgelegt. Denkt also daran, dass Ihr den Typparameter auch innerhalb Eurer Klasse überall dort benutzt, wo es für die Typsicherheit nötig ist. Ihr sollt nun folgende öffentliche Methoden bereitstellen:

- Einen Konstruktor, der eine Kapazität übergeben bekommt und eine entsprechende leere `PriorityQueue` initialisiert.
- Einen Konstruktor, welcher ein Array von Objekten vom Typ `Comparable<T>` übergeben bekommt und daraus einen vollen Heap mit all diesen Objekten initialisiert.
- `push`, welche ein neues Element übergeben bekommt, es dem Heap hinzufügt und die Heap-Eigenschaft wieder herstellt.
- `top`, welche das kleinste Element im Heap zurückgibt ohne es zu löschen und eine `NoSuchElementException` mit entsprechendem Kommentar wirft, falls der Heap leer ist.

- `pop`, welche das gleiche tut wie `top`, aber gleichzeitig das zurückgegebene Element löscht und danach die Heap-Eigenschaft wieder herstellt.
- `size`, welche die Anzahl der Elemente im Heap zurückgibt.
- `isEmpty`, welche genau dann `true` zurückgibt, wenn der Heap keine Elemente enthält.
- Eine Methode `toString`, welche den Heap möglichst anschaulich (d. h. unter Ausnutzung der Tatsache, dass der Heap ein Baum ist) auf die Konsole ausgibt. Hierzu schreibt Ihr eine rekursive Methode, die den Baum sinnvoll traversiert und dabei die gewünschte Ausgabe zusammenbaut. Wie genau die Ausgabe am Ende aussieht bleibt Eurer Kreativität überlassen.

Zur Realisierung dieser Funktionalitäten dürft Ihr neben den bereits vorhandenen keine weiteren Klassen- oder Instanzvariablen benutzen und ausschließlich folgende private Methoden implementieren:

- `_parent`, welche zu einem gegebenen Arrayindex eines Elements den Index seines Vaters zurückgibt oder `UNUSED`, falls er nicht existiert.
- `_left` und `_right`, welche zu einem gegebenen Arrayindex eines Elements den Index seines linken bzw. rechten Sohnes zurückgibt oder `UNUSED`, falls er nicht existiert.
- `_heapify`, welche von einem Element, dessen Arrayindex übergeben wird, aus abwärts die Heapeigenschaft herstellt. Bei ihrem Aufruf soll davon ausgegangen werden, dass die beiden Teilbäume, die an den Kindern des Elements hängen, dessen Index übergeben wird, bereits die Heapeigenschaft erfüllen.

In `PriorityQueue.java` stehen außerdem schon folgende Funktionalitäten bereit:

- Eine private Variable `_lastIndex`, welche Ihr unbedingt benutzen solltet, um Einfügeoperationen effizient zu gestalten und schnellen Zugriff auf die aktuelle Größe des Heaps zu haben.
- Eine boolsche Variable `_checkMode`, welche mit der Methode `setCheck` gesetzt werden kann.
- Eine Methode `_checkHeap`, welche für den Fall `(_checkMode == true)` den gesamten aktuellen Heap auf Konsistenz prüft und Inkonsistenzen ausgibt. Diese Methode ist nur dazu da, Euch die Fehlersuche zu erleichtern. Durch den Schalter `_checkMode` kann also quasi ein (etwas ineffizienterer) Debug-Modus aktiviert werden.

(ii) Schreibt eine `main`-Methode, die als einziges Kommandozeilenargument eine Arraylänge (natürliche Zahl) bekommt, ein Array von Zufallszahlen eben dieser Länge generiert (z. B. mit Hilfe des Packages `java.util.Random`) und mittels Heap Sort sortiert. Am besten benutzt ihr Zahlen vom Typ `Integer` oder `Double`, da diese Klassen bereits die Interfaces `Comparable<Integer>` bzw. `Comparable<Double>` implementieren.

- Die `main`-Methode soll eine Usage-Ausgabe haben, die immer dann erscheint, wenn beim Aufruf keine Argumente angegeben werden. Sie soll dem Benutzer sagen, welche Argumente das Programm erwartet (Syntaxzeile) und dazu ggf. kurze Erläuterungen geben. Eine solche Ausgabe gehört zu allen Applikationen (probiert z. B. in einer Shell das Kommando `scp` ohne Argumente aufzurufen).

- Gebt das Array vor- und nach der Sortierung aus. Gebt außerdem den Heap mittels Eurer `toString`-Methode aus, nachdem Ihr alle Elemente eingefügt und noch keines wieder entfernt habt.

(B) Implementation der Huffman Bäume

(i) Schreibt eine abstrakte Klasse `BinTree<T>` in `padTree`, die die Grundlage für die Implementation verschiedener binärer Bäume darstellt. Verfahrt dazu so, wie in der Übung vom 17.12.2013 beschrieben. Die Klasse soll also unter anderem eine innere Klasse `BinTreeNode<T>` mit entsprechenden Methoden und einen Baumknoten `_dummy` haben, an dem dann als linkes Kind die tatsächliche Wurzel des Baumes hängt. Außerdem sollen folgende Funktionalitäten bereitgestellt werden:

- ein Default-Konstruktor, der einen leeren Baum initialisiert.
- ein Konstruktor, der ein Objekt vom Typ `T` übergeben bekommt und daraus einen einelementigen Baum erzeugt.
- die Methoden `isEmpty`, `getSize`, `getHeight` und `toString`. Nutzt dabei sinnvoll rekursive Strukturen von Bäumen aus und schreibt entsprechende rekursive Hilfsmethoden!
- die Möglichkeit, den Baum mit den Methoden `reset`, `increment` und `isAtEnd` wie in der Übung gezeigt, iterativ zu traversieren.
- zusätzlich die Methoden `isAtLeaf`, die zurückgibt, ob `_curr` gerade auf ein Blatt zeigt, und `currentData`, welche das Objekt vom Typ `T` des Baumknotens zurückgibt, auf das `_curr` gerade zeigt.

(ii) Schreibt im Paket `padTree` eine Klasse `HuffmanTree`, die ihr von `BinTree<Valued>` ableitet und die das Interface `Comparable<HuffmanTree>` implementiert. Das Interface `Valued` steht im Package `padInterfaces` und garantiert die Existenz einer Methode `doubleValue`, welche den einem `Valued`-Objekt zugeordneten `double`-Wert zurückgibt. Gebt eurer Klasse zusätzlich zum von `BinTree` geerbten `_curr` eine weitere Knotenreferenz `_free`, welche ihr beliebig im Baum bewegen könnt (siehe unten). Ihr werdet sie beim Lesen von Codes aus dem Huffman Baum brauchen.

Schreibt in eurer Klasse eine innere Klasse `HuffmanTreeNode`, welche ihr analog von `BinTree.BinTreeNode<Valued>` ableitet. Diese Klasse soll sowohl einen Konstruktor haben, der ein `Valued`-Objekt übergeben bekommt, als auch einen, dem nur ein `double`-Wert übergeben wird. (Wofür werdet ihr die beiden Konstruktoren später brauchen?)

Schreibt außerdem im Package `padTree` eine Hilfsklasse `HuffmanToken.java`, die `Valued` implementiert und alle für den Huffman Code notwendigen Daten eines Zeichens kapselt: das Zeichen selbst als `byte`, seine Häufigkeit als `int` und seinen zugehörigen Bitcode als `padList.LinkedList<Boolean>`. Objekte von diesem Typ werden in den Knoten Eures `HuffmanTree` gespeichert.

Die folgenden Methoden soll Eure Klasse `HuffmanTree` wenigstens enthalten:

- Einen Konstruktor, der ein `Valued`-Objekt übergeben bekommt und daraus einen einelementigen Baum erzeugt.

- Einen Konstruktor, der zwei Objekte vom Typ `HuffmanTree` übergeben bekommt und sie zu einem neuen `HuffmanTree` zusammenfügt.
- Die Methoden `freeToCurr` und `freeToRoot`, die die Referenz `_free` auf `_curr` bzw. auf die Wurzel des Baumes setzen; `freeAtRoot`, `freeAtLeaf` und `freeAtLeftChild`, die zurückgeben ob `_free` auf die Wurzel bzw. ein Blatt bzw. ein linkes Kind zeigt; und `freeData`, die das `HuffmanToken` des Baumknotens als `Valued` zurückgibt, auf den `_free` gerade zeigt.
- Die Methode `advance`, die `_free` in eine bestimmte Richtung (`UP`, `LEFT` oder `RIGHT`) bewegt falls möglich. Die Richtung wird der Methode als `int` übergeben; am besten definiert ihr in `HuffmanTree` entsprechende Konstanten. Außerdem gibt die Methode einen `boolean` zurück, welcher besagt ob `_free` bewegt werden konnte.
- Die Methode `append`, welche entweder `LEFT` oder `RIGHT` und ein `HuffmanToken` übergeben bekommt und einen Baumknoten mit diesem Token in der angegebenen Richtung an den Knoten anhängt, auf den `_free` gerade zeigt, falls dort noch kein Knoten ist. Wieder soll ein `boolean` zurückgegeben werden, der signalisiert ob die gewünschte Operation ausgeführt werden konnte.
- Eine Methode `_checkTree()`, welche die Invarianten des Huffman-Codebaums überprüft: Alle inneren Knoten haben zwei Kinder und der Häufigkeitswert in ihnen ist jeweils gleich der Summe der Häufigkeitswerte dieser Kinder. Die Idee hinter dieser Methode ist ähnlich der von `_checkHeap` in `PriorityQueue`: Ihr ruft diese Methode überall auf, wo ein `HuffmanTree` erstellt oder verändert wird. Verläuft der Check nicht erfolgreich, wird ausgegeben an welcher Stelle des Baumes welche Inkonsistenz entdeckt wurde. Ob der Check erfolgreich war, wird per `boolean` zurückgegeben.

Man kann die Nützlichkeit solcher Methoden beim Programmieren von Datenstrukturen kaum überschätzen! Es lohnt sich also sehr, hier ein wenig Arbeit zu investieren und der Methode eine sinnvolle Ausgabe zu geben.

(iii) Schreibt euch eine kleine Testmethode `main`, die einen beliebigen `HuffmanTree` mit Hilfe der beiden Möglichkeiten (durch Konstruktoren bzw. fortgesetztes `append`) aufbaut und seine Konsistenz mit `_checkTree` überprüft. Lasst euch die Knoten des Baumes mit Hilfe der iterativen Traversierung in `BinTree` ausgeben und überprüft ihre Korrektheit.

(C) Implementation des Huffman Algorithmus

(i) Schreibt ein Kompressionsprogramm `Compress.java`, welches als Kernverfahren den Algorithmus von Huffman verwendet. Dieses Programm soll die Klasse `HuffmanTree` verwenden, aber nicht selbst in Package `padTree` stehen (es ist also die erste Klasse, die direkt in dem Verzeichnis mit eurer Programmieraufgabe steht).

Die zu codierenden Zeichen sind die einzelnen Bytes, Ihr müsst aus der Quelldatei also byteweise lesen, etwa mit der Klasse `FileInputStream` (siehe API). Ihr könnt die Performance beim Lesen steigern, indem Ihr außerdem die Klasse `BufferedInputStream` verwendet. Die Verwendung von `char`s oder der Klasse `StreamTokenizer` kommen hier nicht in Frage!

Zum Schreiben der Zieldatei braucht ihr die Klasse `BitOutputFile` aus dem Package `padIo`, die einzelne Bits schreiben kann (siehe JavaDoc). Das Schreiben der Zieldatei

besteht aus zwei Phasen: Zuerst schreibt ihr die Codetabelle, dann die codierten einzelnen Bytes der Inputdatei. Überlegt euch ein möglichst sparsames Format für die Code-Tabelle.

(D) Implementation des Dekompressionsverfahrens

(i) Schreibt nun das zugehörige Dekompressionsprogramm, welches ebenfalls nicht Teil von `padTree` sein soll. Zuerst wird eine von euch komprimierte Quelldatei mit Hilfe von `padIo.BitInputFile` eingelesen. Dann rekonstruiert ihr aus der Codetabelle den Huffman-Baum von der Wurzel her und hängt in seine Blätter die codierten Zeichen (Bytes). Verwendet den Baum dann zur Wiederherstellung der originalen Quelldatei, welche ihr byteweise mit Hilfe der Klassen `FileOutputStream` (und optional `BufferedOutputStream`) schreibt.

(ii) Ergänzt `Compress.java` um einige Ausgaben zur Bewertung der erzielten Kompression: Gebt aus, wie groß die komprimierte Datei im Verhältnis zur Originaldatei ist (in Prozent) und berechnet die Redundanz des zugehörigen Huffman Codes (siehe Übung vom 18.12.).

Was die Gestaltung Eures Programms angeht, seid Ihr recht frei. Haltet Euch aber in jedem Fall an folgende Vorgaben:

- Ihr dürft keine Dateien komplett in den Speicher lesen. Das bedeutet, dass Ihr zum Komprimieren die Originaldatei zweimal lesen müsst: einmal zum Zählen der Häufigkeiten und einmal zum eigentlichen Komprimieren.
- Dateien, die nur aus der Wiederholung eines und desselben Bytes bestehen, müssen ebenfalls komprimiert werden können. Der Codebaum hat in diesem Fall eine besondere Gestalt.
- Werden die beiden Programme ohne Argumente aufgerufen, soll eine Usage-Beschreibung ausgegeben werden.
- Der Benutzer soll als Argumente jeweils die Namen der Quell- und Zieldatei vorgeben können. Ihr könnt den Namen für die Zieldatei optional machen und eine Default-Endung vorsehen, die an den Namen der Quelldatei angehängt wird (z.B. `.cpr` o.ä.).
- Es soll mit einer „ordentlichen“ Fehlermeldung terminiert werden, wenn keine Quelldatei mit dem angegebenen Namen gefunden wurde. Es soll ebenfalls mit einer Fehlermeldung terminiert und nichts geschrieben werden, wenn eine Datei mit dem Namen für die Zieldatei bereits existiert.
- Die dekomprimierten Dateien müssen *exakt* mit ihren Originaldateien übereinstimmen (Test mit dem LINUX-Kommando `cmp`).

Im Lernraum findet Ihr in `examples.zip` einige Dateien zum Testen eures Programms.

Viel Spaß und Erfolg!