Prof. Dr. Ralf Hinze
Sebastian Schloßer, M.Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Projekt 1: Programmierpraktikum 2022

Ausgabe: 15. Juni 2022
Abgabe: 5. Juli 2022, 15 Uhr

GitLab Team Repositories

Wir verwenden auch für die Projekte wieder die GitLab Repositories. Führen Sie den Befehl `git pull` aus, damit die Vorlagen heruntergeladen werden. Sie sollten nun einen neuen Ordner `P1` sehen, den Sie wie gewohnt in IntelliJ IDEA öffnen können. Es handelt sich wieder um ein Gradle Projekt, sodass Abhängigkeiten automatisch heruntergeladen werden.

Ihre Lösungen müssen Sie in das lokale Repository committen (`git commit`) und dann auf den GitLab Server übertragen (`git push`). Sprechen Sie sich zur Bearbeitung der Aufgaben mit Ihren Abgabepartner*innen ab, damit Sie unnötige Konflikte vermeiden.

Vorlesungsumfrage

Wir sind an Ihrem Feedback interessiert. Bitte nehmen Sie sich die Zeit, die anonyme Vorlesungsumfrage auszufüllen. Studierende mit `@cs.uni-kl.de` Mailadresse haben dazu bereits eine E-Mail erhalten, alle anderen können sich auf https://vlu.cs.uni-kl.de/teilnahme/ z. B. mit ihrer RHRK Mailadresse einen Link anfordern. Ihr Feedback hilft uns, die Vorlesung und die Übungen weiter zu verbessern.

Bearbeitungshinweise

Sie haben für die Bearbeitung des Projekts drei Wochen Zeit und es ist entsprechend umfangreich. **Beginnen Sie also rechtzeitig mit der Bearbeitung** und nutzen Sie die Sprechstunden!

Wir erwarten, dass bis Sonntag, 26.06.2022 23:59 Uhr Aufgabe 1 weitestgehend gelöst ist, d. h. dass sich der Programmcode und die Testfälle im GitLab Repository befinden, ein Großteil der Testfälle bestanden wird und die Code Coverage sich nahe am Zielwert befindet.

Projektabnahmen

Zur Erinnerung: Bei der Projektabnahme müssen Sie Ihre Arbeit präsentieren. **Jedes Teammitglied muss dabei einen Redeanteil haben.** Wer sich nicht beteiligt oder der Projektabnahme unentschuldigt fernbleibt, wird aus dem Programmierpraktikum ausgeschlossen und kann das Modul dieses Semester nicht mehr bestehen. Beachten Sie darüber hinaus die Hinweise auf dem Aufgabenblatt von Miniprojekt 1.

Einleitung

Ziel dieses Projekts ist es, ein Programm zur Komprimierung von Dateien zu entwickeln. Wir verwenden dazu die Huffman-Kodierung[1]. Da der Algorithmus zum Aufbau der Code-Tabelle eine Prioritätswarteschlange[2] benötigt, müssen Sie in Aufgabe 1 zunächst diese Datenstruktur implementieren.

Aufgabe 1 Prioritätswarteschlangen

Eine Prioritätswarteschlange ist eine Datenstruktur, die die folgende Schnittstelle implementiert:

```
public interface PriorityQueue<E> {
    void insert(long priority, E element);
    Entry<E> extractMin();
    Entry<E> peek();
    boolean isEmpty();
}
```

Dabei ist `Entry<E>` wie folgt definiert:

```
public class Entry<E> {
    public final long priority;
    public final E element;

    public Entry(long priority, E element) {
        this.priority = priority;
        this.element = element;
    }
```

Eine Prioritätswarteschlange speichert Elemente jeweils zusammen mit einer Priorität als Zahl vom Typ `long`. Je kleiner die Zahl, desto höher ist die Priorität. Während beim Entnehmen von Elementen aus einer normalen Warteschlangen immer das Element, das sich schon am längsten in der Warteschlange befindet, zuerst entnommen wird, wird bei Prioritätswarteschlangen immer das Element mit der höchsten Priorität (d. h. mit der kleinsten Zahl) zuerst entnommen.

Die Methode `insert` fügt das gegebene Element mit der angegebenen Priorität in die Prioritätswarteschlange ein. Die Methode `extractMin` entfernt das Element mit der höchsten Priorität (d. h. mit der kleinsten Zahl) aus der Prioritätswarteschlange und gibt es zusammen mit seiner Priorität zurück. In funktionalen Programmiersprachen könnte man hier als Rückgabetyp `(long, E)` wählen. Solche Tupel-Typen sind in Java aber nicht möglich. Deshalb haben wir eine eigene Klasse `Entry` definiert, die an dieser Stelle verwendet wird. Wenn die Warteschlange leer ist, dann soll eine `java.util.NoSuchElementException` geworfen werden.

Die Methode `peek` gibt ähnlich wie `extractMin` das Element mit der höchsten Priorität zurück, aber die Prioritätswarteschlange wird nicht verändert, das Element bleibt also weiterhin in der Prioritätswarteschlange enthalten. Außerdem soll `peek` bei einer leeren Warteschlange `null` zurückgeben und keine Ausnahme werfen. Die Methode `isEmpty` prüft, ob die Prioritätswarteschlange leer ist.

[1] https://de.wikipedia.org/wiki/Huffman-Kodierung
[2] https://de.wikipedia.org/wiki/Vorrangwarteschlange

Legen Sie Ihre Lösungen der folgenden Aufgaben im Paket `de.tukl.programmierpraktikum2022.p1.a1` ab: Implementierung in den Ordner `src/main/java`, Testfälle in den Ordner `src/test/java`.

a) In der Vorlesung haben Sie den Ansatz der testgetriebenen Entwicklung (engl. *test-driven development*) kennengelernt. Folgen Sie diesem Ansatz, indem Sie zunächst Testfälle für die Prioritätswarteschlange in der Klasse `PriorityQueueTest` erstellen, bevor Sie mit konkreten Implementierungen in den folgenden Aufgabenteilen beginnen. Sie können dazu als Vorlage die Methode `testExample` verwenden.

   Da unsere Prioritätswarteschlangen alle dieselbe Schnittstelle implementieren, müssen wir nicht für jede Prioritätswarteschlange neue Tests schreiben. Stattdessen instanziiert die Methode `getPriorityQueueInstances` die verschiedenen Prioritätswarteschlangen und übergibt diese an den parametrisierten Test (vgl. Annotationen `@ParameterizedTest` und `@MethodSource`).

   Es ist für diese Aufgabe in Ordnung, wenn in der Testklasse einige Beispiele getestet werden.

   Wenn Sie möchten, dürfen Sie aber gerne auch Zufallstests erstellen und für die Tests den vollen Umfang der Standardbibliothek und von JUnit nutzen.

b) Erstellen Sie eine Klasse `ListQueue`, welche die Schnittstelle `PriorityQueue` implementiert. In Ihrer Klasse soll die Prioritätswarteschlange durch eine **sortierte Liste** repräsentiert werden. Verwenden Sie dazu die Klasse `java.util.LinkedList` aus der Standardbibliothek.

c) Erstellen Sie eine Klasse `SkewHeap`, welche die Schnittstelle `PriorityQueue` implementiert. In Ihrer Klasse soll die Prioritätswarteschlange durch einen **Skew Heap** (selbst-balancierender Heap) implementiert werden. Recherchieren Sie die Eigenschaften eines Skew Heaps in der Literatur oder im Internet.

   Die sogenannte `merge` Operation zum Verschmelzen von zwei Skew Heaps ist von zentraler Bedeutung. Sie ist zwar nicht Teil unserer Schnittstelle, kann aber selbstverständlich als zusätzliche Methode implementiert werden. Sie können `insert` und `extractMin` mit Hilfe von `merge` implementieren.

   Den `merge` Algorithmus haben Sie wahrscheinlich bei Ihrer Literaturrecherche gefunden[3]. Geben Sie in Ihrem Programmcode die verwendete Quelle an.

d) *(Optional)* Implementieren Sie die Schnittstelle `PriorityQueue` als Fibonacci Heap [4].

e) Stellen Sie sicher, dass Sie mit Ihren Tests aus Aufgabenteil a) für jede Ihrer `PriorityQueue` Implementierungen mindestens eine Instruction Coverage von 95% erreichen. Erweitern Sie Ihre Tests gegebenenfalls oder fügen Sie neue Tests hinzu.

[3] z.B. https://de.wikipedia.org/wiki/Skew_Heap#Rekursive_Verschmelzung_zweier_Heaps_(recursive_merge) oder https://en.wikipedia.org/wiki/Skew_heap#Merging_two_heaps

[4] https://de.wikipedia.org/wiki/Fibonacci-Heap

Aufgabe 2 Huffman-Kodierung

Falls Sie die Huffman-Kodierung bislang noch nicht kennen, recherchieren Sie in der Literatur oder im Internet die Hintergründe. Wir gehen hier lediglich auf einige Details spezifisch für unsere Anwendung ein.

Eingabealphabet Man benötigt für die Huffman-Kodierung ein Eingabealphabet, also eine Menge an verschiedenen Symbolen die betrachtet werden. Wenn man *Texte* codieren möchte, bietet es sich an, ein Symbol pro Text-Zeichen (Buchstabe, Leerzeichen, Sonderzeichen, ...) zu definieren. Will man jedoch beliebige *Dateien* (ggf. Binärdaten) mit der Huffman-Kodierung komprimieren, dann muss man die Zeichen anders wählen. Wir entscheiden uns dazu, als Eingabealphabet die 256 verschiedenen Byte-Werte (Zahlen von 0 bis 255) zu wählen. Mit der Unicode-Kodierung verwenden Textdateien ohne Sonderzeichen ein Byte pro Zeichen, sodass wir dem ursprünglichen Ziel nahe kommen. Wählt man die Symbole zu klein, beispielsweise nur zwei Symbole für das 0-Bit und das 1-Bit, dann kann Huffman gar nichts komprimieren, da genau ein Bit benötigt wird, um das eine Bit Information zu kodieren, unabhängig von deren Häufigkeit. Wählt man die Symbole hingegen größer, beispielsweise Blöcke aus mehreren Bytes, dann steigt der Speicherbedarf für die Metadaten (Huffman-Baum). Zudem ist eine fixe Symbolgröße (immer genau ein Byte pro Symbol) von Vorteil, denn das vereinfacht wie Sie später sehen werden die Implementierung.

Häufigkeitstabelle Eingabe für den Huffman-Algorithmus ist eine Häufigkeitstabelle. Wir erhalten für jeden der 256 möglichen Byte-Werte die Info, wie oft dieser Wert vorkommt. Wir verwenden dazu ein Array `long[] frequencies` mit der Länge 256. Ist `frequencies[97] = 4711`, so kommt der Byte-Wert 97 (Zeichen `a` in der Unicode/Ascii-Tabelle) 4711 mal vor. Wir benutzen **`long`** zum Zählen der Zeichen, da **`int`** nur Zahlen bis $2^{31} - 1$ unterstützt und wir so bei Dateien ab 2 Gigabyte Probleme bekommen. Mit **`long`** unterstützen wir alle Dateien, die kleiner als 8 Exabyte (2^{63} Bytes) sind.

Datentyp für Bytes In Java gibt es einen Datentyp **`byte`** der die 256 verschiedene Werte -128 bis $+127$ annehmen kann. Durch den negativen Bereich wäre die Verwendung etwas umständlich, beispielsweise wenn man die **`byte`**s als Index in einem Array verwenden will (siehe Häufigkeitstabelle). Daher nutzen wir den Typ **`int`** und verwenden nur die Zahlen 0 bis 255. Das ist übrigens auch der Typ, den die Standardbibliothek gewählt hat, um einzelne Byte-Werte zu lesen[5] und schreiben[6]. In den Testfällen können Sie auch **`char`**s verwenden, um Bytes für lesbare Zeichen zu erzeugen. So ist beispielsweise `(int) 'a' = 97`.

Datentyp für Bits Der Huffman-Algorithmus weist jedem Symbol eine Bitfolge zu. Je häufiger das Symbol vorkommt, desto kürzer ist die Bitfolge. Java kennt keinen Typ namens `bit`, aber es gibt einen anderen Typ, der genau zwei Werte annehmen kann: **`boolean`**. Wir nutzen **`false`** für das 0-Bit und **`true`** für das 1-Bit.

[5] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html#read()
[6] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStream.html#write(int)

Huffman-Baum Beim Verarbeiten der Häufigkeitstabelle bauen wir den Huffman-Baum auf. In einer funktionalen Programmiersprache (beispielsweise F#) hätten wir dazu den folgenden Datentyp definiert:

```
type Tree = | Leaf of int
            | Node of (Tree * Tree)
```

Ein Baum ist also *entweder* ein Blatt mit einer Zahl zwischen 0 und 255, *oder* ein innerer Knoten mit einen linken und einen rechten Teilbaum. Mit Musterabgleich (engl. *pattern matching*) könnte man dann zwischen den beiden Fällen unterscheiden. In Java lässt sich das leider nicht so schön umsetzen. Verwendet man verschiedene Klassen für Blätter und innere Knoten, dann muss man mit **`instanceof`** den Typ prüfen und anschließend auf die jeweilige Klasse casten. Da es in der Verwendung einfacher ist, nutzen wir nur eine Klasse, die sowohl Blätter als auch innere Knoten darstellen kann. Die Klasse ist bereits vorgegeben:

```
public final class Node {
    public Node(int data) { ... }
    public Node(Node left, Node right) { ... }

    public boolean isLeaf() { ...  }
    public int getData() { ... }
    public Node getLeft() { ... }
    public Node getRight() { ... }
    ...
}
```

Es gibt zwei verschiedene Konstruktoren: Der erste erzeugt ein Blatt, der zweite einen inneren Knoten. Die Methode `isLeaf` überprüft, ob es sich um ein Blatt (Rückgabe **`true`**) oder um einen inneren Knoten handelt (Rückgabe **`false`**). Die Methoden `getData`, `getLeft` und `getRight` geben jeweils die im Blatt enthaltene Zahl, den linken Teilbaum bzw. den rechten Teilbaum zurück. Wichtig ist, dass Sie vor der Verwendung dieser Methoden mit `isLeaf` überprüfen, um welche Art von Knoten (Blatt oder innerer Knoten) es sich handelt. Die jeweils nicht unterstützten Methoden werfen eine `UnsupportedOperationException`.

In der Literatur haben Sie vielleicht gelesen, dass auch die Häufigkeit Teil des Knotens sein muss. Das ist bei uns nicht der Fall, denn wir speichern zum Aufbau des Baumes die Knoten in einer `PriorityQueue<Node>` und können die Häufigkeit dann aus dem `Entry<Node>` auslesen – die `priority` im `Entry` ist die Häufigkeit.

BitInputStream und BitOutputStream Wir wollen beim Komprimieren von (möglicherweise sehr großen) Dateien *nicht* die komplette Bitfolge im Arbeitsspeicher haben. Stattdessen lesen und schreiben wir direkt aus der Eingabe- bzw. in die Ausgabedatei. Dazu verwendet man üblicherweise einen `InputStream` bzw. einen `OutputStream` aus der Standardbibliothek. Leider können diese beiden Klassen nur mit ganzen **`byte`**s arbeiten, wir müssen aber auch einzelne Bits lesen und schreiben können. Deshalb haben wir Ihnen im Paket `de.tukl.programmierpraktikum2022.p1.util` Hilfsmittel bereitgestellt, mit denen auch einzelne Bits gelesen bzw. geschrieben werden können:

```
public interface BitInputStream {
    boolean readBit() throws IOException;
    int readByte() throws IOException;
}

public interface BitOutputStream {
    void writeBit(boolean bit) throws IOException;
    void writeByte(int data) throws IOException;
}
```

Sie müssen diese beiden Schnittstellen *nicht selbst implementieren*, die Implementierung ist von uns vorgegeben. In den Teilaufgaben c) und d) bekommen Sie Parameter vom Typ `BitInputStream` bzw. `BitOutputStream` übergeben, dort können Sie diese Methoden also *verwenden*.

Legen Sie Ihre Lösungen der folgenden Aufgaben im Paket `de.tukl.programmierpraktikum2022.p1.a2` ab: Implementierung in den Ordner `src/main/java`, Testfälle in den Ordner `src/test/java`.

Starten Sie mit Teilaufgabe a). Anschließend können Sie b), c) und d) parallel bearbeiten.

a) Erstellen Sie eine Klasse `HuffmanImpl` **`implements`** `Huffman`. Die von der Schnittstelle geforderten Methoden sollen zunächst keine Implementierung haben. Der Rumpf der Methoden soll daher jeweils eine Ausnahme werfen: **`throw new`** `RuntimeException("TODO");`

   Öffnen Sie anschließend die bereits angelegte Klasse `HuffmanFactory` und geben Sie in der statischen Methode `getHuffman` eine Instanz Ihrer `HuffmanImpl` Klasse zurück.

   Die folgenden Teilaufgaben widmen sich den einzelnen Methoden aus der `Huffman` Schnittstelle. Schreiben Sie Ihre Implementierung dazu dann nach und nach in Ihre `HuffmanImpl` Klasse.

b) In dieser Teilaufgabe sollen Sie die folgende Schnittstellen-Methode implementieren und testen:

```
Node buildTree(long[] frequencies);
```

   Sie können annahmen, dass die übergebene Häufigkeitstabelle die Länge 256 hat, alle Einträge ≥ 0 sind und mindestens ein Eintrag > 0 ist. Die Methode soll den Huffman-Baum aufbauen und zurückgeben.

   Gehen Sie dazu wie folgt vor:

   1. Erstellen Sie eine (leere) Prioritätswarteschlange. Nutzen Sie eine der Klassen, die Sie in Aufgabe 1 implementiert haben.
   2. Erstellen Sie für jedes Symbol aus der Häufigkeitstabelle, das eine Häufigkeit > 0 hat, ein Blatt und fügen Sie es mit der jeweiligen Häufigkeit als Priorität in die Prioritätswarteschlange ein. Ist beispielsweise `frequencies[97] = 4711`, dann sollen Sie ein Blatt mit dem Inhalt `97` erstellen und dieses mit der Priorität `4711` in die Prioritätswarteschlange einfügen.
   3. Wiederholen Sie die folgenden Schritte, bis die Prioritätswarteschlange nur noch einen Eintrag hat:
      i. Entnehmen Sie zwei Einträge aus der Prioritätswarteschlange.
      ii. Erstellen Sie einen neuen inneren Knoten, der die beiden Bäume der soeben entnommenen Einträge als Teilbäume enthält.
      iii. Fügen Sie den neuen Knoten in die Prioritätswarteschlange ein. Wählen Sie als Priorität dabei die Summe der Prioritäten der beiden in Schritt i entnommenen Einträge. Somit ist die Priorität eines Baumes immer die Summe der Häufigkeiten aller Symbole, die in diesem Baum enthalten sind.
   4. Geben Sie den Knoten zurück, der im letzten noch in der Warteschlange verbleibenden Eintrag enthaltenen ist. Das ist die Wurzel des Huffman-Baumes.

   Tipp: Für Schritt 3 können Sie eine **`while`** `(true)` Schleife benutzen, die Sie mit einem **`return`** beenden, wenn Sie keinen zweiten Eintrag aus der Prioritätswarteschlange entnehmen können.

   Testen Sie nun Ihre Implementierung. Die Klasse `HuffmanTest` enthält eine Vorlage `testExample1`, die Sie erweitern können. Erstellen Sie zusätzliche Testfälle mit verschiedenen Häufigkeitstabellen.

   Tipp: Die von uns vorgegebene Klasse `Node` bietet eine `equals` Methode. Daher können Sie mit `assertEquals(expectedNode, actualNode)` im Testfall direkt die Gleichheit zweier `Nodes` sicherstellen.

c) In dieser Teilaufgabe sollen Sie die folgenden Schnittstellen-Methoden testen und implementieren:

```
Encoder buildEncoder(Node rootNode);
Decoder buildDecoder(Node rootNode);
```

Beide Methoden bekommen den Huffman-Baum (also den Wurzelknoten) übergeben. Die Methode `buildEncoder` erstellt ein Objekt, das Daten mit Hilfe dieses Huffman-Baumes kodieren kann, `buildDecoder` eines, das Daten dekodieren kann. Es ist wichtig, dass die Implementierungen aufeinander abgestimmt werden, damit das Dekodieren von kodierten Daten die ursprünglichen Daten liefert.

`Encoder` und `Decoder` sind die folgenden Schnittstellen:

```
public interface Encoder {
    void writeData(int data, BitOutputStream out) throws IOException;
}
public interface Decoder {
    int readData(BitInputStream in) throws IOException;
}
```

Die Methode `writeData` nimmt einen Byte-Wert (eine Zahl zwischen 0 und 255) entgegen und wendet den Huffman-Code an. Die entstandene Bitfolge wird in den `BitOutputStream` geschrieben.

Die Methode `readData` liest so lange Bits aus dem gegebenen `BitInputStream` ein, bis ein Symbol im Huffman-Code erkannt wurde. Das erkannte Symbol (Zahl zwischen 0 und 255) wird zurückgegeben.

Erstellen Sie zunächst einige Testfälle. In `testExample2` können Sie sehen, wie Sie den für den Aufruf benötigten `BitOutputStream` und `BitInputStream` erstellen können: Wir haben Ihnen zum Testen zwei Klassen bereitgestellt, die nicht mit Dateien interagieren, sondern die Daten tatsächlich im Arbeitsspeicher ablegen. Die Größe des dazu verwendeten Speicherplatzes (in Bytes) wird dem `TestBitOutputStream` Konstruktor übergeben. Für kleinere Testfälle funktioniert das, zum Verarbeiten von mehrere Gigabyte an Daten wäre das ungeeignet. Die Idee ist, dass wir mit dem `TestBitOutputStream out` eine Bitfolge vom Programm empfangen und dabei mit `out.getWrittenBits()` die Anzahl der geschriebenen Bits regelmäßig überprüfen. Anschließend erzeugen wir mit `new TestBitInputStream(out.getWrittenBuffer())` einen `BitInputStream`, der die geschriebene Bitfolge wieder zum Einlesen bereitstellt. Wichtig ist, dass `out` nach dem Aufruf von `getWrittenBuffer` nicht mehr verwendet werden kann.

Überprüfen Sie mit verschiedenen Huffman-Bäumen und verschiedenen Daten, dass das Dekodieren von kodierten Daten die ursprünglichen Daten liefert.

Anschließend können Sie die Implementierung angehen. Sie können entweder Objekte einer anonymen Klasse direkt in den Methode `buildEncoder` und `buildDecoder` implementieren und zurückgeben, oder zwei Klassen `EncoderImpl implements Encoder` und `DecoderImpl implements Decoder` erstellen und Instanzen dieser Klassen in `buildEncoder` und `buildDecoder` zurückgeben. Die Konstruktoren von `EncoderImpl` und `DecoderImpl` sollten dann den Wurzelknoten des Huffman-Baumes entgegen nehmen.

Tipp 1: Dekodieren ist die einfachere Operation, beginnen Sie damit. Sie starten in der Wurzel des Baumes. Wenn Sie einen inneren Knoten haben, lesen Sie ein Bit ein das Ihnen sagt, ob Sie zum linken oder zum rechten Teilbaum weiter gehen müssen. Wenn Sie bei einem Blatt angekommen sind, können Sie die im Blatt gespeicherten Daten zurückgeben. Achten Sie darauf, dass der nächste Aufruf von `readData` wieder in der Wurzel starten muss.

Tipp 2: Zum Kodieren ist der Baum selbst nicht so hilfreich. Die Methode `writeData` bekommt den Wert übergeben, der in einem der Blätter enthalten sein muss. Aber Sie haben weder die Möglichkeit, schnell dieses Blatt zu finden, noch die Möglichkeit, vom Blatt ausgehend eine Bitfolge zu konstruieren, die den Pfad beschreibt. Daher sollten Sie zuerst (z.B. im Konstruktor und ggf. einer rekursiven Hilfsmethode) eine Code-Tabelle konstruieren, die für jeden möglichen Byte-Wert die Bitfolge enthält. Dann kann `writeData` einfach in dieser Tabelle nachschlagen. Nutzen Sie als Code-Tabelle ähnlich wie für die Häufigkeitstabelle ein Array: `boolean[][] encodings = new boolean[256][]`. Laufen Sie einmalig durch den gesamten Baum, merken sich dabei den bisher abgelaufenen Pfad als Bitfolge und tragen dann für jedes Blatt diese Bitfolge in das Array ein.

Hinweis: Die Implementierung dieser Teilaufgabe ist maßgeblich entscheidend für die Geschwindigkeit Ihres Kompressionsprogramms. Für jedes einzelne Byte der Eingabedatei wird die Methode ein mal aufgerufen, bei einer Datei mit 2 Gigabyte also 2.147.483.648 mal.

d) In dieser Teilaufgabe sollen Sie die folgenden Schnittstellen-Methoden testen und implementieren:

```
void writeNode(Node node, BitOutputStream out) throws IOException;
Node readNode(BitInputStream in) throws IOException;
```

Mit den bisherigen Teilaufgaben haben wir alle benötigten Mittel geschaffen, um aus einer Häufigkeitstabelle für eine Eingabedatei den Huffman-Baum zu erstellen und mit diesem dann die Datei zu kodieren. Zur Dekodierung wird allerdings der Huffman-Baum wieder benötigt. Wir könnten ihn zwar erneut aus der Häufigkeitstabelle erstellen, aber dazu brauchen wir ja wieder die originale Datei, die durch das Dekodierungsverfahren erst wiederhergestellt werden soll. Wir müssen den Baum also zusammen mit den kodierten Daten ablegen, damit er zum Zeitpunkt der Dekodierung wieder zur Verfügung steht.

Die Methode `writeNode` soll eine Repräsentation des gegebenen Knotens in den `BitOutputStream` schreiben. Umgekehrt soll die Methode `readNode` so lange Bits aus dem `BitInputStream` einlesen, bis ein vollständiger Knoten erkannt wurde und diesen dann zurückgeben. Die Implementierungen müssen also aufeinander abgestimmt sein, damit der ursprünglich an `writeNode` übergebene Knoten von `readNode` wiederhergestellt werden kann.

Erstellen Sie zunächst einige Testfälle. Sehen Sie sich dazu das Beispiel `testExample3` in der Vorlage an und beachten Sie die Hinweise bzgl. Testen aus der vorherigen Teilaufgabe. Überprüfen Sie mit verschiedenen Huffman-Bäumen, dass wenn ein Baum geschrieben und dann wieder eingelesen wird, der ursprüngliche Baum zurückgegeben wird.

Anschließend können Sie die Implementierung angehen. Es gibt verschiedene Möglichkeiten, einen Baum als Bitfolge zu repräsentieren. Damit unser Komprimierungsverfahren gut ist, muss die Darstellung möglichst klein sein. Sonst macht die Größe der Metadaten den Effekt der Datenkomprimierung zunichte. Ein mögliches Verfahren ist wie folgt[7]:

- Zum Schreiben überprüfen Sie zunächst, ob es sich um ein Blatt oder um einen inneren Knoten handelt. Schreiben Sie ein Bit, um diese Information zu speichern. Wenn es ein Blatt ist, dann schreiben Sie anschließend die Daten. Wenn es ein innerer Knoten ist, dann schreiben Sie durch rekursive Aufrufe nacheinander die beiden Teilbäume.
- Zum Einlesen gehen Sie genau umgekehrt vor. Lesen Sie zunächst ein Bit, um zu erfahren, ob Sie ein Blatt oder einen inneren Knoten erstellen müssen. Für ein Blatt lesen Sie anschließend die Daten (hier ist es sehr nützlich, dass die Daten immer genau ein Byte groß sind). Für einen inneren Knoten lesen Sie rekursiv nacheinander die beiden Teilbäume ein.

e) Stellen Sie sicher, dass Sie mit Ihren Tests aus den Aufgabenteilen b), c) und d) für alle von Ihnen selbst erstellten Klassen eine Instruction Coverage von 100% erreichen. Jede von Ihnen geschriebene Instruktion muss in den Tests also mindestens ein mal ausgeführt werden. Erweitern Sie Ihre Tests gegebenenfalls oder fügen Sie neue Tests hinzu.

[7] Die Idee stammt von hier: https://en.wikipedia.org/wiki/Huffman_coding#Decompression

Aufgabe 3 Datei-Kompressionsprogramm ausprobieren

Probieren Sie Ihr Datei-Kompressionsprogramm aus. Führen Sie das Gradle Task `jar` aus (in IntelliJ IDEA befindet es sich in der Rubrik `build`). Anschließend befindet sich im Ordner `build/libs` eine Datei `Huffman.jar`. Das ist Ihr fertiges Programm. Sie können es unabhängig vom Projekt irgendwo hin kopieren und dort ausführen – auch auf einem anderen Computer (sofern dort eine Java Runtime Environment installiert ist).

Navigieren Sie in einem Terminal-Fenster in den Ordner, in dem `Huffman.jar` gespeichert ist. Führen Sie dann den Befehl `java -jar Huffman.jar compress X Y` aus, wobei `X` der Name der zu komprimierenden Datei ist und `Y` der gewünschte Dateiname für die komprimierte Datei (darf noch nicht existieren). Auf der Konsole erscheinen einige Informationen (mehr dazu gleich) und die Ausgabedatei `Y` wird erstellt. Führen Sie anschließend `java -jar Huffman.jar decompress Y Z` aus, wobei `Y` die komprimierte Datei ist und `Z` der gewünschte Dateiname für die wiederhergestellte Datei (darf noch nicht existieren). Prüfen Sie, dass der Inhalt der wiederhergestellten Datei `Z` identisch zum Inhalt der ursprünglichen Datei `X` ist.

In der Ausgabe des Komprimierungsprogrammes sehen Sie die Häufigkeitstabelle für die eingelesene Datei, den erstellten Huffman-Baum sowie einige Informationen zu den geschriebenen Daten. Bei der Dekomprimierung wird Ihnen der eingelesene Huffman-Baum angezeigt, der natürlich identisch zu dem bei der Komprimierung sein muss.

Wir speichern neben dem Huffman-Baum auch die Größe der ursprünglichen Datei in der komprimierten Datei ab, damit man beim Dekomprimieren weiß, wann kein weiteres Zeichen mehr eingelesen werden darf. Das ist notwendig, weil unsere komprimierten Daten eventuell mit einzelnen Bits enden, Dateien aber nur ganze Bytes (Blöcke aus je 8 Bits) enthalten können. Somit befinden sich am Ende der komprimierten Datei ggf. zusätzliche Null-Bits, die nicht verarbeitet werden dürfen.

Aufgabe 4 Vorbereitung auf Projektabnahme

Bereiten Sie sich auf die Projektabnahme vor (siehe Deckblatt). In der Abnahme sollen neben Ihrem Code auch folgende Fragestellungen diskutiert werden. Bereiten Sie entsprechende Antworten vor:

- Was ist der höchste Kompressionsgrad, den Sie erreichen können? (Siehe Ausgabe des fertigen Programms, Aufgabe 3.)
- Wie sieht der Huffman-Baum aus, wenn nur ein einziger Byte-Wert verwendet wird (alle anderen haben die Häufigkeit 0)?
- Erstellen Sie eine Datei, die nur ‘`a`’ Zeichen enthält (ganz viele davon), aber keinen Zeilenumbruch am Ende. Komprimieren Sie die Datei und achten Sie darauf, dass die angezeigte Häufigkeitstabelle nur einen Eintrag hat. Wie gelingt hier die Komprimierung und anschließende Dekomprimierung?
- Wann ist eine Datei besonders gut bzw. besonders schlecht geeignet, um mit Huffman ein gutes Kompressionsergebnis zu erzielen?