Praktikum zur Lehrveranstaltung
Algorithmen und Datenstrukturen
WiSe 2023/24

HAW Hamburg
Department Informatik
L. Hamann

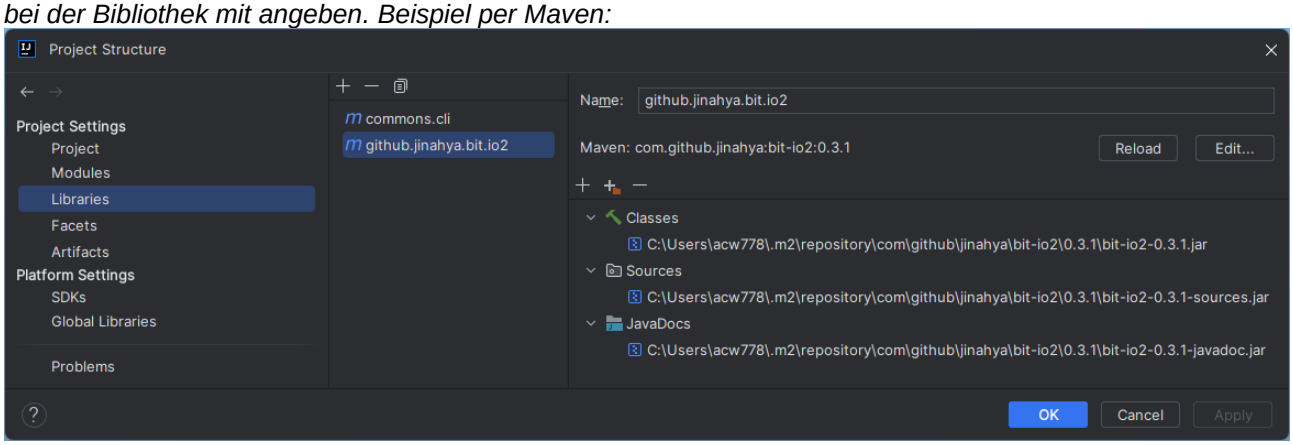
Aufgabenblatt 4
Stand: 15.12.2023

Bearbeiten Sie alle Aufgaben aller Übungsblätter – soweit nicht anders angegeben – in festen **Zweier**gruppen. Beide Gruppenmitglieder müssen die Aufgabe gemeinsam bearbeiten und die Lösung alleine erläutern können.

Vorgaben für alle Aufgaben
Im Praktikum müssen Sie zeigen, dass Sie die folgenden Dateien aus dem MS Teams Raum verlustfrei komprimieren und dekomprimieren können:
- ./Kursmaterialien/Praktikum1/List.use
- ./Kursmaterialien/ad_5_DynamischesProgrammieren.pdf

Weiterhin bin ich an den Größen der komprimierten Dateien interessiert. Diese also nicht automatisch löschen. Sie prüfen bitte die Kompressionsraten für typische Dateien: .txt, .json, .exe, .docx, … Geben Sie zusätzlich zu den oben vorgegebenen Dateien für mindestens drei Dateiarten die erreichten Werte für Ihre Implementierungen aus 4.1 und 4.2 in der Ausarbeitung in tabellarischer und/oder grafischer Form an.

Hinweis zum bitweisen Schreiben in Java
Für alle Aufgaben müssen Sie Daten bitweises einlesen und ausgeben. Nutzen Sie dafür die Bibliothek https://github.com/jinahya/bit-io2, da Java per se nur byteweise schreibt.
Diese können Sie z. B. per Maven oder manuell einbinden. Auf jeden Fall sollten Sie mindestens die Dokumentation einbinden. Bei Maven, indem Sie auch Doc und oder Source mit laden lassen oder manuell bei der Bibliothek mit angeben. Beispiel per Maven:



Dadurch werden Ihnen die Dokumentation der Klassen, Methoden etc. in der IDE angezeigt.
***Wichtig**: Bevor Sie den BitOutput-Stream schließen müssen Sie die letzten Bits per „align(1)“ herausschreiben. Diese werden sonst nicht die die Datei geschrieben und das Einlesen schlägt fehl!*

Beispiel:
```
try(OutputStream output = new FileOutputStream(outputFile)) {
    BitOutput bitOut = BitOutputFactory.from(output);
    …
    bitOut.align(1);
}
```

Aufgabe 4.1 – Huffman-Codierung
Implementieren Sie die Huffman-Codierung und -Dekodierung. Im Anhang finden Sie diese als Pseudocode, an dem Sie sich bitte orientieren!
Nutzen Sie für den Huffman-Tree Ihren in Aufgabe 3.1 implementierten `BinaryTree`.

Aufgabe 4.2 – Wahlweise LZSS oder LZW

Implementieren Sie das LZSS oder das LZW-Verfahren und bestimmen Sie die zum Algorithmus angegebenen Messwerte. Diese dokumentieren Sie ebenfalls in der Ausarbeitung.

Für LZSS:
Abweichend zur Vorlesung (LZSS), nehmen Sie die Codierung wie folgt vor:

Token für Phrase:	0/offset,equals	z. B. 0/12,4
Token für Literale:	1/character	z. B. 1/'+'

Also nach dem Token für die Phrase keinen Character mehr hinzufügen. Nutzen Sie für die Suche im Textfenster Ihren in Aufgabe 3.2 implementierten Binary Search Tree.

Experimentieren sie mit unterschiedlichen Werten für das Textfenster und den Lookahead. Bestimmen Sie dazu die folgenden Parameter:

- Anzahl Tokens für Phrasen und Literale
- Mittlere Zahl codierter Zeichen

Für LZW:
Verwenden Sie für die Implementierung Ihren in der Aufgabe 3.2 implementierten Binary Search Tree. Experimentieren Sie mit der Größe des Dictionary, z. B. 10 – 12 Bit (1024 bis 4096) Einträge. Bestimmen Sie zusätzlich zur Kompressionsrate noch die jeweilige mittlere codierte Stringlänge

Aufgabe für Neugierige: ersetzen Sie den Binary Search Tree durch eine HashMap und vergleichen Sie die Laufzeiten!

Viel Spaß bei der Umsetzung!

Lars Hamann

Huffman-Implementierung (Java-PseudoCode)

```
class BinaryTree<D> {
  D data;
  BinaryTree<D> left;
  BinaryTree<D> right;
}

class HuffNode {
  int char;
  int frequency;
}

class Huffman {
  static int chars  = 256

  int                  freq[chars]      = 0
  int                  codeTable[chars] = 0
  Heap<>               heap             = new Heap
  Stack<>              path             = new Stack
  BinaryTree<HuffNode> root

  /**
   * A) Encoding
   */
  void calculateCharacterFrequencies();
  void buildHuffmanTree();
  void calculateCodeFromHuffmanTree();
  void writeCharacterFrequencies();
  void encodeDataFile();
```

```
  /*
   * B) Decoding
   */
  void    readCharacterFrequencies();
  root  = buildHuffmanTree();
  count = readCharacterCount();
  void    decodeDataFile(root, count);
}
```

calculateCharacterFrequencies

```
for( ch ∈ inputStream) frequ[ch]++;

for( ch ∈ chars) {
   if (frequ[ch] > 0)
     heap.add(BinaryTree.makeTree(
               new HuffNode(ch, frequ[ch]));
}
```

buildHuffmanTree

```
while (heap.size() > 1) {
  {node1, node2}  =  heap.remove();

  heap.add(BinaryTree.makeTree(
    new HuffNode(ch,
      node1.data.frequ + node2.data.frequ, node1, node2));
}

root = heap.remove()
```

calculateCodeFromHuffmanTree

```
traverseHuffmanTree(root);


traverseHuffmanTree(BinaryTree aTree) {
  if (aTree.isLeaf()) {
    codeTable[aTree.data.char] = getPath();
    return;
  }

  Path.push(0);
  traverseHuffmanTree(aTree.left);
  path.top(1);
  traverseHuffmanTree(aTree.right);
  path.pop();
}
```

encodeDataFile

```
for( ch ∈ inputStream) {
  outputStream.writeBits(codeTable[ch]);
}
```

decodeDataFile

```
for (int i = 0; i < count; i++)
  decode(root);

decode(BinaryTree<HufNode> aTree)
  if (aTree.isLeaf()) {
        return aTree.data.char;
  if (readNextBit == 0)
    this.decode(aTree.left);
  else
    this.decode(aTree.right);
```