DM507 Algoritmer og datastrukturer

Forår 2022

Projekt, del III

Java version

Institut for matematik og datalogi
Syddansk Universitet

21. april, 2022

Dette projekt udleveres i tre dele. Hver del har sin deadline, således at arbejdet strækkes over hele semesteret. Deadline for del III er søndag den 22. maj kl. 23:59. De tre dele I/II/III er ikke lige store, men har omfang omtrent fordelt i forholdet 15/30/55. Projektet skal besvares i grupper af størrelse to eller tre.

Mål

Målet for del III af projektet er at lave dit eget værktøj til at komprimere filer. Komprimeringen skal ske via Huffman-kodning. Der skal laves to programmer: et til at kode/komprimere en fil, og et til dekode den igen.

Vær sikker på at du forstår Huffmans algoritme (Cormen et al. afsnit 16.3 indtil side 433) før du læser resten af denne opgavetekst.

Opgaver

Opgave 1

Du skal i Java implementere et program, der læser en fil og laver en Huffman-kodet version af den. Den præcise definition af input og output følger nedenfor. Programmet skal hedde `Encode.java`, og skal kunne køres sådan:

```
java Encode nameOfOriginalFile nameOfCompressedFile
```

Input-filen skal ses som en sekvens af bytes (8 bits). Hver byte udgør et tegn. Der er derfor $2^8 = 256$ mulige forskellige tegn i input, og vores alfabet har

således størrelse 256. Tegn kaldes i resten af opgaven for bytes. For at læse bytes fra en fil, skal man bruge `read`-metoden fra `FileInputStream` (eller evt. `BufferedInputStream`).[1] I Java repræsenteres bytes ved `int`'s, hvor man kun bruger værdierne 0 til 255,[2] og dette er typen af output for `read`-metoden fra `FileInputStream` (samt typen af input for `write`-metoden fra `FileOutputStream`). Når man har nået enden af filen, returnerer `read()` værdien -1 (i stedet for en værdi mellem 0 og 255).

> En byte er derfor otte bits i filer på disk, men er en `int` undervejs i Java-programmet.

Sammenhængen mellem `int`'s og bytes er som for det binære talsystem, dvs. som vist nedenfor. Denne sammenhæng kender `read()` og `write`, og man skal ikke selv beskæftige sig med den i Java-programmet. Man skal blot tænke på bytes som heltal mellem 0 og 255.

`int`	Byte
0	00000000
1	00000001
2	00000010
3	00000011
⋮	⋮
254	11111110
255	11111111

Programmet `Encode` skal virke således:

1. Scan inputfilen og lav en tabel (et array med 256 entries) over hyppigheden af de enkelte bytes (husk at bytes er `int`'s mellem 0 og 255, og kan bruges som indekser i arrays).
2. Kør Huffmans algoritme med tabellen som input (alle 256 entries, også dem med hyppighed nul[3]).
3. Konvertér Huffmans træ til en tabel (et array med 256 entries) over kodeord for hver af de mulige bytes (husk at bytes er `int`'s mellem 0 og 255, og kan bruges som indekser i arrays).
4. Skriv de 256 hyppigheder (dvs. 256 `int`'s) til outputfilen.

[1] Man skal ikke bruge en stream fra `Reader`-familien.

[2] Bemærk at der findes en datatype kaldet `byte` i Java, men denne er et signed 8-bit heltal i two's complement og har derfor ikke de rette egenskaber, så den skal *ikke* anvendes.

[3] Huffmans algoritme virker kun for alfabeter med mindst to tegn. Hvis man udelader tegn med hyppighed nul, vil filer med indhold af typen `aaaaaaa` give et alfabet af størrelse ét. Derfor dette krav.

5. Scan inputfilen igen. Undervejs find for hver byte dets kodeord (ved opslag i tabellen over kodeord), og skriv dette kodeords bits til outputfilen.

I ovenstående scannes inputfilen to gange. Dette er at foretrække frem for at scanne den én gang og derefter gemme dens indhold i et array til videre brug, eftersom RAM-forbruget derved stiger fra $O(1)$ til inputfilens størrelse (som kan være meget stor).

I punkt 3 skal man lave et rekursivt gennemløb (svarende til et inordergennemløb af et søgetræ) af Huffman-træet og derved generere alle kodeordene. Under gennemløbet vedligeholder man hele tiden et kodeord svarende til stien fra roden til den nuværende knude, og når man når et blad, kan dette kodeord gemmes i tabellen. Kodeord (som jo ikke er lige lange) skal i tabellen repræsenteres af `String`'s af `'0'` og `'1'` tegn. En sådan streng kan så efter et opslag i tabellen gennemløbes tegn for tegn og konverteres til bits, som skrives til outputfilen.

Huffman-kodning skal implementeres via en prioritetskø (jvf. Cormen et al. side 431). Hertil skal genbruges interfacet `PQ` fra del I, samt gruppens implementation `PQHeap` heraf. Også klassen `Element` skal genbruges, og skal her repræsentere deltræer genereret under Huffman-algoritmens kørsel. Derfor skal `data` i `Element` være et træ-objekt, og `key` skal være dets tilhørende hyppighed. Træ-objektet gemt i `data`-delen er et binært træ med en byte (`int` med værdi mellem 0 og 255) gemt i blade. Se figuren i bogen side 432.[4] Træobjekterne fra del II kan *ikke* direkte genbruges (vi skal f.eks. her bruge andre operationer på dem end søgetræsoperationer), men kan tjene som inspiration. Bemærk at `getData()` fra `Element` returnerer noget af typen/klassen `Object`, som man derefter skal type-castes[5] til den type/klasse, ens træobjekt har.

Den præcise specifikation af output programmet `Encode` er:

> Først 256 `int`'s (hvilket fylder $256 \cdot 32$ bits i alt), som angiver hyppighederne af de 256 mulige bytes i input, derefter de bits, som Huffman-kodningen af input giver.

Bemærk at for korte filer (eller lange filer, som ikke kan komprimeres væsentligt med Huffmans metode) kan output af `Encode` være lidt længere end den oprindelige fil. Man kan tænke sig mange måder at undgå eller begrænse denne situation på, men dette er ikke en del af projektet.

[4] Hyppigheden vist i knuderne i illustrationerne i bogen behøves ikke implementeret, kun rodens tal bruges i Huffmans algoritme og det bliver her i stedet gemt som `key` i `Element`

[5] Læs evt. om type-casting her. Bemærk at `Object` er en superklasse for alle klasser, herunder også træ-objekt klassen, som du definerer.

Når man skriver et kodeord i output, har man brug for at skrive bits én ad gangen. Der er i Javas bibliotek ikke metoder til at læse og skrive enkelte bits til disk (mindste enhed er en byte), men underviseren har udleveret et bibliotek (klasserne `BitOutputStream` og `BitInputStream`), som indeholder metoder, som kan gøre dette. Der er også metoder til at læse og skrive hele `int`'s (som fylder 32 bits), hvilket skal bruges, når man laver den første del af output (hyppighederne).

Bemærk at output fra `Encode` *ikke* kan læses med normale editors eller tekstbehandlingsprogrammer. De gemte Huffman-kodeord giver jo kun mening, når de fortolkes som Huffman-koderne fra jeres Huffman-træ, og dem kender andre programmer ikke.[6] Kun jeres `Decode` fra opgave 2 vil kunne læse den del af output. Samme problematik gør sig gældende for første del af output, som har gemt hyppighedstabellen med `writeInt(int i)` fra den udleverede klasse `BitOutputStream`. Denne del kan kun læses af `readInt()` fra den udleverede klasse `BitInputStream`, sådan som `Decode` skal starte med at gøre (se næste afsnit).

> **Recap:** I opgave 1 skal man bruge `read`-metoden fra Java bibliotekets `FileInputStream` til at læse bytes fra inputfilen (den originale fil). Man skal bruge metoderne `writeInt(int i)` og `writeBit(int b)` fra den udleverede klasse `BitOutputStream` til at skrive henholdsvis `int`'s (for hyppighedstabel) og bits (for Huffmanskoderne) til outputfilen (den komprimerede fil). Constructoren for en `BitOutputStream` skal have en `FileOutputStream` som argument.

Opgave 2

Du skal i Java implementere et program, som læser en fil med data genereret af dit program fra opgave 1, og som skriver en fil med det originale (ukomprimerede) indhold. Programmet skal hedde `Decode.java`, og skal kunne køres sådan:

```
java Decode nameOfCompressedFile nameOfDecodedFile
```

Programmet `Decode` skal virke således:

1. Indlæs fra inputfilen tabellen over hyppighederne for de 256 bytes.

[6] Andre programmer vil forsøge at fortolke bits som de plejer, f.eks. som utf8-, latin1- eller ASCII-encoded tekst. Det vil resultere i meningsløst output.

2. Generer samme Huffman-træ som programmet fra opgave 1 (dvs. *samme* implementation skal bruges begge steder, så der i situationer, hvor Huffman-algoritmen har flere valgmuligheder, vælges det samme).

3. Brug dette Huffman-træ til at dekode resten af bits i inputfilen, og imens som output skrive den originale version af filen. Dette gøres ved at bruge de læste bits til at navigere ned gennem træet (mens de læses). Når et blad nås, udskrives dets byte, og der fortsættes fra roden.

Alt dette kan gøres i ét scan af input.

Bemærk at det samlede antal bits fra udskrivningen af Huffman-koderne af det udleverede bibliotek om nødvendigt bliver rundet op til et multiplum af otte (dvs. til et helt antal bytes), ved at nul-bits tilføjes til sidst, når filen lukkes. Dette skyldes, at man på computere kun kan gemme filer, som indeholder et helt antal bytes. Disse muligt tilføjede bits må ikke blive forsøgt dekodet, da ekstra bytes så kan opstå i output. Derfor må man under dekodning finde det samlede antal bytes i originalfilen ved at summere hyppighederne, og under rekonstruktionen af den ukomprimerede fil holde styr på hvor mange bytes, man har skrevet.

For at skrive bytes til en fil, skal man bruge `write`-metoden fra `FileOutputStream` (eller evt. `BufferedOutputStream`).[7]

> **Recap:** I opgave 2 skal man bruge metoderne `readInt()` og `readBit()` fra den udleverede klasse `BitInputStream` til at læse henholdsvis `int`'s (for hyppighedstabel) og bits (for Huffmanskoderne) fra inputfilen (den komprimerede fil). Man skal bruge `write`-metoden fra Java bibliotekets `FileOutputStream` til at skrive bytes til outputfilen (den genskabte originale fil). Constructoren for en `BitInputStream` skal have en `FileInputStream` som argument.

Formalia

Du skal kun aflevere Java source-filer. Disse skal indeholde grundige kommentarer. De skal også indeholde navnene og SDU-logins på gruppens medlemmer. Dine programmer vil blive testet med mange typer filer (`txt`, `.doc`, `.jpg`,...), og du bør selv gøre dette inden aflevering, men du skal ikke dokumentere disse test.

[7] Man skal ikke bruge en stream fra `Writer`-familien.

Du skal aflevere `Encode.java` og `Decode.java` samt alle andre `.java` filer, som kræves for at køre dem, f.eks. `PQHeap.java`, `PQ.java` og `Element.java` fra del I. Filerne skal enten afleveres som individuelle filer eller som ét `zip`-arkiv. Der må ikke være `package` statements i filerne

Filerne skal afleveres elektronisk i itslearning i mappen Afleveringsopgaver under faneblad Ressourcer. Afleveringsmodulet er også sat ind i en itslearning plan i kurset.

Under afleveringen skal man erklære gruppen ved at angive alle medlemmernes navne. Man skal kun aflevere én gang per gruppe. Bemærk at man under aflevering kan oprette midlertidige "drafts", men man kan kun aflevere *én gang*.

Aflever materialet senest:

Søndag den 22. maj, 2022, kl. 23:59.

Bemærk at aflevering af andres kode eller tekst, hvad enten kopieret fra medstuderende, fra nettet, eller på andre måder, er eksamenssnyd, og vil blive behandlet særdeles alvorligt efter gældende SDU regler. Man lærer desuden heller ikke noget. Kort sagt: kun personer, hvis navne er nævnt i den afleverede fil, må have bidraget til koden.