

Suunnitteludokumentti

Ohjelma koostuu kahdesta pääluokasta: Pakkaaja ja Purkaja. Pakkaaja pakkaa annetun tiedoston ja Purkaja purkaa sen. Pakkausvaiheessa syntyy lisäksi ylimääräinen header-tiedosto johonka tallennetaan blokki-huffman-koodi -parit. Tämän nimi on sama kuin pakatun tiedoston nimi + .header-pääte (esim. pakattu.dat + pakattu.dat.header).

Tiedostokäsittely hoituu ITiedostoLukija ja ITiedostoKirjoittaja-rajapintojen kautta koska tämä mahdollistaa yksikkötesteissä korvaavien objektien käytön jotka teeskentelevät olevansa tiedostoja.

Pakkaajan toiminta & rakenne

Pakkaajalle määritetään blokkikoko eli kuinka monen tavun rypäissä pakkaaja lukee lähdetiedostoa. Blokeille muodostetaan kanonisoidut huffman-koodit joiden avulla tiedosto pakataan. Pakkauksen yhteydessä syntyy header-tiedosto jonka rakenne on määritelty tarkemmin tiedostoformaatti.txt-tiedostossa.

Pakkaaja muodostaa ensiksi huffman-koodit KoodiMuodostaja-luokan avulla. KoodiMuodostaja lukee ITiedostoLukija-objektin avulla tiedostosta blokkeja jotka järjestetään esiintymistiheyden mukaan. Näistä blokeista muodostetaan huffman-puu, josta generoidaan huffman-koodit. Huffman-koodit kanonisoidaan vielä Kanonisoija-objektin avulla. Kanonisoiduilla koodeilla on se etu, että ne voidaan muodostaa pelkästään koodin pituustiedon mukaan kunhan ne on tallennettu headeriin oikeassa järjestyksessä. Tämä mahdollistaa sen että headeriin voidaan tallentaa vain koodin pituus eikä itse koodia tarvitse tallentaa.

Kun kanonisoidut koodit on muodostettu, välitetään koodit tiivistäjä-objektille. Objekti lukee lähdetiedostosta ITiedostoLukijan kautta blokkeja ja tallentaa uuteen tiedostoon blokeille määritettyjä koodeja ITiedostoKirjoittaja-objektin avulla. Tiivistäjä palauttaa tiedon siitä montako bittiä on merkitseviä viimeisessä tavussa. Tämä on tarpeellinen tieto koska tiedostonkäsittely hoituu tavutasolla ja koodit kirjoitetaan bittitasolla – tämän takia todennäköisesti syntyy tilanne että viimeisessä tavussa on käyttämättömiä bittejä joiden arvot ovat roskaa.

Kohdetiedosto kirjoittamisen jälkeen välitetään vielä HeaderMuodostaja-objektille koodit järjestettynä kasvavaan pituusjärjestykseen ja tieto siitä montako bittiä on merkitseviä viimeisessä tavussa.

Purkajan toiminta & rakenne

Purkaja lukee ensiksi header-tiedostosta blokit ja koodien pituudet HeaderLukija-luokan avulla (tarkempi formaatti tiedostoformaatti.txt-tiedostossa). Koska koodit on kanonisoitu, pelkkä tieto koodin pituudesta riittää muodostamaan koodit. Koodi – blokkiparit tallennetaan hajautustauluun joka välitetään PurkuKoodi-luokalle (sivuhuomio; sopivien nimien keksiminen on vaikeaa) joka lukee pakattua tiedostoa tavu kerrallaan. Tavuja puolestaan luetaan bitti kerralla ja jokaisen luvun jälkeen tarkistetaan löytyykö tähän asti kasattua koodia hajautustaulusta. Jos vastine löytyy, tallennetaan koodiin sidottu blokki uuteen tiedostoon. Tämä operaatio toistetaan kunnes tiedosto on purettu.

Parannettavaa

Ohjelman tietorakenteet ovat aika rajoitettuja – olen valtaosin implementoinut vain ne metodit mitä tarvitsen; esimerkiksi hash mapista ei voi poistaa avaimia. Lisäksi prioriteettijono on erittäin tehoton, kuluttaen helposti 8 – 10 kertaa enemmän aikaa samojen operaatioiden tekemiseen kuin vastaava javan oma tietorakenne. Tämä on erittäin ikävää koska prioriteettijonoa käytetään ohjelman vaativimmassa vaiheessa, koodien muodostamisessa, jonka aikavaativuus on $O(n \log n)$. Näin hidas prioriteettijono hidastaa varsinkin suurilla uniikkien koodien määrillä merkittävästi ohjelman toimintaa.

Pakkaustehokkuutta voisi parantaa myös järkevämällä blokkien muodostamisella. Nyt ohjelma lukee tiedostosta kiinteämittaisia blokkeja välittämättä tiedoston rakenteesta. Tämä johtaa siihen että varsinkin rakenteellisessa tiedossa (kuten xml) usein toistuva tieto ei pakkaudu kunnolla koska blokit voivat silpoa toistuvia tietorakenteita. Esimerkiksi `<tag>` - tyyppiset rakenteet voivat hajota useampaan erilaiseen yhdistelmään kuten `<t, ta, ag, g>` jne. Suurempi määrä uniikkeja koodeja kasvattaa myös headerin kokoa mikä entisestään heikentää pakkaustehokkuutta.

Headerformaattia voisi myös hieman tiivistää. Nyt headeriin tallennetaan koodin pituus yhteen tavuun toteutuksen helpottamiseksi, mutta huffman-koodin pituus on rajattu 64 bittiin. Näin headeriin pituuden tallentamiseksi tarvittaisiin vain 6 bittiä eikä 8 bittiä mitä nyt käytetään. Lisäksi 64 bittiä on melko paljon koodin pituuden määrittämiseksi; 32 bitilläkin saisi jo ~4,3 miljardia uniikkia koodia mikä riittäisi jo aika pitkälle; speksaamalla koodi pienemmäksi säästettäisiin headersissa vielä yksi bitti per huffman-koodi.

Lähteet

http://en.wikipedia.org/wiki/Huffman_coding – sivu tarkistettu 1.8.2013

https://en.wikipedia.org/wiki/Canonical_Huffman_code - sivu tarkistettu 12.8.2013

Introduction To Algorithms, Third edition sivut 428 – 435 – Thomas H. Cormen et al.

Hashausalgoritmi arraylistiin ja TiedostoBlokki-luokkaan:

http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx - tarkistettu 12.8.2013

Pakkaajan aika- ja tilavaativuudet

KanonisoidunKoodinMuodostaja-luokka

muodostaKanoninenHuffmanKoodi() - pseudokoodi

```
++koodi;  
if (nykyinenPituus > vanhaPituus  
    koodi = siiräBittejäOikealle(koodi, nykyinenPituus – vanhaPituus)  
    vanhaPituus = nykyinenPituus  
for (int i = 0; i < nykyinenPituus; ++i)  
    asetaBitti(uusiKoodi, haeBitti(vanhakoodi(uusiPituus – 1 – i ))  
  
return uusiKoodi
```

Alussa vakioaikaisia operaatioita vakiomäärä, aikavaativuus $O(1)$. Silmukan aikavaativuus on $O(m)$ missä m on koodin bittien määrä. Maksimipituus on määritetty 64 bittiin jolloin silmukka suoritetaan pahimmillaan 64 kertaa \rightarrow vakioaikainen $\rightarrow O(1)$. Bittioperaatiot ovat vakioaikaisia. Aikavaativuus $O(1)$

Metodissa luodaan muutama väliaikaismuuttuja, tilavaativuus $O(1)$

Metodin aikavaativuus on siis **$O(1)$** .

Metodin tilavaativuus on siis **$O(1)$** .

Kanonisoija-luokka

kanoniso() - pseudokoodi

```
avaimet = koodit.avaimet()  
sort(avaimet)  
for (i : avaimet)  
    listaBlokkiKoodiPareista = koodit.get(i)  
    for (pari : listaBlokkiKoodiPareista)  
        uusipari = Pari(pari.blokki, muodostaKanoninenHuffmanKoodi(pari.koodi))  
        kanonisoidutkoodit.put(uusipari)  
        järjestetytKoodit.add(uusiPari)  
  
return kanonisoidutkoodit
```

Aluksi avaimet sortataan heap sortilla $\rightarrow O(m \log m)$. Koska avaimena toimii koodin pituus mikä on rajattu välille 1 – 64, on sortattavia lukuja korkeintaan 64 kappaletta $\rightarrow O(64 * \log 64) \rightarrow O(1)$. Kartassa on arvona listoja joissa on blokki-koodi-parit. Kahden for-loopin avulla nämä käydään läpi \rightarrow aikavaativuus $O(n)$, verrannollinen parien määrään. Koodi kanonisoidaan joka on $O(1)$ -operaatio ja tallennetaan uuteen karttaan ja listaan joka pidetään järjestyksessä headermudostusta varten. Aikavaativuudet $O(1) \rightarrow$ Kokonaisaikavaativuus on $O(n)$

Tilaa vaaditaan uudelle kartalle $O(n)$ ja listalle $O(n) \rightarrow$ tila-vaativuus $O(n)$

Metodin aikavaativuus on siis **$O(n)$** .

Metodin tilavaativuus on siis **$O(n)$** .

KoodiMuodostaja-luokka

laskeEsiintymisTiheydet() - pseudokoodi

```
while (ei_olla_tiedoston_lopussa)
    luettu_tavuja = lueTiedostosta(max_blokin_ verran_tavuja)
    tiedostoBlokki.tavut = kopioiLuetutTavut
    if (!esiintymistiheydet.contains(tiedostoBlokki)
        esiintymistiheydet.put(tiedostoBlokki, 1)
    else
        esiintymistiheydet.put(tiedostoBlokki, hashmap.get(tiedostoBlokki) + 1)
return esiintymistiheydet
```

Tämän metodi käy tiedoston jossa n tavua läpi – tiedot on luettava eli käytävä ainakin kerran läpi -> aikavaativuus $O(n)$. Lisäksi tiedot kopioidaan uuteen taulukkoon koska luettujen tavujen määrä voi erota blokin koosta -> $O(n)$ operaatiota. Tämän jälkeen blokkia käytetään hashmapin avaimena contains-operaatiossa joka on hashmapille $O(1)$, ja tämän jälkeen kirjoitetaan mappiin joka on $O(1)$.

Tilan kannalta pahin tapaus on jos tiedosto on huonosti pakkautuva (esimerkiksi jo pakattu tiedosto) ja blokkikoko on tarpeeksi suuri että jokainen luettu blokki on uniikki (esimerkiksi blokkikoko 5 = $\max 2^{(5 \cdot 8)}$ uniikkia blokkia; jokainen blokki voi olla uniikki jos tiedostokoko alle ~ 5 gigatavua). Tällöin blokkeja on verrannollisesti tiedoston tavumäärään -> $O(n)$.

Metodin aika- ja tilavaativuus on näin **$O(n)$** .

muodostaPrioriteettiJono() - pseudokoodi

```
avaimet = esiintymistiheydet.avaimet()

for (avain : avaimet)
    treenode.avain = esiintymistiheydet.get(avain)
    treenode.arvo = avain
    treenode.vasenlapsi = treenode.oikealapsi = null
    prioriteettijono.push(TreeNode(esiintymistiheys, blokki))

return prioriteettijono
```

Metodissa käydään ensiksi koko hashmap läpi mikä on lineaarinen operaatio. Tämän operaation aikavaativuus on $O(n)$ missä n on tiedoston tavumäärä koska pahimmillaan jokainen luettu blokki esiintyy vain kerran. Jokaisella kierroksella luodaan uusi treenode joka koostuu kolmesta vakio-operaatiosta -> $O(1)$. Tämän jälkeen tallennetaan node prioriteettijonoon joka on sisäisesti heap. Tämä aiheuttaa heapify-operaation jonka aikavaativuus on $O(\log n)$ (<http://www.cs.helsinki.fi/u/floreen/tira2013/tira.pdf> – sivu 345). Metodissa suoritetaan siis n kertaa $\log n$ – operaatio -> metodin aikavaativuus $O(n \log n)$

Prioriteettijonoon tallennetaan kaikki blokit. Tämä vie $O(n)$ tilaa.

Metodin aikavaativuus on siis **$O(n \log n)$** .

Metodin tilavaativuus on siis **$O(n)$** .

muodostaHuffmanPuu() - pseudokoodi

```
while (!prioriteettijono.tyhja())
    silmu1 = prioriteettijono.pop()
    silmu2 = prioriteettijono.pop()
    uusiSilmu.vasenlapsi = silmu1
    uusisilmu.oikealapsi = silmu2
    uusisilmu.avain = silmu1.avain + silmu2.avain
    uusisilmu.arvo = null
    prioriteettijono.push(uusinode)

return prioriteettijono.pop()
```

Metodissa muodostetaan huffman-puu. Joka kierroksella poistetaan kaksi alkia prioriteettijonosta ja lisätään yksi -> nettona vähenee yhdellä per kierros -> aikavaativuus $O(n)$. Uuden solmun muodostus onnistuu vakioajassa jolloin koko metodin aikavaativuus on $O(n)$.

Puuhun luodaan $O(n)$ uutta solmua ja lehtisolmuja on $O(n)$ -> tilavaativuus $O(2n) = O(n)$

Metodin aikavaativuus on siis **$O(n)$** .

Metodin tilavaativuus on siis **$O(n)$** .

käyPuuLäpiRekursiivisesti() - pseudokoodi

```
if (node.onLehti)
    koodi(kasattuKoodi, pituus)

    if (!koodiMap.contains(pituus))
        koodiMap.put(pituus, new OmaArrayList())
    koodiMap.get(pituus).add(Pari(pituus, koodi))

return
```

```
käyPuuLäpiRekursiivisesti(node.vasenlapsi)
käyPuuLäpiRekursiivisesti(node.oikealapsi)
```

Metodissa käydään puu läpi rekursiivisesti. Jokainen puun solmu käydään läpi -> $O(n)$ operaatiota. Lisäksi kun saavutetaan juurisolmu, lisätään kasattu koodi karttaan jos on jo lista kartassa koodin pituudelle ajassa $O(1)$, tai luodaan uusi lista jos sitä ei ole olemassa ennen lisäystä $O(1)$ -> Aikavaativuus $O(n)$.

Tilan kannalta puussa edetään alaspäin kunnes osutaan juuren, jolloin rekursiopino on pahimmillaan puun syvyinen -> $O(\log n)$. Lisäksi kasataan tauluun kaikille blokeille koodit -> tilavaativuus $O(n)$ koska jokainen blokki on taulussa.

Metodin aikavaativuus on siis **$O(n)$** .

Metodin tilavaativuus on siis **$O(n)$** .

muodostaKoodit() - pseudokoodi

```
esiintymisTiheydet = laskeEsiintymisTiheydet()
prioriteettijono = muodostajono(esiintymistiheydet)
puu = muodostaHuffmanPuu(prioriteettijono)
koodilista = käyPuuLäpiRekursiivisesti()
return kanonisoiija.kanonisoi(koodilista);
```

Aikavaativuus: $O(n) + O(n \log n) + O(n) + O(n) \rightarrow O(n \log n)$

Tilavaativuus: $O(n) + O(n) + O(n) + O(n) + O(n) \rightarrow O(n)$

Metodin aikavaativuus on siis **$O(n \log n)$** .

Metodin tilavaativuus on siis **$O(n)$** .

Tiivistaja-luokka

tiivista()-pseudokoodi

```
while (ei_olla_tiedoston_lopussa)
    lueTiedostosta(blokki, max_blokin_verran_tavuja)
    pakkauskoodi = koodit.get(blokki)
    kopioiBittejaTavuun(tavu, blokki)
    if (tavu_taynna)
        tallennaTiedostoon(tavu)
```

tallennaviimeinenTavu

return viimeisessaTavussaMerkitseviaBitteja

Metodi lukee tiedostosta kaikki sen tavut $\rightarrow O(n)$ aikavaativuus. Hashmapping get on vakioaikainen, bittejä kopioidaan korkeintaan 8 \rightarrow vakioaikainen operaatio. Tallennetaan $O(n)$ tavua. Viimeisen tavun tallennus vaatii vakiomäärän operaatioita. Aikavaativuus siis $O(n)$

Metodi luo muutaman väliaikaismuuttujan \rightarrow tilavaativuus $O(1)$

Metodin aikavaativuus on siis **$O(n)$** .

Metodin tilavaativuus on siis **$O(1)$** .

HeaderMuodostaja-luokka

muodostaHeader()-pseudokoodi

```
tallennaTiedostoon(blokinPituus)
tallennaTiedostoon(merkitsevienBittienMaaraViimeisessaTavussa)
```

```
for (blokkiKoodiparit : koodit)
    if (blokkiKoodiParit.blokit.pituus != blokinPituus)
        tallennaTiedostoon(0)
        tallennaTiedostoon( blokkiKoodiParit.blokki.pituus)

    tallennaTiedostoon( blokkiKoodiParit.koodi.pituus)
    tallennaTiedostoon(blokkiKoodiParit.blokki.blokki)
```

for-luupin aikavaatimus on $O(n)$ koska koodeja on pahimmillaan $O(n)$. Muut operaatiot ovat vakioaikaisia $\rightarrow O(1)$

Metodi luo väliaikaismuuttujia muutaman $\rightarrow O(1)$ tila-vaativuus

Metodin aikavaativuus on siis **$O(n)$** .

Metodin tilavaativuus on siis **$O(1)$** .

Pakkaaja-luokka

pakkaa()-pseudokoodi

Pseudokoodi:

```
koodit = koodimuodostaja.muodostakoodit(lukija(lähdetiedosto))
bittejaKaytetty = tiivistaja.tiivista(lukija(lähdetiedosto), kirjoittaja(kohdetiedosto), koodit)
headermuodostaja(kirjoittaja(kohdetiedosto + ".header"),
    koodiMuodostaja.kooditJarjestyksessa(), bittejaKaytetty, BlokinKoko))
tulostaStatsit()
```

Aikavaativuus: $O(n \log n) + O(n) + O(n) = O(n \log n)$

Tilavaativuus $O(n)$ koodeille

Pakkauksen aikavaativuus on siis **$O(n \log n)$** missä n on tiedoston tavujen määrä

Pakkauksen tilavaativuus on siis **$O(n)$** missä n on tiedoston tavujen määrä.

Purkajan aika- ja tilavaativuudet

HeaderLukija-luokka

lueHeaderAlku()-pseudokoodi

```
lueTiedostosta(puskuri, 2) // luetaan 2 tavua  
merkitsevienBittienMaara = puskuri[0]  
blokinKoko = puskuri[1]
```

Luetaan tiedostosta kaksi ensimmäistä tavua; ensimmäinen tavu kertoo montako merkitsevää bittiä on pakatun tiedoston viimeisessä tavussa. Toinen tavu kertoo mikä on oletusblokkikoko pakatussa tiedostossa. Metodissa on vakiomäärä operaatioita ja apumuuttujia -> tila- ja aikavaativuus $O(1)$

Metodin aikavaativuus on siis **$O(1)$** .

Metodin tilavaativuus on siis **$O(1)$** .

lueKoodiBlokkiParit()-pseudokoodi

```
while (ei_olla_tiedoston_lopussa)  
    blokinPituus = oletusPituus // mikä luettiin headerin alusta  
    lueTiedostosta(koodinPituusBiteissa, 1) // luetaan tavu  
  
    if (koodinPituusBiteissa == 0) // erikoisblokki  
        lueTiedostosta(blokinPituus, 1)  
        lueTiedostosta(koodinPituusBiteissa, 1) // oikea pituus  
  
    lueTiedostosta(blokki, blokinPituus)  
    huffmanKoodi.pituus = koodinPituusBiteissa  
    huffmanKoodi.koodi = muodostaKanoninenHuffmanKoodi(koodinPituusBiteissa)  
    blokkiKoodiParit.put(huffmanKoodi, blokki)  
  
return blokkiKoodiParit
```

While-loopin aikana tiedosto luetaan kertaalleen läpi -> $O(m)$ operaatiota missä m on headerin pituus tavuissa. Lisäksi on muutama vakio-operaatio mukaanlukien kanonisoidun koodin muodostus ja arvojen asettaminen hashmappiin -> aikavaativuus $O(m)$. Tilavaativuuden kannalta taas hashmappiin tallennetaan kaikki tiedostosta luetut blokit ja blokeille generoidut huffman-koodit. Blokkien määrä on verrannollinen tiedoston pituuteen (koska tiedosto lähinnä koostuu blokeista) -> tilavaativuus $O(m)$

Metodin aikavaativuus on siis **$O(m)$** .

Metodin tilavaativuus on siis **$O(m)$** .

lueHeader()-pseudokoodi

```
lueHeaderinAlku  
return lueKoodiBlokkiParit
```

Kutsuu lähinnä yllä listattuja metodeja. Tila- ja aikavaativuus määrittyy lueKoodiBlokkiParit-metodista -> $O(m)$

Metodin aikavaativuus on siis **$O(m)$** .

Metodin tilavaativuus on siis **$O(m)$** .

PurkuKoodi-luokka

kasitteleTiedosto()-pseudokoodi

```
while (ei_olla_tiedoston_lopussa)  
    lueTiedostosta(tavu, 1)  
    merkitseviaBittejaTavussa = 8  
  
    if (viimeinenTavu)  
        merkitseviaBittejaTavussa = viimeisessaTavussaMerkitseviaBitteja)  
  
    for (kasiteltavaBitti : merkitseviaBittejaTavussa)  
        kasiteltavaKoodi.koodi = liitaBitti(tavu, kasiteltavaBitti)  
        kasiteltavaKoodi.pituus++  
  
    if (blokkiKoodiParit.contains(kasiteltavaKoodi))  
        kirjoitaTiedostoon(blokkiKoodiParit.get(kasiteltavaKoodi))  
        kasiteltavaKoodi.pituus = 0  
        kasiteltavaKoodi.koodi = 0
```

Metodi lukee pakattua tiedostoa tavu kerrallaan, ja jokainen tavu käsitellään bitti kerrallaan. Jos bittikoodille löytyy vastaavuus blokkiKoodiParit-hashmapista, kirjoitetaan blokki purettavaan tiedostoon. Tämän operaation aikavaativuus on $O(n)$ missä n on pakatun tiedoston tavumäärä – tiedosto luetaan kerran ja kirjoitettavien tavujen määrä on verrannollinen luettuihin tavuihin; keskimäärin jokaista x bittiä kohden on olemassa blokki joka tulee kirjoittaa. Muut operaatiot ovat vakioaikaisia – for loop suoritetaan max 8 kertaa. Näin ollen aikavaativuus on $O(n)$. Tilavaativuuden kannalta luodaan muutama muuttuja mutta näiden määrä on vakio – tilavaativuus näin $O(1)$.

Metodin aikavaativuus on siis **$O(n)$** .

Metodin tilavaativuus on siis **$O(1)$** .

Purkaja-luokka

pura()-pseudokoodi

```
blokkiKoodiParit = lueHeader()  
kasitteleTiedosto(blokkiKoodiParit)  
tulostaStatsit()
```

Aikavaativuus: $O(m) + O(n) \rightarrow O(m + n)$

Tilavaativuus: $O(m) + O(1) \rightarrow O(m)$

Metodin aikavaativuus on siis **$O(m + n)$** missä m on headerin koko tavuissa ja n on pakatun tiedoston koko tavuissa.

Metodin tilavaativuus on siis **$O(1)$** .