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. 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.

Pakkaajan aika- ja tilavaativuudet

KanonisoidunKoodinMuodostaja-luokka

muodostaKoodi() - 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 -> vakioaikainen -> 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

kanonisoi() - pseudokoodi

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

return kanonisoidutkoodit
```

Aluksi avaimet sortataan heap sortilla -> O(m log m). Koska avaimena toimii koodin pituus mikä on rajattu välille 1 – 64, on sortattavia lukuja korkeintaan 64 kappaletta -> O(64 * log 64) -> O(1). Kartassa on arvona listoja joissa on blokki-koodi-parit. Kahden for-loopin avulla nämä käydään läpi -> 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) -> Kokonaisaikavaativuus on O(n)

Tilaa vaaditaan uudelle kartalle O(n) ja listalle O(n) -> 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)
        byteWrapper.tavut = kopioiLuetutTavut
        if (!esiintymistiheydet .contains(byteWrapper)
                esiintymistiheydet.put(byteWrapper, 1)
        else
                esiintymistiheydet.put(bytewrapper, hashmap.get(bytewrapper) + 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 blokkikoko on 1 ja jokainen blokki on uniikki. Tällöin blokkeja on yhtä monta kuin tiedostossa on tavoja -> tilavaativuus 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. Tämän operaation aikavaativuus on O(n). 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 alkiota 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 kanonisoija.kanonisoi(koodilista);
```

Aikavaativuus: O(n) + O(n log n) + O(n) + O(n) -> O(n log n)
Tilavaativuus: O(n) + O(n) + O(n) + O(n) + O(n) -> 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 -> O(n) aikavaativuus. Hashmaping get on vakioaikainen, bittejä kopioidaan korkeintaan 8 -> vakioaikainen operaatio. Tallennetaan O(n) tavua. Viimeisen tavun tallennus vaatii vakiomäärän operaatioita. Aikavaativuus siis O(n)

Metodi luo muutaman väliaikaismuuttujan -> 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 -> O(n)

Metodi luo väliaikaismuuttujia muutaman -> 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.