

Toteutusdokumentti

Ohjelman yleisrakenne

Ohjelman yleisrakenne on yksinkertainen ja selkeä. Se sisältää pakkaukset "Apuvalineet", "Tietorakenteet" ja "Toteutus". Toteutus -pakkauksen alla on pakkaus "Huffman", jonka alla vielä pakkaukset "Pakkaaminen" ja "Purkaminen". Näiden alla on luonnollisesti niihin kuuluvia luokkia.

Apuvalineet sisältää mm. luokat tiedoston lukemiseen ja kirjoittamiseen, Tietorakenteet taas minimikeon ja hajautustaulun ja Huffman HuffmanPuun.

Näiden lisäksi on olemassa testipakkaukset ja -luokat, jotka on nimetty samalla lailla.

Saavutetut aika- ja tilavaativuudet

(täyd.)

Suorituskykyvertailua -ja testausta

9.6. klo 12:00

Alkuperäinen teksti tyyppiä:

"Ensimmäinen runo

Mieleni minun tekevi, aivoni ajattelevi
lähteäni laulamahan, saa'ani sanelemahan,
sukuvirttä suoltamahan, lajivirttä laulamahan.
Sanat suussani sulavat, puhe'et putoelevat,
kielelleni kerkiävät, hampahilleni hajoovat."

Alkuperäisen tekstin koko: 550kt

Tekstin pakkaamisen viemä aika vaihtelee n. 2s - 3s.

Pakatun tekstin koko: 432kt

Purkamisen viemä aika: 2.2s

Puretun tekstin koko: 556kt

Tyyppiä:

“Ensimmäinen runoMieleni minun tekevi, aivoni ajattelevilähteäni laulamahan, saa'ani sanelemahan,sukuvirttä suoltamahan, lajivirttä laulamahan.Sanat suussani sulavat, puhe'et putoelevat,kielelleni kerkiävät, hampahilleni hajoovat.”

10.6. klo 16:20

Nyt tiedoston pakkaamisen ja purun jälkeen alkuperäisteksti näyttää tekstieditorilla täysin samalta kuin alkuperäinenkin. Ero kuitenkin löytyy: ohjelmani koodaa ääkköset (eli tässä tapauksessa merkit “Ä, Ö, ä ja ö” 2 tavun unicode -formaatisissa alkuperäisen tekstin 1 tavun formaatin sijaan). Tästä johtuen kun alkuperäistekstin koko on n. 550kt, pakkaamisen ja purun jälkeen koko on 571kt. Muita ongelmia tässä ei näyttäisi tapahtuvan.

Pakatun tiedoston koko oli nyt 448kt ja tilankäyttö alkuperäiseen tekstiin verraten 81% ja purettuun tiedostoon 78%.

Huffman

Pakkaaminen

Tekstin lukeminen pakkaamisen alussa vie pienellä syötteellä (3kt) 200ms ja isolla syötteellä (550kt) n. 1200ms.

Huffman -puun sekä puun määrittämien merkkien bittiesitysten muodostaminen on nopea operaatio ja vie kaikenkokoisilla syötteillä vain vähän aikaa (enimmillään n. 10ms). Suurin kokeilemani tiedosto oli kalevalan .txt versio ja sillä tämä vei max. 7ms (puu) + 1ms (bittiesitykset).

Pakattava sisällön muodostaminen viei suhteellisen paljon aikaa. Pienillä syötteillä (< 70kt) pakkaus aika on alle 150ms ja isommilla syötteillä (550kt) yli 600ms. Tämä käyttää selvästi enemmän aikaa kuin muut ohjelman osat, joten testaani sen metodien käyttämää suoritusaikaa kattavammin ja selvitan, mikä luokan metodeista vie eniten aikaa.

Pakattavan sisällön kirjoittaminen ei vienyt aikaa kovinkaan paljon. Suurilla syötteillä (550kt) aikaa kuluu hieman yli 30 millisekuntia.

Kokonaisajat pakatessa:

lorem ipsum (67kt)	n. 700ms
kalevala (550kt)	n. 1900ms

Pakkaamisen kokonaisaika riippuu siis suureksi osaksi tekstin lukemiseen sekä pakatun sisällön muodostamiseen kuluva ajasta. Tekstin lukua voi olla hankala nopeuttaa, sillä se toimii suhteellisen hyvin optimoidulla yksinkertaisella algoritmilla, mutta pakatun sisällön muodostaminen sen sijaan on monivaiheinen operaatio, mikä voi sisältää hitaita osuuksia.

Pakatun sisällön muodostaminen

Koko suoritusaika kuluu käytännössä sen pakkauksen osan muodostamiseen, joka sisältää ainoastaan tekstin ascii -merkkeinä. Tähän operaatioon kuluu kaksi osaa, joista toinen vie kalevala.txt:n kanssa suoritusaajasta n. 140ms ja toinen n. 500ms.

Nopeampi näistä operaatioista on pakkauksen tekstin kelaaminen merkki merkiltä läpi, ottaen omatekoisesta hajautustaulusta merkkiä vastaava avain ja lisäten tämä avain StringBuilder-oliolle.

Hitaampi operaatio taas käyttää tästä nopeammasta operaatiosta saatua StringBuilderin sisältöä (11001...) ja kelaat sitä 8 merkin pätkinä muodostaen jokaisesta pätkästä ascii -merkin ja lisäten tämän toisella StringBuilder -oliolle. Operaatio on todennäköisesti hidas muutamien "turhien" algoritmien takia, jotka jouduin koodaamaan, koska Javaa käyttämällä oli ongelmia tehdä saada aikaan ascii -merkki mistä tahansa kokonaisluvusta ("char" int) saattoi palauttaa jotain ihan muuta kuin ohjelman toiminnan kannalta olisi pitänyt).

Purkaminen ja hajautus

Pakkauksen purku koostuu käytännössä kahdesta vaiheesta; kirjoitettavan tekstin muodostamisesta pakkauksen sisällön pohjalta sekä ko. tekstin kirjoittamisesta. Kirjoittaminen ei kuitenkaan ollut pitkä operaatio (kuten em.), joten keskityn tässä ainoastaan kirjoitettavan tekstin muodostamiseen ja selvittämään, mitkä operaatiot sen muodostettaessa vievät eniten aikaa.

Tekstin muodostus sen sijaan vei todella paljon aikaa omaa hajautustauluani käyttäen. Toisaalta kun kokeilin vastaavaa Javan HashMap -tietorakennetta käyttäen, pakkaaminen sujui huomattavasti nopeammin.

Kokonaisajat omalla hajautustaululla (arvot 11.6. klo 21:00) sekä Javan HashMap:illä.

Pakkaus	HajautusTaulu	HashMap
huffman.txt.hemi	250ms	140ms
lorem ipsum.txt.hemi	2000ms	660ms
kalevala.txt.hemi	17000ms	3000ms

(Seuraavia kokeiluja varten käytän omaa tämän hetkistä hajautustauluani.)

Tekstin muodostaminen on nelivaiheinen prosessi;

- 1) pakkausten sisällön luku
- 2) sisällön "Huffman -puu" -osan läpikäynti
- 3) pakkausten teksti -osan (String teksti = 010110...) muodostaminen vastaavasta tavukoodista
- 4) kirjoitettavan tekstin muodostus kohtien 2) ja 3) avulla.

Tekstin luku on nopea prosessi (kestää kalevalan kohdalla alle 300ms) ja puun läpikäynti ei vie läheskään edes sitä vertaa (< 20ms).

Kohta 3 vie huomattavasti enemmän aikaa (lorem ipsum vie noin 300ms ja kalevala hieman alle 2s). Se ei käytä hajautustaulua mihinkään ja sen käyttämä aika on lineaarisesti suoraan verrannollinen tekstin (011010...) pituuteen pakatussa tiedostossa tavuina.

Hidas operaatio, joka tämän aikana suoritetaan on jokaisen tavun muuntaminen 8-merkkiseksi String -olioksi BinaariMuuntaja -luokkaa hyödyntäen. Jos sen koodaisi toimimaan eri lailla ja tietysti nopeammin, tulisi tässä ajansäästöä.

Kohta 4 taas vei loppuajan, ja koska kolme edeltävää vaihetta ovat nopeita suhteessa tekstin muodostamisessa kuluvaan aikaan ja purkamisen kokonaisaika on huomattavasti pienempi Javan valmista hajautusrakennetta käyttäen, kulutettu aika liittyy oman hajautustauluni hitaampaan toimintaan.

Kohta (4), oma haj.taulu (11.6 klo 21:00)

Pakkaus	HajautusTaulu	HashMap
huffman.txt.hemi	200ms	59ms
lorem ipsum.txt.hemi	1500ms	140ms
kalevala.txt.hemi	15000ms	550ms

12.6 klo 15:30

En tehnyt vielä hajautustauluun muutoksia, mutta aloin testaamaan sitä paremmin. Löysin hajautustaulusta ongelman; hajautustaulu voi olla todella suuri ja sisältää ainoastaan vähän avaimia. Tällöin sekä poisto -että "sisältääAvaimen" operaatiot ovat äärimmäisen hitaita mikäli avain ei ole taulussa. Tällöinhän koko suuri hajautustaulu käydään läpi etsiessä avainta joka ei siellä ole.

Lisäys hajautustauluun on nopea operaatio niin kauan kuin hajautustaulu ei ole todella iso (> 10000 avainta). Tämän jälkeen - vaikka käytetään kaksoishajautusta - syntyy todennäköisesti paljon yhteentörmäyksiä, joka hidastaa paikan etsimistä, minne avain voidaan lisätä.

Kokeilenkin siis seuraavaksi muokata hajautustauluni operaatioita siten että uudelleenhajautus toteutuu aina kun hajautustaulussa on paljon tavaraa (koon kasvatus) tai kun vastaavasti siellä on vain vähän tavaraa (koon pienennys).

15.6 klo 17:20

Käytin viimeiset 2 päivää kirjoittamalla hajautustietorakenteeni kokonaan alusta, jotta saisin sen operaatiot "lisää", "poista" ja "etsi" toimimaan vakioajassa. Toteutus ei ole vielä ihan valmis, mutta se on nyt jo huomattavasti nopeampi kuin ennen ja toimii lähes ajassa $O(1)$ pienillä syötekoilla (< 10000).

Pakkaus	HajTaulu	HashMap
huffman.txt.hemi	120ms	59ms
lorem ipsum.txt.hemi	550ms	140ms
kalevala.txt.hemi	1800ms	550ms

Keko

Testasin omaa tietorakennetta "MinKeko" verraten sitä Javan valmiiseen "PriorityQueue":een. Toteutin testauksen luomalla while -silmukan sisällä uusia solmuja. Ensin 100kpl, sitten 1000kpl, 10000kpl ja lopuksi 100000kpl ja katsoin, miten lisäykseen kuluva aika näillä vaihtelee ja lisäyksen jälkeen poistin samat solmut yksitellen.

Solmujen lisääminen ja poisto / kuluva aika

Lisättyjä solmuja	MinKeko	PriorityQueue
10000	7ms	9ms
100000	20ms	25ms

Poistettuja solmuja	MinKeko	PriorityQueue
10000	24ms	17ms
100000	45ms	34ms

MinKeko tietorakenne näyttäisi toimivan moitteettomasti (myös siten että keko on alussa pieni ja sitä joutuu kasvattamaan, jotta uusia solmuja voidaan siihen lisätä) aikavaativuudessa $O(n \log n)$.

Mahdolliset puutteet ja parannusehdotukset

(täyd.)

Lähteet

(täyd.)