1 Totetusdokumentti

Järjestelmä muodostuu LZW-algoritmin [1] toteuttavista funktioista sekä tietorakenteista. Molemmat ovat toteutettu funktionaaliseen tyyliin sivuvaikutuksia ja mutaatioita välttäen. [2, s. 39–60] Pakkauslogiikka rakentuu **fs2**[1] kirjaston päälle, kutsuen `Stream`-tyypille erinäisiä funktioita jotka muuttavat datavirtaa. Pakkauslogiikka on jaettu kahteen tiedostoon: `Format.scala` pakkaa ja purkaa bittijonoja yksittäisten tavujen yli, ja tiedostossa `LZW.scala` olevat funktiot hoitavat itse sanakirjan ylläpitämisen ja koodien generoinnin.

Yhtenä projektin tavoitteena oli lisäksi saavuttaa yhteensopivus Unixin `compress`[2] -työkalun kanssa: `CompressHeader`-luokassa olevat funktiot lisäävät ja poistavat tiedoston alussa olevan tavujonon, jolla `compress` tunnistaa rakenteeltaan yhteensopivan tiedoston.

1.1 Tietorakenteet

LZW-algoritmin ydin tarvitsee toimiakseen jonkinlaisen hajautustaulutoteutuksen ja käytännössä hajautustaulu sekä ohjelman monet osat käyttävät listaa aputietorakenteena. Molemmat on toteutettu käyttäen *trie*-nimistä tietorakennetta, joka on puurakenne, jonka lehdet vastaavat tietorakenteeseen tallennettuja arvoja ja sisäsolmut vastaavat avainolion jotain osaa (esimerkiksi merkkijonossa tiettyä kirjainta tai yleisemmin bittijonossa tiettyä osabittijonoa). [3]

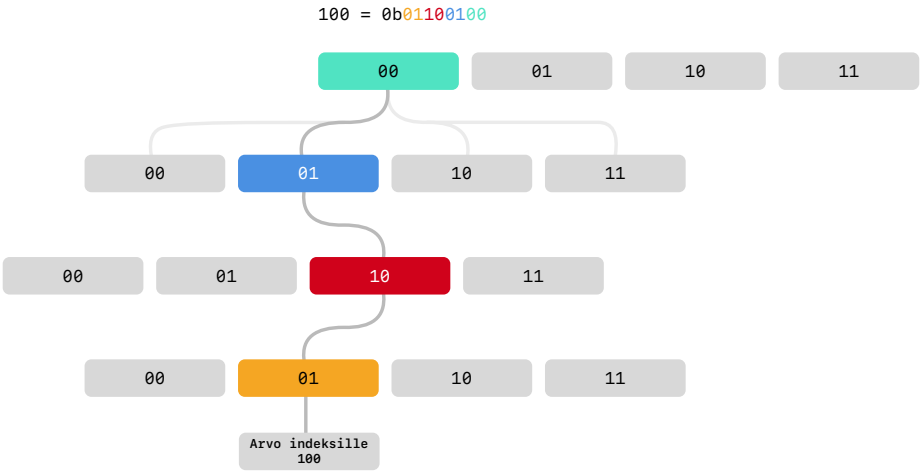
Tässä projektissa käytetty trie haarautuu puun joka tasolla 32 lapsisolmuun ja indeksoi tallennetut arvot pilkkomalla 32-bittisen kokonaisluvun bitit osabittijonoiksi, joiden avulla reititys arvon sisältävään solmuun tapahtuu. Tällöin havaitaankin, että

- rakenteeseen voi tallettaa enintään 2^{32} arvoa ja
- arvosolmun löytämisen aikavaativuus on $O(\log_{32} n)$. 32-bittisen kokonaisluvun suurin arvo on $2^{32} - 1$, joten reititys tekee korkeintaan $\log_{32}(2^{31} - 1) \approx 6.4 = 7$ hyppyä.

Triellä siis toisin sanoen saavutetaan tilanne, jossa $O(\log n)$ aikavaativuudellinen tietorakenne kasvaa aikavaativuudeltaan niin hitaasti, että useimmilla käytännön syötteillä rakenteen nopeus on lähestulkoon vakioaikainen. Trien toimintaa on havainnollistettu kuvassa 1.

Koska projektin tietorakenteet ovat toteutettu funktionaalisesti, ne eivät salli mutaatioita itseensä. Tällöin arvon lisääminen listaan tai hajautustauluun vaatii uuden tietorakenteen luomista ilman että vanhaan rakenteeseen tulee muutoksia. n-alkioisen taulukon käyttö vaatisi siis joka lisäyksen tai poiston yhteydessä kaikkien taulukon alkioiden kopioimista. Trie sen sijaan mahdollistaa puun polkujen kierrättämisen tietorakenteiden välillä siten, että vain muuttunut polku juurisolmusta johonkin tiettyyn arvosolmuun täytyy rakentaa uudelleen muutoksia tehdessä. Kun jokaisella puun tasolla on 32 lasta, on helppo nähdä, että

[1] `https://github.com/functional-streams-for-scala/fs2`
[2] `https://man.openbsd.org/compress.1`



Kuva 1: Indeksin 100 arvon hakeminen bitti-indeksoidusta triestä, jonka haarautumisaste on 4. Koska 4 alkiolla voidaan esittää vain binääriluvut $\{00, 01, 10, 11\}$, luvun 100 binääriesitys pilkotaan 2 bitin pituisiin osajonoihin.

jokainen päivitys vaatii maksimissaan 6 32-bittisen taulukon rakentamista uudelleen. Tämä mahdollistaa tehokkaan muistinkäytön verrattuna koko rakenteen kopiointiin.

Projekti sisältää kaksi trietä käyttävää tietorakennetta: `HashMapVector`-luokan, joka pohjimmiltaan on vain kevyt lisäys `SparseVector`in päälle kääntäen olion 32-bittiseksi kokonaisluvuksi jotain hajautusfunktiota käyttäen ja lisäten ylivuotolistat mahdollisia hajautusarvotörmäyksiä varten.

Toinen trietä käyttävä tietorakenne on `ListVector`, joka tarjoaa listamaisen rajapinnan trien käyttöön. Se siis varmistaa, että rakenteen avaimet eivät ole toisistaan erillään (rakenne on isomorfinen i-alkioisen linkitetyn listan kanssa), mikä mahdollistaa listan ensimmäisen ja viimeisen alkion hakemisen $O(1)$ ajassa. Toisaalta verrattuna normaaliin linkitettyyn listaan `ListVector` tarjoaa $O(\log n)$-nopeuksisen pääsyn mihin tahansa alkioonsa.

Pakkausalgoritmi käyttää apuna myös `BitBuffer`-nimistä aputietorakennetta, joka on käytännössä listarajapinta 64-bittisen kokonaisluvun päälle. Sen avulla on mahdollista yhdistää useita bittijonoja yhdeksi bittijonoksi, mikä on kätevää pakattaessa bittijonoja tavuiksi.

1.2 Funktionaalisesta ohjelmoinnista

Yksi projektin päätavoitteesta oli harjaannuttaa funktionaalisessa ohjelmoinnissa. Sen vuoksi projektissa käytetään huomattavasti **cats** [3]-kirjaston tarjoamia tyyppiluokkia (*type class*), mitkä mahdollistavat esimerkiksi algebrallisten rakenteiden mallinnuksen sekä toiminnallisuuden lisäämisen luokkiin loogisina

[3] `https://github.com/typelevel/cats`

kokonaisuuksina. Näitä tyyppiluokkia voidaan ajatella rajapintoina, joita projektin eri komponentit toteuttavat. Siten ne myös mahdollistavat selkeän erottelun järjestelmän eri osa-alueiden välille.

1.3 Saavutetut aikavaativuudet

Sekä hajautustaulurakenne että listarakenne saavuttavat ajassa $O(\log n)$ toimivan haun, lisäyksen sekä poiston trien avulla (kts. luku 1.1). `BitBuffer`-rakenne käyttää bittioperaatiota toimintaansa, joten sen aikavaativuus em. operaatioille on O(1). Järjestelmä pakkaa ja purkaa jonkin tiedoston yhdellä läpikäynnillä (*single pass*), joten sen aikavaativuus on tällöin $O(n \log n)$.

Käytännössä ohjelma on hyvin hidas verrattuna mihin tahansa moderniin pakkausalgoritmiin, joka on toteutettu imperatiivisesti. Empiirisen suorituskykytestauksen tulokset löytyvät testausdokumentista.

Viitteet

[1] T. A. Welch. “A Technique for High-Performance Data Compression”. *Computer* 17.6 (kesäkuu 1984), s. 8–19. ISSN: 0018-9162. DOI: 10.1109/MC.1984.1659158.

[2] Paul Chiusano ja Rnar Bjarnason. *Functional Programming in Scala*. 1st. Greenwich, CT, USA: Manning Publications Co., 2014. ISBN: 1617290653, 9781617290657.

[3] Donald E. Knuth. “The Art of Computer Programming, Volume III: Sorting and Searching”. Teoksessa: 1973.