Toteutusdokumentti - Sanaindeksi

Ohjelman perusrakenne

Ohjelman pääluokkana toimii Sanaindeksi, jossa ohjelman tekstikäyttöliittymä toimii. Se antaa käyttäjän antamat tekstitiedostot yksikerrallaan Lukijalle, joka lukee ne rivi kerrallaan ja antaa kunkin sanan tiedosto- ja rivitietoineen erikoismerkeistä ja numeroista siistittynä Puulle. Puu lisää sanat Solmuihinsa merkki kerrallaan ja sanan lopuksi lisää solmuun rivitiedon. Jokaisen tiedoston luvun lopuksi Lukija palauttaa Sanaindeksille Taulukon, jonka kussakin indeksissä on tekstitiedoston rivi myöhempää tulostusta varten. Taulukot sanaindeksi tallentaa erillisen taulukkoon, jonka kussakin indeksissä on kunkin tekstitiedoston rivit ja jonka koko määräytyy, kun alussa kysytään tiedosten määrä. Vastaavasti jokaisessa Solmussa on saman kokoinen taulukko, jonka jokaisessa indeksissä on vastaavan tekstitiedoston riviesiintymiset ko. solmuun loppuvalle sanalle.

Kun kaikki tiedostot on luettu, voi käyttäjän hakea sanoja, jolloin Puu suorittaa etsinnän hakusanalla tai sanoille. Sanojen alkujen haku toimii myös, koska jokaisessa solmussa on sen arvon esiintymiset. Trie-puussa solmun arvohan koostuu solmun, sen vanhemman ja kaikkien esivanhempien arvosta eli Solmu eroaa vanhemmastaan vain yhden merkin. Usean hakusanan tapauksessa Puu etsii jokaisen sanan erikseen ja vertaa näiden rivitietoja: kaksi ensimmäistä ensin, näiden esiintymisten leikkausta kolmanteen ja niin edelleen.

Aika-analyysi

Taulukko-luokka on dynaaminen taulukko, joka kaksinkertaistaa kokonsa täyttyessään. Dynaamisesta taulukosta haku indeksio perusteella on taulukon tapaan O(1) ja etsintä O(n), missä n on taulukon alkioiden lukumäärä. Dynaamisen taulukkon alkion lisäys lähestyy O(1):tä, koska se on useimmilla kerroilla O(1) ja vain taulukon täyttyessä O(n), jolloin kaikki arvot pitää kopioida uuteen taulukkoon. Taulukon kasvaessa kasvatustarve tulee yhä harvemmin vastaan. Dynaamisen taulukon tilavaativuus on noin kaksikertaa tavallisen taulukon O(n). Dynaamista taulukkoa käytetään ohjelmassa Solmun lasten, tekstitiedostojen sekä solmujen rivitietojen säilytykseen.

Solmu

Solmun luominen ja sen arvon hsku ovat O(1). Lapsen lisääminen solmulle vie O(n), missä n on solmujen lukumäärä, koska pitää tarkistaa onko jo olemassa lapsisolmu ko. arvolla.

Trie-puu

Solmun lisäys on kuvattu seuraavassa metodissa. Itse ohjalmassa jokaiseen solmuun lisätään myös rivitietoja, mutta ne tapahtuu lisäämällä uusi alkio taulukossa olevaan dynaamiseen taulukkoon, jonka indeksit tulevat Lukijalta, ja jonka aika vaativuus on O(1). Solmun lapsisolmujen läpikäynti vie ajan O(n), missä n on lapsisolmujen määrä ja tämä pitää toistaa sanan kaikkien merkkien kohdalla. Aikavaativuus on siis luokkaa O(n*z), missä n on lapsien lukumäärä ja z sanan pituus. Jos arvoidaan, että lapsisolmuja on enintään 29 erillaista (suomalaisten aakkosten määrä), niin tällöin aikavaatimukseksi voisi arvioida O(29*z) = O(z).

```
lisays(String sana){
Solmu nyt = juuri


for(int i = 0:sana.length() i++ ){
Solmu seuraava = nyt.lapsisolmu(sana(i))
if(seur == null){
Solmu x = new Solmu(sana(i))
nyt.uusiLapsi(x)
nyt = x }
else nyt = seur }


}
```

Yhden sanan etsinnässä on ideana edetä puuta pitkin juuresta alaspäin sanan kirjain kerrallaan. Jos seuraavaa kirjainta ei löydykään, palautetaan null. Muuten palautetaan viimeisen solmun rivitiedot. Aikavaativuus on taas vastaava kuin sanan lisäämisessä: O(n*z) tai vaihtoehtoisesti O(z) kuten edellä, missä n on solmujen lasten määrä ja z etsityn sanan pituus.

```
etsiSana(String sana){
Solmu nyt = this.juuri


for(int i = 0:sana.length(); i++){


Solmu seur = nyt.lapsisolmu(sana(i))
if(seur == null) return null
else nyt = seur}
```

return rivitiedot
}

Useamman sanan etsinnässä etsitään jokainen sana erikseen ja sitten vertaillee näiden rivitiedostoja. Uloin for-looppi käy hakusanoja läpi, kun sisempi käy kunkin sanan tekstitiedostokohtaisia riviesiintymistaulukoita. Sisin while käy läpi yhtä aikaa rivitiedot1:n ja rivitiedot2:n k:nsa rivitaulukoita ja etsii samoja arvoja. While-loopissa siis käydään läpi järjestettyjä rivinumerotaulukoita. Tämä voi kestää O(r+r) = O(r), missä r on kyseessä olevan tiedoston rivien lukumäärä. Sisempi for-looppi käy läpi kaikkia luettuja tiedostoja, joiden lukumäärään merkitään nyt t:llä. Uuden rivitiedoston alustus vie O(t) ajan ja uuden sanan haku O(n*z). Uloimman for-loopin sisällön aikavaativuus on siis luokkaa O(rt)+O(nz). Uloin for-looppi käy läpi hakusanoja, joita on h kappaletta. Lopullinen aikavaativuus olisi siis O(h(rt+nz)), jonka voisi olettaa pyöristyvät O(hrt), kun tekstitiedostot ovat riittävän suuria ja oletamme, että solmun lapsia n on enintään joitakin kymmeniä ja sanan pituus z on ”luonnollinen”.

etsiSanat(String[] hakusana){

rivitedot1 = etsiSana(hakusana[0])
if(rivitiedot1 == null) return null

for(int i= 1:hakusanat.length){

rivitiedot2 = etsisana(hakusana[i])
if(rivitiedot2 == null) return null

rivitiedotuusi.alustus()

for(int k = 0; k<rivitiedot.koko){
taulukoiden rivitiedot1[k] ja rivitiedot2[k] vertailu ja yhteisten arvojen lisäys rivitiedotuusi[k]:n
}
rivitiedot1 = uusi;
}
return rivitiedot uusi; }

Aikavaativuuden hahmottamista voisi helpottaa seuraava.

for(sanoja){
for(tiedostoja){
for(rivejä+rivejä){
} } }

Lukija

Tiedostoja lisätessä käydään tekstiä läpi rivikerrallaan ja kunkin rivin jokainen sana lisätäöän puuhun erikseen, eli aikavaativuus on luokkaaa O(sanojen määrä) jokaista tiedostoa kohden.

Kokonaisuus

Yhden sanan sanahaun voidaan sanovan olevan O(n*z), missä z on sanan pituus ja n solmun lasten maksimi. Useamman sanan hakeminen on noin O(hakusanat *rivit * tiedostot), eli huomasttavasti huonompi.

Testeistä

Puun rakenuksesta ja sanojen hausta kerättiin JUnit-testeissä dataa. Puun rakennukseen laskettiin mukaan myös tiedostojen luku ja tekstitiedostojen talletus. Tulokset korreloivat jonkin verran koon kanssa, mutta tarkempia arvoita on mahdoton tehdä, koska tekstitiedoston kokoa mitattiin riveillä. Sanahauissa aikavaativuuteen tuntui vaikuttavan merkittävästi sanan yleisyys hakutekstissä. Samoin useamman sanan haussa: jos sana ei ollut jollekin lähdeteksteistä tyypillinen, oli sillä myös vähemmän rivitietoja, joita verrata.

Parannusehdotuksia

Ohjelmaa voisi parantaa lisäämällä mahdollisuuden lisätä tiedostoja kesken ajon eikä vain aluksi. Tämän voisi tehdä esim. asettamalla tekstitiedostojen säilytys ja solmujen rivitiedot dynaamisiksi taulukoiksi, mutta tämä saattaa aiheuttaa ongelmia juuri Solmujen rivitietojen kohdalla. Pitäisikö uutta tiedostoa lisätessä käydä päivittämässä kaikki solmut? Vai pitäisikö sen sijaan solmu päivittää vain, kun sen tietoihin tulee lisäys, jolloin pitäisi varautua lisäämään tieto tyhjistä tiedoista. Jälkimmäinen olisi tehokkaampaa, mutta mahdollisesti myös riskialttiimpaa. Lisäksi käyttöliittymää voisi kehittää vähemmän herkäksi käyttäjän virheille tiedostoja syöttäessä.

Solmun lapsisolmut voisivat olla järjestetyssä taulukossa. Tällöin solmun haun voisi suorittaa binäärihaulla ajassa O(log n) nykyisen O(n) sijaan. Täs-

sä ohjelmassa Solmun haun yläraja on O(29), jos pysyttään suomenkielisisä teksteissä.

Lähteet

Trie: http://www.cs.helsinki.fi/u/ejunttil/opetus/tiraharjoitus/trie.html
Trien tulostus, Esa Junttila, 2005: http://www.cs.helsinki.fi/u/ejunttil/opetus/tiraharjoitus/tr
Trie: http://en.wikipedia.org/wiki/Trie
Tioterakenteet, Patrik Floreen, 2012: http://www.cs.helsinki.fi/u/floreen/tira2012/tira.pdf