Tietorakenteet 2012 -kurssin harjoitustyö: Taulukko vs. tasapainotettu hakupuu hakusanojen indeksoinnissa

Ohjaaja: Heikki Hyyrö
heikki.hyyro@uta.fi

1 Harjoitustyön aihe

Harjoitustyössä toteutetaan Java-ohjelmointikielellä hakusanaindeksi, jonka avulla voi tehokkaasti selvittää, missä kohdissa indeksoitua dokumenttia käyttäjän antama hakusana sijaitsee. Tällaisen indeksin toteuttamiseen on monia eri tapoja. Tässä harjoitustyössä indeksi toteutetaan, menetelmien vertailun vuoksi, seuraavilla kahdella tavalla:

1. Järjestettynä taulukkona (josta etsitään annettu hakusana binäärihakua käyttäen).
2. Vaihtoehtoisesti joko (2,4)-puuna tai sen yleisenä varianttina, B-puuna. Valinta vaikuttaa pisteytykseen (B-puun toteutuksesta saa enemmän pisteitä).

1.1 Harjoitustyön teosta ja palautuksesta

- Harjoitustyö tehdään yksin (joskin "konsultaatio-tyyppinen" yhteistyö on sallittua).
- Harjoitustyö palautetaan Weto-järjestelmään, joka sijaitsee osoitteessa https://ohop.cs.uta.fi/weto/.
- **Palautuksen aikaraja on 15.1.2013.**
  - Palautuksia voidaan vastaanottaa myöhässäkin, mutta myöhästyminen johtaa pistesakkoon.
  - Jos huomaat, ettet ehdi palauttaa työtä ajoissa, lähetä sähköpostia ohjaajalle jo ennen aikarajan 15.1.2013 umpeutumista ja sovi myöhäisestä palautuksesta.

Harjoitustyön teko kannattaa pyrkiä aloittamaan mahdollisimman aikaisessa vaiheessa, jotta palautuksen aikaraja ei pääse yllättämään. Vaikka asiasta vuosittain muistutetaan, tuppaa monen opiskelijan harjoitustyö viivästymään. Jos harjoitustyön teossa on esim. ohjelmointiteknisiä ongelmia, voi ohjaajalta tarvittaessa pyytää apua.

1.2 Harjoitustyön taustalla oleva sovellus: tekstuaalisen datan indeksointi

Monien tekstuaalisesta datasta koostuvien tietolähteiden, kuten tietokantojen, tietosanakirjojen tai tietoverkkojen, käytön yhteydessä tehdään hakusanaan pohjautuvia hakuja, joiden tuloksena saadaan tietoa siitä, missä kohdissa kyseinen hakusana (tai laajemmin ottaen sanat) esiintyy. Esimerkiksi tyypilliset internet-hakukoneet, kuten Google, tai kirjastojen tietokannat suorittavat pohjimmiltaan tämänkaltaisia hakuja. Hakusanapohjainen haku on siis hyvin perustavanlaatuinen tiedonhaun apuväline.

Tässä harjoitustyössä toteutetaan yksinkertainen hakusanaindeksi, jonka avulla voidaan paikantaa yksittäisen hakusanan sijaintikohdat yksittäisen tekstimuotoisen dokumentin sisällä. Tilanne on siten hieman yksinkertaistettu: toisaalta haun yhteydessä annettavien hakusanojen määrä on rajoitettu yhteen ja toisaalta haut kohdistuvat ainoastaan yhteen dokumenttiin. Harjoitustyössä muodostettava indeksi vastaa hieman esimerkiksi kirjojen indeksejä, joissa tyypillisesti on ilmoitettu yksittäisten hakusanojen sijaintikohdat (sivunumerot) kyseisen kirjan sisällä. Harjoitustyön tilanne kuitenkin eroaa kirjojen indekseistä siinä, että harjoitustyössä indeksoidaan sellainen tekstimuotoinen dokumentti, jota ei ole jaettu erillisiin sivuihin. Näin ollen harjoitustyössä indeksi ei ilmoita hakusanan sijaintikohdiksi esiintymäsivuja vaan niiden tarkat merkkipohjaiset sijaintikohdat: kuinka monen merkin päässä dokumentin alusta esiintymä sijaitsee. Tässä mielessä kohtelemme indeksoitavaa dokumenttia yksittäisenä yhtenäisenä merkkijonona, josta haluamme löytää ne indeksit, joiden kohdista alkaen hakusanan esiintymät löytyvät.

Harjoitustyössä voidaan yksinkertaisuuden vuoksi olettaa, että indeksoitavassa dokumentissaa esiintyy ainoastaan pienistä kirjaimista 'a'...'z' tai numeroista $1 \ldots 9$ koostuvia sanoja sekä niitä erottavia välilyöntejä. Hakusanan esiintymiksi hyväksytään kaikki sellaiset kohdat, joissa hakusana täsmää dokumentin sanan alkuosan kanssa (eli sanan ensimmäisestä merkistä alkaen). Esimerkiksi hakusanan "asia" esiintymäksi lasketaan sen kanssa täysin samanlaisen sanan "asia" lisäksi myös sanan "asiakas" alkuosa. Sen sijaan sanan "aasia" loppuosaa tai sanan "rasiat" keskellä olevaa esiintymää ei huomioida.

Tässä tehtävänannossa esitetyissä tehokkuutta koskevissa luonnehdinnoissa oletetaan, että indeksoitavassa dokumentissa on N merkkiä, n sanaa ja $\hat{n}$ keskenään erilaista sanaa.

1.3 Esimerkki

Oletetaan esimerkkinä, että indeksoitavan tekstimuotoisen dokumentin sisältö on "`ala jossa alati salaillaan on salainen ala`". Alla on esitetty tämä dokumentti siten, että kunkin merkin indeksi näkyy sen yläpuolella (lue indeksit ylhäältä alas).

```
         11111111112222222222333333333344
012345678901234567890123456789012345678901
ala jossa alati salaillaan on salainen ala
```

Edelliseen dokumenttiin hakusanalla “ala” tehtävän haun tuloksena tulisi saada vastauksena sijaintikohdat 0, 10 ja 39 eli niiden sanojen ensimmäisten merkkien indeksit, joiden alkuosana “ala” esiintyy.

2 Indeksin ensimmäinen toteutustapa: järjestetty taulukko

Sanojen esiintymien hakeminen esimerkiksi selaamalla jokaisen haun yhteydessä koko dokumentti alusta loppuun vaatisi perustoteutuksella $\mathcal{O}(N)$ sanavertailua. Yksinkertainen tapa tämän tehostamiseen on lajitella sanat aakkosjärjestykseen, poistaa joukosta turhat duplikaatit (saman sanan moninkertaiset esiintymät) ja merkitä kunkin sanan kohdalle, missä kohdissa dokumenttia se sijaitsee.

Alla on esitetty esimerkkinä, miltä tällainen järjestetty taulukko voisi näyttää edellisen esimerkkidokumentin kanssa.

Sana	**Esiintymäkohdat**
`ala`	0, 39
`alati`	10
`jossa`	4
`on`	27
`salaillaan`	16
`salainen`	30

Käytännön Java-toteutuksessa tällainen taulukko voisi olla esim. muotoa `Esiintymat[] indeksi`, jossa indeksin i omaava `Esiintymat`-olio `indeksi[`i`]` kuvaa rivin i tiedot (jos ajatellaan, että rivien numerointi alkaa nollasta). Luokka `Esiintymat` voisi siten omata jäseninään yhden `String`-muuttujan (sisältää sanan) ja yhden `int`-taulukon (sisältää sanan esiintymäkohdat).

Koska taulukko on järjestetty sanojen aakkosjärjestyksen mukaan, sijaitsevat saman hakusanan alkuosanaan omaavat sanat peräkkäin taulukossa. Aluksi voidaan hakea yksi näistä peräkkäisistä sanoista soveltamalla taulukkoon binäärihakua. Kun binäärihaulla on löydetty yksi tällainen sana, voidaan kaikki muut alkuosaltaan hakusanan kanssa täsmäävät sanat etsiä selaamalla taulukkoa binäärihaun löytämästä sanasta taakse- ja eteenpäin niin kauan, kuin alkuosat vielä täsmäävät. Hakumenetelmä on esitetty alla korkean tason pseudokoodilla.

1. alusta tyhjä tulosjoukko
2. hae binäärihaulla indeksi i, jolla olion $indeksi[i]$ sanan alkuosa täsmää hakusanan kanssa
3. **if** tällainen indeksi i löytyi **then**
4. $\quad j \leftarrow i - 1$
5. $\quad$ **while** $j >= 0$ **and** olion $indeksi[j]$ sanan alkuosa täsmää hakusanan kanssa **do**

6. lisää olioon $indeksi[j]$ talletetut esiintymäkohdat tulosjoukkoon
7. $j \leftarrow j - 1$
8. $j \leftarrow i$
9. **while** $j < \hat{n}$ **and** olion $indeksi[j]$ sanan alkuosa täsmää hakusanan kanssa **do**
10. lisää olioon $indeksi[j]$ talletetut esiintymäkohdat tulosjoukkoon
11. $j \leftarrow j + 1$
12. lajittele tulosjoukko kasvavaan järjestykseen ja palauta se

Yllä kuvatulla menetelmällä sanavertailujen määräksi tulee $\mathcal{O}(\log(\hat{n})+\hat{m})$, missä $\hat{m}$ on sellaisten keskenään erilaisten sanojen lukumäärä, joiden alkuosa täsmää etsityn hakusanan kanssa.

3 Indeksin toinen toteutustapa: tasapainotettu hakupuu

Järjestettyyn taulukkoon pohjautuva menetelmä on sinänsä kohtalaisen tehokas. Se ei kuitenkaan sovellu hyvin sellaiseen tilanteeseen, jossa indeksoitavaan sisältöön kohdistuu usein muutoksia. Esimerkiksi uuden sanan lisäys järjestetyn taulukon keskelle vaatii keskimäärin ainakin $\mathcal{O}(N)$ verran työtä, kun taulukon alkioita joudutaan siirtämään. Tilanne on oleellisesti parempi, jos talletamme keskenään erilaisten sanojen esiintymäkohdat ilmaisevat `Esiintymat`-oliot järjestetyn taulukon sijaan sopivaan tasapainotettuun hakupuuhun. Tällöin haun aikavaativuus ei muutu (binäärihaku ja tasapainotetusta hakupuusta haku ovat jokseenkin yhtä tehokkaita hakuoperaatioita), mutta sanajoukkoon kohdistuvat lisäykset tai poistot onnistuvat selvästi tehokkaammin kuin järjestetyssä taulukossa.

Kun alkiot on talletettuna hakupuuhun, muuttuu hakumenetelmää kuvaava pseudokoodi muotoon:

1. alusta tyhjä tulosjoukko
2. hae hakupuusta `Esiintymat`-olio v, jota vastaavan sanan alkuosa täsmää hakusanan kanssa
3. **if** tällainen olio v löytyi **then**
4. $u \leftarrow$ olion v edeltäjä
5. **while** $u \neq$ `null` **and** olion u sanan alkuosa täsmää hakusanan kanssa **do**
6. lisää olioon u talletetut esiintymäkohdat tulosjoukkoon
7. $u \leftarrow$ olion u edeltäjä
8. $u \leftarrow v$
9. **while** $u \neq$ `null` **and** olion u sanan alkuosa täsmää hakusanan kanssa **do**
10. lisää olioon u talletetut esiintymäkohdat tulosjoukkoon
11. $u \leftarrow$ olion u seuraaja
12. lajittele tulosjoukko kasvavaan järjestykseen ja palauta se

Harjoitustyössä toteutettavan hakupuun tulee olla joko (2,4)-puu tai B-puu. Seuraavissa alikappaleissa on annettu tarkentavia ohjeita siitä, miten nämä tulisi toteuttaa. Oleellista on, että puiden toteutusten tulisi olla yleisiä eli niitä pitäisi voida käyttää muissakin tilanteissa kuin tämän harjoitustyön yhteydessä.

3.1 (2,4)-puun toteutuksesta

Puu tulee toteuttaa geneerisenä[1] luokkana `TwoFourTree<E>`. Edellä luokan `TwoFourTree` nimen perässä annettiin kulmasulkujen sisällä ns. tyyppiparametri, jonka suhteen kyseinen luokkatoteutus on parametrisoitu. Tässä `E` on ikäänkuin paikanpitäjä jollekin konkreettiselle tyypille, ja luokkaa `TwoFourTree` varsinaisesti käytettäessä annetaan jokin konkreettinen tyyppi, jolla tämän tyyppiparametrin `E` esiintymät korvataan. Esimerkiksi `TwoFourTree<Esiintymat>` vastaisi (2,4)-puuta, johon talletetaan `Esiintymat`-olioita. Hieman yksinkertaistetusti tällöin voisi ikäänkuin ajatella, että luokka `TwoFourTree` käyttäytyy kuin sen toteutuksessa olevien tyyppiparametrin `E` esiintymien tilalla olisi alunperinkin ollut luokka `Esiintymat`[2].

3.1.1 Toteutettavat (2,4)-puun metodit

Puun toteutukseen tulee kuulua ainakin seuraavat julkiset metodit:

- `TwoFourTree(Comparator<E> cmp)`: Rakennin saa parametrinaan `Comparator<E>`-objektin `cmp`, joka määrittää puuhun talletettavien alkioiden järjestyksen. Kun puun toteutuksessa verrataan kahta siihen talletettua `E`-oliota `a` ja `b`, tehdään tämä kutsulla `cmp.compare(a, b)`. Tutustu Javan `Comparator<E>`-rajapinnan dokumentaatioon http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html.

- `void insert(E new_item)`: lisää alkion `new_item` hakupuuhun. Jos puussa on jo sellainen alkio `item`, jolle pätee `cmp.compare(item, new_item) == 0`, asetetaan alkio `new_item` aiemman alkion `item` tilalle.

- `void remove(E rem_item)`: poistaa puusta sellaisen alkion `item`, jolle pätee `cmp.compare(item, rem_item) == 0`. Puuta ei muokata, jos siinä ei ole tällaista alkiota `item`.

- `E smallest()`: palauttaa pienimmän puussa olevan alkion. Jos puu on tyhjä, palautetaan `null`.

  – Metodi tallentaa puun sisäiseen kirjanpitoon tiedon palautetusta alkiosta sekä sen sijainnista. Jälkimmäinen tieto koskee sekä solmua että kohtaa solmun

[1] Ellei geneerisyys ole ennestään tuttu aihe, tutustu Javan opassivuun "Lesson: Generics" http://docs.oracle.com/javase/tutorial/extra/generics/

[2] Tämä ei kuitenkaan aivan täysin pidä paikkaansa. Javan geneerisyydestä lisää kappaleessa 4.1.

sisällä (solmuissahan voi olla useita alkioita). Jos palautusarvo on `null`, riittää kirjata muistiin, että palautusarvo oli `null`.

  - Nämä kirjattavat tiedot ovat kaikille alkion palauttaville metodeille yhteisiä ja koskevat vain kaikkein viimeisimmäksi palautettua alkiota. Aina kun jokin alkio palautetaan, tulee mahdolliset aiemmat palautusarvoa koskevat muistiinpanot ylikirjoitetuksi.

- `E largest()`: palauttaa suurimman puussa olevan alkion. Jos puu on tyhjä, palautetaan `null`.

  - Metodi tallentaa puun sisäiseen kirjanpitoon tiedon palautetusta alkiosta sekä sen sijainnista.

- `E find(E query_item)`: Etsii puusta sellaisen alkion `item`, jolle pätee `cmp.compare(item, query_item) == 0`. Jos tällainen löytyy, palauttaa metodi viitteen alkioon `item`. Muuten palautetaan `null`.

  - Metodi tallentaa puun sisäiseen kirjanpitoon tiedon palautetusta alkiosta sekä sen sijainnista.

- `E next()`: Palauttaa viimeiseksi palautetun alkion seuraaja-alkion, missä viimeiseksi palautettu alkio identifioidaan puun sisäisen kirjanpidon avulla. Tässä seuraaja-alkio tarkoittaa alkiota, joka on puun järjestyksen perusteella pienin viimeksi palautettua alkiota suurempi alkio. Palautusarvo on `null`, jos tällaista alkiota ei ole olemassa (joko viimeksi palautettua alkiota ei ole olemassa tai viimeksi palautettu alkio oli puun suurin alkio).

  - Tämän metodin avulla voidaan selata puun alkioita suuruusjärjestyksessä, esimerkiksi seuraavantapaisesti:

```
E alkio = bpuu.smallest(); // Haetaan ensimmäinen eli pienin alkio
while(alkio != null)   // Selataan alkioita, kunnes tulee null
{
  // Tässä voisi tehdä alkiolle jotain?
  alkio = bpuu.next(); // Siirrytään seuraavaan alkioon
}
```

- `E prev()`: Palauttaa viimeiseksi palautetun alkion edeltäjä-alkion (sen alkion, joka on puun järjestyksen perusteella suurin kyseistä alkiota pienempi alkio). Tämä metodi toimii muuten vastaavaan tapaan kuin `next()`, mutta etenemissuunta on päinvastainen. Palautusarvo on `null`, jos tällaista alkiota ei ole olemassa.

- `E curr()`: Palauttaa viimeisimmäksi palautetun alkion nojautuen puun sisäiseen kirjanpitoon. Palautusarvo on `null`, jos tällaista alkiota ei ole olemassa.

- `E largerEq(E query_item)`: Palauttaa sen alkion, joka on puun järjestyksen perusteella pienin alkio, joka ei ole pienempi kuin `query_item`. Eli jos puussa on alkio `item`, jolla pätee `cmp.compare(item, query_item) == 0)`, palautetaan `item`. Muussa tapauksessa palautetaan puun pienin alkio, joka on suurempi kuin `query_item`. Palautusarvo on `null`, jos tällaista alkiota ei ole olemassa (eli jos puu on tyhjä tai sen kaikki alkiot ovat pienempiä).

  – Esimerkki: jos puussa on alkiot 0, 3, 4, 8, palauttaa kutsu `larqerEq(1)` alkion 3, kutsu `largerEq(-1)` alkion 0, kutsu `largerEq(4)` alkion 4 ja kutsu `largerEq(10)` `null`-viitteen.

- `E smallerEq(E query_item)`: Palauttaa sen alkion, joka on puun järjestyksen perusteella suurin alkio, joka ei ole suurempi kuin `query_item`. Eli jos puussa on alkio `item`, jolla pätee `cmp.compare(item, query_item) == 0)`, palautetaan `item`. Muussa tapauksessa palautetaan puun suurin alkio, joka on pienempi kuin `query_item`. Palautusarvo on `null`, jos tällaista alkiota ei ole olemassa (eli jos puu on tyhjä tai sen kaikki alkiot ovat suurempia).

  – Esimerkki: jos puussa on alkiot 0, 3, 4, 8, palauttaa kutsu `smallerEq(1)` alkion 0, kutsu `smallerEq(-1)` `null`-viitteen, kutsu `smallerEq(4)` alkion 4 ja kutsu `smallerEq(10)` alkion 8.

- `int n()`: palauttaa puun sisältämien alkioiden lukumäärän.

- `int size()`: palauttaa puun sisältämien solmujen lukumäärän.

- `void printTree()`: tulostaa puun solmut esijärjestyksessä. Yksittäisen solmun tulostaminen tarkoittaa tässä sitä, että tulostetaan solmun sisältämät alkiot välilyönnein eroteltuina yhdelle riville.

- `void printItems()`: tulostaa puun alkiot suuruus- eli jokseenkin[3] välijärjestyksessä. Alkiot tulostetaan yhdelle riville siten, että kutakin (mös ensimmäistä) arvoa edeltää yksi välilyönti.

3.1.2 Muita huomioita

Harjoitustyön toteutuksessa tulee noudattaa seuraavia sääntöjä, jotka ovat sopusoinnussa oheismateriaalin `bpuu.pdf` kanssa:

- Jos alkio poistetaan sisäsolmusta, tuodaan poistetun alkion tilalle sen edeltäjäalkio (seuraavaksi pienempi alkio puussa).

[3]Välijärjestys on määritelty lähinnä binääripuulle. (2,4)-puun kohdalla toimenpide on kuitenkin hieman vastaava.

- Jos alivuodon tilanteessa siirto tai sulautus olisi mahdollista tehdä sekä vasemmanettä oikeanpuoleisen sisarsolmun kanssa (esim. jos kummallakin sisarella on yhtä monta alkiota), suoritetaan kyseinen toimenpide oikeanpuoleisen sisarsolmun kanssa.
- Kun solmu jaetaan kahtia ylivuodon yhteydessä, tulee vasemmanpuoleiseen puoliskoon 2 alkiota ja oikeanpuoleiseen 1 alkio.

3.2 B-puu

Tietorakenteet 2012 -kurssilla on pääasiallisesti käyty läpi (2,4)-puu. B-puu on tällaisen puun yleistetty versio (tai oikeastaan: (2,4)-puu on B-puun erikoistapaus). B-puu on monitiehakupuu, jolle on määritelty **minimiaste** t. (2,4)-puu on yhtäkuin B-puu, jonka minimiaste on $t = 2$. Koska B-puu noudattaa samaa logiikkaa kuin (2,4)-puu, ei sen toteutus ole juurikaan työläämpää tai vaikeampaa kuin stavanomaisen (2,4)-puun toteutus. B-puun rakennetta on kuvattu ohessa olevassa dokumentissa `bpuu.pdf`.

B-puu tulee toteuttaa geneerisenä luokkana `BTree<E>`.

3.2.1 B-puun rakenne

Kullekin B-puun solmulle juurta ja lehtisolmuja lukuunottamatta pätee seuraavat säännöt:

- Solmun lasten lukumäärä = 1 + solmun alkioiden lukumäärä (eli puulla on tavanomainen monitiehakupuun rakenne).
- Solmulla on vähintään t ja enintään $2t$ lasta.
- Solmussa on vähintään $t - 1$ ja enintään $2t - 1$ alkiota.

Lehtisolmuja koskevat muuten samat säännöt, mutta niilä ei luonnollisestikaan ole lapsisolmuja. Juurisolmu taas on siitä poikkeava, että kun puussa on alle $2t$ alkiota, ovat nämä kaikki juurisolmussa eikä juurisolmulla ole yhtään lapsisolmua (puun ainoa solmu on juurisolmu).

3.2.2 B-puun ylläpito

Yleisen, minimiasteen t omaavan B-puun ylläpito tapahtuu täysin samalla logiikalla kuin erikoistapauksen $t = 2$ eli (2,4)-puun kohdalla. B-puun solmussa (juurta lukuunottamatta) on alivuoto, jos siinä on vähemmän kuin $t-1$ alkiota (vrt. (2,4)-puu, jossa $t = 2$ ja alivuoto tapahtuu, jos alkioita vähemmän kuin 2-1 = 1). B-puun solmun alivuoto korjataan

samaan tapaan kuin (2,4)-puussa eli tekemällä siirto tai sulautus. Vastaavasti B-puun solmussa tapahtuu ylivuoto, jos $2t-1$ alkiota sisältävään solmuun yritetään lisätä vielä yksi alkio (vrt. (2,4)-puu, jossa $t = 2$ ja $2t - 1 = 3$ alkiota sisältävään solmuun lisäys johtaa ylivuotoon). Myös B-puun solmun ylivuoto korjataan samaan tapaan kuin (2,4)-puussa eli jakamalla solmu kahtia.

B-puun ylläpitotoimenpiteet on kuvattu Tietorakenteet 2007 -kurssin kalvo-osuudessa `bpuu.pdf`, joka on `tira12.zip`-paketissa.

B-puun toteutuksen tulee olla sellainen, että minimiaste t annetaan parametrina rakentimelle puun luontihetkellä. Minimiaste on luonnollisesti puukohtainen (kaikilla saman puun solmuilla on sama minimiaste eikä sitä voi vaihtaa puun luonnin jälkeen), mutta saman kooditoteutuksen pohjalta toki voidaan luoda eri minimiasteen omaavia B-puita.

Samoin kuin (2,4)-puun tapauksessa, myös B-puun toteutuksessa tulee noudattaa seuraavia sääntöjä:

- Jos alkio poistetaan sisäsolmusta, tuodaan poistetun alkion tilalle sen edeltäjäalkio (seuraavaksi pienempi alkio puussa).
- Jos alivuodon tilanteessa siirto tai sulautus olisi mahdollista tehdä sekä vasemmanettä oikeanpuoleisen sisarsolmun kanssa (esim. jos kummallakin sisarella on yhtä monta alkiota), suoritetaan kyseinen toimenpide oikeanpuoleisen sisarsolmun kanssa.
- Kun solmu jaetaan kahtia ylivuodon yhteydessä, tulee vasemmanpuoleiseen puoliskoon t alkiota ja oikeanpuoleiseen $t - 1$ alkiota.

3.2.3 Toteutettavat B-puun metodit

B-puun tulee tarjota muuten täysin vastaavat metodit kuin (2,4)-puulle kuvattiin kappaleessa 3.1.1, mutta rakennin on hieman erilainen:

- `BTree(int t, Comparator<E> cmp)`: Rakennin saa parametrinaan B-puun minimiasteen t sekä `Comparator<E>`-objektin `cmp`, joka määrittää puuhun talletettavien alkioiden järjestyksen.

4 Toteutusteknisiä huomioita

Kappaleessa 3 annetussa sanahaun pseudokoodissa etsitään hakupuusta jokin sana, jonka alkuosana hakusana esiintyy. Tämä ei onnistu hakupuun tavallisella hakumetodilla `find`, koska se löytää ainoastaaan hakusanan kanssa kokonaan samanlaisen sanan. Sekä kokonaisen sanan että sanan alkuosan suhteen täsmäävä haku onnistuu hakupuun metodin `largerEq` avulla. Sen avulla voi etsiä aakkosjärjestyksessä pienimmän sanan, jonka alkuosana hakusana on. Tämä myös hieman yksinkertaistaa kappaleen 3 pseudokoodia, koska sen riveillä $4 \ldots 7$ tehty sanojen taaksepäin selaus voidaan jättää pois.

4.1 Javan geneerisyydestä

Javan geneerisyyden ideana on tarjota turvallisempi tapa toteuttaa luokkia (ja myös funktioita), joita voi hyödyntää monien erityyppisten olioiden yhteydessä. Esimerkiksi perinteisesti yleinen kaksi alkiota tallettava luokka `Pari` voisi olla muotoa

```
class Pari
{
  public Object a;
  public Object b;
}
```

Tässä on kuitenkin se potentiaalinen ongelma, että ohjelmoijan on halutessaan hankala kontrolloida sitä, minkätyyppisiä alkioita mihinkin `Pari`-olioihin talletetaan. Esimerkiksi

```
Pari x = new Pari();
x.a = new String("Yksi");
x.b = new Integer(2);
```

olisi laillista silloinkin, kun ohjelmoijan tarkoitus olisi tallettaa `Pari`-olioihin vain keskenään samantyyppisiä alkiopareja.

Javan geneerisyys antaa ohjelmoijalle mahdollisuuden kiinnittää alkioiden tyyppi käännösaikana johonkin tiettyyn tyyppiin. Esimerkiksi muotoa

```
class Pari<E>
{
  public E a;
  public E b;
}
```

olevaa geneeristä `Pari`-luokkaa käytettäessä koodin

```
Pari<Integer> x = new Pari<Integer>();
x.a = new String("Yksi");
x.b = new Integer(2);
```

toinen rivi tuottaisi kääntäjän virheilmoituksen, koska alkion tyyppi (tyyppiparametri `E`) oli käännöksen aikana kiinnitetty tyypiksi `Integer`. Javan geneerisyys toimii kuitenkin (hieman kömpelösti) niin, että tyyppiparametreja käytetään vain käännösaikaisiin tyyppitarkistuksiin ja lopullisessa käännetyssä koodissa tyyppiparametrit on korvattu yleisellä luokalla (oletusarvoisesti kaikkien olioiden yliluokalla `Object`). Esimerkiksi edellinen geneerinen `Pari`-luokka degeneroituisi käännöksen jälkeen tavalliseksi `Object`-pohjaiseksi `Pari`-luokaksi. Tällaisen degeneroituneen olion voi luoda myös suoraan itse, jättämällä

geneerisen luokan tyyppimääreet pois tapaan `Pari x = new Pari();`. Tällöin geneerisen `Pari`-luokan pohjalta tulee luotua alussa näytetyn tavallisen `Object`-pohjaisen `Pari`-luokan mukainen olio (jonka käytön suhteen kääntäjä ei tee tyyppitarkistuksia). Tällainen tyyppiparametriton oliotyyppi on samalla se tyyppi, minkä kääntäjä geneerisen luokan pohjalta lopulta oikeasti generoi (jahka on ensin tehnyt tyyppiparametrin mukaiset tyyppitarkistukset). Javan geneerisyys on siten ainoastaan käännösaikainen työkalu tyyppitarkistusten mahdollistamiseksi.

Javan geneerisyyden toteutustavan yksi tyypillinen konkreettinen ongelma on, että geneerisen luokan sisällä ei voi suoraan sellaisenaan luoda tyyppiparametrin mukaisia olioita tai siitä riippuvia taulukoita. Esim. lause `E x = new E();`, missä `E` on tyyppiparametri, tuottaisi kääntäjän virheilmoituksen, koska se vastaisi lopullisessa käännetyssä ohjelmassa lausetta `Object X = new Object();` riippumatta siitä, mitä konkreettista tyyppiä tyyppiparametrin `E` tilalla käytettiin. Kääntäjä ei luonnollisesti voi sallia tätä.

Harjoitustyössä ei tulle eteen tilannetta, jossa haluaisit luoda tyyppiparametrin mukaisen luokan objektin. Sen sijaan tyyppiparametrin mukaisia tai siitä riippuvia alkioita tallettavan taulukon luonti voi olla tarpeen (esimerkiksi solmuluokan sisällä). Annan esimerkin, miten tämä voisi esimerkiksi onnistua. Yksi tapa puun solmuluokan toteutukseen on toteuttaa se puuluokan sisäisenä luokkana. Tällöin esim. luokan `BTree` ja sen solmuluokan `Node` rakenne voisi olla vaikkapa muotoa:

```
public class BTree<E>
{
  private class Node
  {
    Node[] child;     // Viitteet lapsisolmuihin
    E[] item;         // Viitteet alkioihin
    // Mahdollisia luokan muita jäseniä...
    Node()   // Rakennin (voisi ehkä ottaa jonkin parametrinkin?)
    {
      child = new BTree.Node[2*t];    // Lapsia voi olla korkeintaan 2*t
      item = (E[]) new Object[2*t-1]; // Alkioita voi olla korkeintaan 2*t-1
      // Loput rakentimen tekemät toimenpiteet...
    }
  }
  // Luokan BTree toteutus jatkuu...
}
```

Edellisessä esimerkissä on lähinnä kaksi huomattavaa seikkaa.

Lapsisolmujen taulukko luodaan tyyppiä `BTree.Node` olevana taulukkona. Tyyppi `BTree.Node` on se lopullinen tyyppi, jota `Node` käännöksen jälkeen vastaa (luokan `BTree` tyyppiparametri on jätetty pois), ja tällaisesta "konkreettisesta" tyypistä saa luoda olioi-

ta/taulukoita. Kääntäjä tuottaa tästä kohdasta tyyppimuunnosta koskevan varoituksen, mutta sen voi jättää huomioimatta.

Alkioiden taulukko voidaan luoda yleisenä `Object`-taulukkona. Jotta kääntäjä sallisi lopputuloksen asetuksen tyyppiä `E[]` olevaan taulukkoviitteeseen, tarvitaan tyyppimuunnos. Kääntäjä antaa tyyppimuunnoksesta varoituksen, mutta sen voi jättää huomioimatta.

5 Hakusanaindeksin toteutus sekä testiaineisto

Harjoitustyön kummallekin hakusanaindeksityypille tulee luoda pääohjelma, joka vastaanottaa komentoriviltä kaksi parametria: dokumenttitiedoston nimen ja hakusanatiedoston nimen. Ohjelma lukee ja indeksoi dokumenttitiedoston sisällön ja kohdistaa sen jälkeen indeksiin, yksi kerrallaan, haun kullakin hakusanatiedoston sisältämällä hakusanalla. Kukin hakusanatiedoston sisältämä hakusana on omalla rivillään.

Edellämainitut pääohjelmat tulee lisäksi toteuttaa siten, että

- Järjestettyyn taulukkoon pohjautuvan indeksin (luku 2) pääohjelma on toteutettuna tiedostossa `TableDictionary.java`.
- Hakupuuhun pohjautuvan indeksin (luku 3) pääohjelma on toteutettuna tiedostossa `TreeDictionary.java`.
  - Koodi tukeutuu joko toteuttamaasi (2,4)-puuhun tai B-puuhun.
- Ohjelmat tulostavat ruudulle kunkin hakusanan `query` sijaintikohdat siten, että ensin tulostetaan otsakerivi
  `Positions where "query" occurs:`
  Tässä sanan `query` tilalla tulostetaan luonnollisesti kulloinkin kyseessä oleva hakusana. Seuraavalle riville tulostetaan yksitellen ja kasvavassa järjestyksessä kukin löytynyt sijaintikohta siten, että jokaisen sijaintikohdan eteen, ensimmäinen mukaanlukien, tulostetaan yksi välilyönti. Näiden jälkeen tulostetaan yksi ylimääräinen rivinvaihto.

5.1 Testiaineistot

Paketin `tira12.zip` alikansiossa `testing` on annettu valmiina seuraavat apuvälineet auttamaan oman koodisi toimivuuden testauksessa.

5.1.1 Hakusanaindeksien testiaineisto

- `smalldoc.txt`: Pieni dokumenttitiedosto.

- `smallquery.txt`: Pieni hakusanatiedosto.
- `americana.txt`: Suurehko dokumenttitiedosto.
  - Tarkoitettu lähinnä toteutuksesi lopputestaukseen. Eli testaa ensin, että toteutuksesi toimii oikein tästä dokumentista pienen katkelman sisältävän pienen dokumenttitiedoston `smalldoc.txt` kanssa.
- `SlowIndex.java`: Esimerkki "hakuindeksin" pääohjelman toteutuksesta. Tämä toteutus ei indeksoi dokumenttia vaan ainoastaan lukee sen yhteen `String`-merkkijonoon ja etsii hakusanojen esiintymät naiivilla (ja hitaalla) lineaarisella haulla, käyttäen `String`-luokan metodia `indexOf`.
  - Tätä ohjelmaa voi käyttää vertailukohtana testatessasi omia hakusanaindeksitoteutuksiasi. Suorita haku sekä tällä (esim. tapaan `java SlowIndex smalldoc.txt smallquery.txt`) että omalla toteutuksellasi ja vertaa tulosteita. Niiden pitäisi olla samanlaiset.

Edellämainittu esimerkkidokumentti `americana.txt` on otettu tekijänoikeuksista vapaasta *Encyclopedia Americana*-tietosanakirjasta[4].

5.1.2 Puutoteutuksen testiaineisto

Harjoitustyössä toteutettavan hakupuun toteutuksen tulee olla siinä mielessä yleinen, että se toimii myös muissa yhteyksissä kuin harjoitustyön hakusanaindeksin apuvälineenä. Testaa hakupuusi toimintaa seuraavalla testiaineistolla:

- `Test24.java`: (2,4)-puun testiohjelma. Käännä ja aja tämä, jos olet toteuttanut (2,4)-puun. Ohjelma vaatii toimiakseen rinnalleen luokan `TwoFourTree` toteutuksen. Ohjelma luo (2,4)-puun ja kokeilee sen tarjoamien eri metodien toimivuutta.
- `output24.txt`: Testiohjelmaa `Test24.java` vastaava esimerkkitulostus. Kun suoritat testiohjelman oman (2,4)-puun toteutuksesi kanssa, tulisi lopputuloksena saada samanlainen tuloste.
- `TestB.java`: B-puun testiohjelma. Käännä ja aja tämä, jos olet toteuttanut B-puun. Ohjelma vaatii toimiakseen rinnalleen luokan `BTree` toteutuksen. Ohjelma luo B-puun ja kokeilee sen tarjoamien eri metodien toimivuutta.
- `outputB.txt`: Testiohjelmaa `TestB.java` vastaava esimerkkitulostus. Kun suoritat testiohjelman oman B-puun toteutuksesi kanssa, tulisi lopputuloksena saada samanlainen tuloste.

[4] http://en.wikipedia.org/wiki/Encyclopedia_Americana

6 Harjoitustyön pisteytys

Hyväksytystä harjoitustyöstä voi saada 0-10 pistettä. Eri osista saatavissa olevat pistemäärät ovat:

- Järjestetyn taulukon / binäärihaun toteutus (pääohjelma `TableDictionary.java`): 0-2 pistettä.
- Puun toteutus: 0-6 pistettä, siten, että (2,4)-puun toteutuksesta saa 0-3 pistettä tai vaihtoehtoisesti B-puun toteutuksesta saa 0-6 pistettä.
  - Pisteytys koskee sekä pääohjelman `TreeDictionary.java` että puuluokan `TwoFourTree` tai `BTree` yleistä toimintaa (esim. testiohjelman `Test24.java` tai `TestB.java` kanssa).
- Toteutuksen kommentointi: 0-2 pistettä.
  - Koodiin tulisi sisällyttää riittävästi kommentteja. Kunkin luokan, metodin ja jäsenmuuttujan rooli/toimintaa tulisi lyhyesti kuvata. Sama koskee koodin hiemankin hankalampia kohtia.
  - 2 pistettä vaatii, että koodiin sisällytetään Javadoc-kommentit kaikille luokille ja metodeille.

Huomaa, että kussakin edellämainitussa kohdassa 0 pistettä on alin hyväksyttävästä toteutuksesta saatava pistemäärä. Jos jokin osa-alue on toteutettu liian heikosti (tai jopa jätetty kokonaan toteuttamatta), ei kyseistä osa-aluetta (ja samalla koko harjoitustyötä) hyväksytä.

Edellämainittujen kohtien yhteenlasketusta pistemäärästä tehdään lisäksi vähennyksiä, jos harjoitustyö palautetaan myöhässä.

Kunkin hyväksytyn osa-alueen pisteet määräytyvät toisaalta koodin toimivuuden mukaan (toimiiko koodi kaatumatta ja loogisesti oikein) ja toisaalta toteutuksen muun laadun mukaan.

Yleisenä hyväksymiskriteerinä voisi lisäksi mainita, että harjoitustyön tulee pohjautua omaan ohjelmointityöhön, vaikka toki neuvojen kysyminen muilta on sallittua. Työssä ei lisäksi saa käyttää mitään Javan valmiita varsinaisia tietorakenneluokkia tai apualgoritmeja. Esimerkiksi lajittelu ja binäärihaku tulee toteuttaa itse. Javan luokkakirjastosta saa käyttää lähinnä tiedostojen lukuun, merkkijonojen tallettamiseen, tulostamiseen tai poikkeusten käsittelyyn liittyviä perusluokkia. Rajapintoihin saa viitata vapaasti.

7 Työn palautuksesta

Harjoitustyö palautetaan Weto-järjestelmään viimeistään 15.1.2013.

Harjoitustyön mukana palautetaan tekstidokumentti, jossa:

- mainitaan tekijän nimi, opiskelijanumero ja sähköpostiosoite,
- kuvataan lyhyesti, mitkä osat on toteutettu, ja
- voi halutessaan antaa vapaamuotoista palautetta harjoitustyöstä.

Nimeä ylläkuvattu tekstidokumentti omalla opiskelijanumerollasi (esim. `11111.txt`, jos opiskelijanumero on 11111).

Laita dokumentaatio sekä toteutustiedostosi zip-pakettiin, joka on nimetty opiskelijanumerollasi, esim. `11111.zip`.