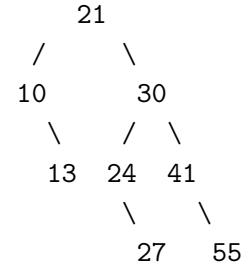
Tietorakenteet, laskuharjoitus 6, 23.-26.2.

Huom: ohjelmointitehtäviä ei saa suorittaa koneilla melkki, melkinkari, melkinpaasi

Huom2: Kolmannen TRAKLA-kierroksen deadline 28.2.

1. (a) Piirrä pienimmät ja suurimmat AVL-puut, joiden korkeus on 3 ja 4. Puiden solmujen arvot voit valita vapaasti kunhan ne vain täyttävät binäärihakupuuehdon.

   (b) Tarkasta, että alla oleva puu täyttää AVL-puuehdon:



2. (a) Näytä miten AVL-insert toimii kun tyhjään puuhun lisätään luetellussa järjestyksessä seuraavat avaimet 41, 38, 31, 12, 19, 8, 27 ja 49. Näytä jokaisen lisäyksen yhteydessä tehtävät kierto-operaatiot.

   (b) Sama kuin edellä, mutta avaimet lisätään alussa tyhjään puuhun päinvastaisessa järjestyksessä eli 49, 27, 8, 19, 12, 31, 38 ja 41.

3. (a) Näytä miten AVL-delete toimii kun se poistaa edellisen tehtävän (a)-kohdan lisäysten jälkeen tuloksena olevasta puusta avaimet 12, 49, 31 ja 8. Poistot tapahtuvat luetellussa järjestyksessä.

   (b) Ota lähtökohdaksi edellisen tehtävän (b)-kohdan tuloksena oleva puu ja poista siitä järjestyksessä avaimet 12, 8, 49 ja 41

4. Toteuta Javalla binäärihakupuu, jossa on operaatiot insert ja search. Puuhun talletettavien avainten tyyppinä on `long`. Tätä tehtävää varten puussa ei välttämättä tarvita parent-viiteitä ollenkaan.

   Testaamista varten on hyödyllistä toteuttaa algoritmi, joka tulostaa koko puun sisällön. Yksinkertaisimmillaan puun sisällön voi tulostaa esim. käymällä solmut läpi sisäjärjestyksessä, tällöin solmujen pitäisi tulostua suuruusjärjestyksessä.

   Voit myös tehdä puulle "kehittyneemmän" visualisoinnin laskareiden 5 tehtävän 5 tyyliin tai esim. tulostamalla puun solmut esijärjestyksessä siten, että tason i solmujen eteen tulostetaan esim. i*4 välilyöntiä, näin puu tulostuu "sivuttain peilikuvana".

   Bonus: Toteuta algoritmi, joka testaa puun korkeuden. Korkeuden testaus onnistuu samaan tyyliin kuin viikon 5 laskareiden tehtävässä 2 tehty puun solmumäärän summan laskeminen. Tämän tehtävän rastin saat myös ilman korkeuden testaavan algoritmin toteuttamista. Algoritmia tarvitaan joka tapauksessa seuraavassa tehtävässä.

5. Puun tehokkuustestaus

   (a) Tavallisen binäärihakupuun heikkous on, että puun korkeus voi kasvaa suureksi. Jos puuhun tulevat avaimet voidaan lisätä siihen yhdessä erässä, yksi ratkaisu ongelmaan on sekoittaa avaimet ensin satunnaiseen järjestykseen.

   Käytetään edellisen tehtävän binäärihakupuuta ja tutkitaan puiden korkeutta eri avainten määrällä, kun avaimet sekoitetaan ennen lisäystä. Käytännössä voit sekoittamisen sijasta lisätä puuhun satunnaisia avaimia. Javan valmiin Random-luokan avulla on helppo generoida satunnaislukuja.

   Mittaa useilla eri lisättyjen avainten määrillä miten lähellä puun korkeus on optimaalista korkeutta.

   (b) Tutustu Javan valmiiseen tasapainoiseen binäärihakupuutoteutukseen, eli luokkaan TreeSet.

   Vertaa empiirisesti TreeSet-puun ja oman puusi tehokkuutta. Tee kolme mittaussarjaa:

   - Lisää molempiin puihin n kpl satunnaisisia avaimia. Mittaa lisäämiseen kuluva aika.
   - Lisää molempiin puihin satunnaisia solmuja m kpl ja suorita n kpl satunnaisia search-operaatiota. Mittaa ainoastaan searcheihin kuluva aika.
   - Lisää molempiin puihin solmut $1, 2, 3, \ldots, m$ suuruusjärjestyksessä ja suorita n kpl satunnaisia search-operaatiota. Mittaa ainoastaan searcheihin kuluva aika.

   Kaikissa mittaussarjoissa kokeillaan useita m:n ja n:n arvokombinaatioita ja jälleen kiinnostuksen kohteena lähinnä isot syötteet. Algoritmin suoritusaikaa voit mitata samaan tapaan kuin viikon 4 laskareiden tehtävänannossa.

   Mitä opimme mittaustuloksista? Miten tulokset suhtautuvat O-analyysiin?

   Huom: jos tunnet erittäin suurta antipatiaa Javaa kohtaan voit tehdä tämän ja edellisen tehtävän myös jollain muulla kielellä. Jos toimit näin, joudut etsimään käyttämästäsi kielestä Javan TreeSet:iä (eli tasapainoitettua binäärihakupuuta) vastaavan kirjastototeutuksen. Ainakin C++:n STL:stä löytyy sopiva vertailukohta (set), mutta esim. Pythonissa ei taida olla valmista toteutusta.

6. (a) Binäärihakupuun operaatioiden *min* ja *max* (pienimmän ja suurimman alkion etsiminen) aikavaativuus on $\mathcal{O}(h)$, jossa h on puun korkeus. Miten puun pseudokooditoteutusta tulisi muuttaa, että operaatioiden *min* ja *max* aikavaativuus olisi vain $\mathcal{O}(1)$ ja muiden operaatioiden aikavaativuus säilyy ennallaan.

   (b) Binäärihakupuun operaatioiden *succ* ja *pred* (seuraavan ja edellisen alkion etsiminen) aikavaativuus on $\mathcal{O}(h)$, jossa h on puun korkeus. Miten puun pseudokooditoteutusta tulisi muuttaa, että operaatioiden *succ* ja *pred* aikavaativuus olisi vain $\mathcal{O}(1)$ ja muiden operaatioiden aikavaativuus säilyy ennallaan.

   (c) Mitä hyötyä ja haittaa edellisistä muutoksista on?