**Tietorakenteet, laskuharjoitus 3**, 31.1.-4.2.

Muista että viikon 2 pajatehtävien deadline on maanantai 31.1. klo 23.59.

**1.** **(a)** Seuraava algoritmi laskee summan $1 + 2 + 3 + \ldots + n$:

```
summa = 0
for i = 1 to n
    summa = summa + i
```

Määritä algoritmin aika- ja tilavaativuus.

**(b)** Summan 1+2+...+n voi laskea myös rekursiivisesti seuraavasti:

```
Summa(k)
    if k==0
        return 0
    else
        return summa( k-1 ) + k
```

Määritä algoritmin aikavaativuus ja tilavaativuus syötteellä $n$.

**(c)** Voiko summan laskea niin, että aikavaativuus on vain $\mathcal{O}(1)$?

**2.** "Linkitetyt rakenteet for dummies"

Harjoitellaan hieman linkitettyjen rakenteiden käyttöä Javalla. Ota pohjaksesi monisteen sivujen 110-112 pinototeutus. Koodi myös seuraavassa:

```
class PinoSolmu {
    int key;
    PinoSolmu next;

    PinoSolmu(int k, PinoSolmu seur) {
        key = k; next = seur;
    }
}

public class Pino {
    private PinoSolmu top;

    public Pino() {
        top = null;
    }

    public void push(int k) {
        PinoSolmu x = new PinoSolmu(k, top);
        top = x;
    }

    public int pop() {
        PinoSolmu x = top;
        top = x.next;
        return x.key;
    }
```

```
    public boolean empty() {
        return (top == null);
    }
}
```

On mahdotonta saada linkitettyihin rakenteisiin perustuvia ohjelmia toimimaan "sattumalta". Muista siis koko ajan piirrellä kuvia, jotka auttavat hahmottamaan mitä pinon olioiden välillä tapahtuu, eli miten oliot roikkuvat toistensa langan päissä. Myös debuggerin avulla voi tarkastella mikä pinon tila kulloinkin on.

**(a)** Tee pääohjelma jossa luot pinon ja lisäät pinoon neljä lukua.

**(b)** Tee pinolle metodi `int paalla()` joka palauttaa pinon päällimmäisen luvut. Metodi ei poista pinosta mitään.

Tässä kohdassa ja kaikissa seuraavissa muista tehdä pääohjelmaan koodi joka testaa, että metodisi toimii.

**(c)** Tee pinolle metodi `int toisiksiYlimpana()` joka palauttaa pinon toisiksi ylimpänä olevan luvun. Metodi ei poista pinosta mitään.

Miten käy jos pinossa on vähemmän kuin 2 alkiota?

**(d)** Tee pinolle metodi `int kolmanneksiYlimpana()` joka palauttaa pinon kolmanneksi ylimpänä olevan luvun. Metodi ei poista pinosta mitään.

**(e)** Tee pinolle metodi `void tulostaKolmeYlinta()` joka tulostaa pinon kolme ylinta lukua.

**(f)** Tee pinolle metodi `int poistaToisiksiYlin()` joka poistaa pinossa toisena olevan luvun. Luvun sisältämä PinoSolmu täytyy siis poistaa pinosta. Metodi myös palauttaa poistetun luvun.

Voit olettaa, että pinossa on ainakin 2 alkiota.

**(g)** Tee pinolle metodi `void lisaaToiseksi(int k)` joka lisää pinon ensimmäisenä ja toisena olevan alkion väliin uuden luvun k.

**Jos et osaa tehdä tehtävää, tule Tira-pajaan.** On erittäin tärkeä hankkia rutiini linkitettyjen rakenteiden käsittelyyn!

**3.** Jatketaan pinon muokkaamista. Ota pohjaksi yllä oleva pinon alkuperäinen koodi.

Tee jokaisen uuden metodin yhteyteen pääohjelmaan koodi joka testaa, että lisätty metodi toimii.

**(a)** Tee pinolle metodi `int koko()` laskee pinossa olevien alkioiden määrän ja palauttaa sen.

**(b)** Tee pinolle metodi `void tulostaKaikki()` joka tulostaa pinon sisällön, eli pinoon talleteutut luvut.

**(c)** Tee pinolle metodi `void lisaaPohjalle(int k)` joka lisää pinon pohjalle luvun k

**(d)** Mieti miten saat edellisen metodin toimimaan ajassa $\mathcal{O}(1)$ ja tee muutos koodiisi. Huom: olet oikeastaan tehnyt pinostasi jonon! Jonollahan on operaatiot lisää jonon perään ja ota kärjestä.

**Jos et osaa tehdä tehtävää, tule Tira-pajaan.** On erittäin tärkeä hankkia rutiini linkitettyjen rakenteiden käsittelyyn!

**4.** Toteuta Javalla tai jollain haluamallasi kielellä taulukkoon perustuva pino, jossa taulukon kokoa kasvatetaan jos push-operaation yhteydessä taulukossa ei ole enää tilaa uudelle alkiolle.

Toteuta pinosta kaksi versiota: toisessa taulukon koko aina kaksinkertaistetaan, toisessa taas taulukon koko kasvaa aina sadalla. Molemmissa taulukon koko on aluksi 100.

Vertaile empiirisesti toteutustesi suorituskykyä, eli mittaa kuinka hyvin jonot toimiviat eripituisilla suurilla push-operaation peräkkäisillä suorituksilla. Suoritukseen kuluvan ajan voi mitata esim. seuraavasti:

```
// kokeile useita suuria arvona, esim. väliltä 1000-1000000
int kertaa = 1000;

KaksinkertaistavaPino p1 = new KaksinkertaistavaPino();
long alku = System.currentTimeMillis();
for ( int i=0; i<kertaa; i++ ) p1.push(i);
long loppu = System.currentTimeMillis();
System.out.println("aikaa kului :"+(loppu-alku)+" millisekuntia");
```

Olemme kiinnostuneita lähinnä isoista syötteistä. Piirrä kuvaaja molempien versioiden käyttämästä ajasta suoritettujen push-operaatioiden suhteen.

Mitä johtopäätöksiä pinojen suorituskykyjen empiirisestä analyysistä voi tehdä? Mistä suorituskykyero johtuu?

**Huom: vaikka kyseessä on Tietorakenteet-kurssi, niin ohjelmointi kannattaa tehdä tyylillä.**

Eli jos haluat toimia järkevästi, määrittele rajapinta pino:

```
interface Pino {
   void push(int luku);
   int pop();
   boolean empty();
}
```

Laita molemmat pinototeutuksesi toteuttamaan rajapinta:

```
public class KaksinkertaistavaPino implements Pino {
   // ...
}
```

Tee metodi jolla hoidat ajan mittaamisen:

```
public static long mittaaAika(Pino pino, int operaatoita) {
    long alku = System.currentTimeMillis();
```

```
        for ( int i=0; i<operaatoita; i++ )
            pino.push(i);

        long loppu = System.currentTimeMillis();
        return loppu-alku;
    }
```

Kutsu pääohjelmastasi (tai jostain muualta) sopivasti tätä metodia (joka toimii siis kaikille Pino-rajapinnan toteuttajile) toistuvasti kaikilla syötteillä, ja tulosta mittaukset ruudulle tai tiedostoon tms.

Jos muotoilet syötteen oikein, on mittauksista helppo tulostaa kuva GNUPlotilla, ks. `http://blogs.cs.helsinki.fi/tira2010/2010/02/03/gnuplot-yhdessa-minuutissa/`

**5.** $\mathcal{O}$-filosofiaa

**(a)** Tehtävänä on toteuttaa algoritmi, jolle annetaan $n$ lukua sisältävä taulukko ja joka laskee taulukon lukujen summan. On helppoa keksiä algoritmi, jonka aikavaativuus on $\mathcal{O}(n)$: tavanomainen for-silmukka, joka käy luvut läpi ja laskee niiden summan muuttujaan. Perustele, miksi $\mathcal{O}(n)$ on myös paras mahdollinen aikavaativuus ongelman ratkaisevalle algoritmille.

**(b)** Tilavaativuus tarkoittaa, kuinka paljon muistia algoritmi käyttää syötteen lisäksi. Ei ole harvinaista, että algoritmin aikavaativuus on $\mathcal{O}(n)$, mutta tilavaativuus on vain $\mathcal{O}(1)$. Tällöin algoritmi tulee toimeen kiinteällä määrällä apumuuttujia. Onko sen sijaan mahdollista, että algoritmin aikavaativuus on $\mathcal{O}(1)$, mutta tilavaativuus on $\mathcal{O}(n)$?

**(c)** Tarkastellaan taulukkoa, jossa on kokonaislukuja:

| 5 | 2 | 1 | 3 | 1 |
|---|---|---|---|---|

Koko taulukon sisällön voi kutistaa yhteen kokonaislukuun kertomalla peräkkäisiä alkulukuja, joiden potenssit ovat taulukon luvut: $2^5 \cdot 3^2 \cdot 5^1 \cdot 7^3 \cdot 11^1 = 5433120$. Tästä luvusta saa selville alkuperäiset luvut etsimällä luvun alkutekijät ja lukemalla niiden potenssit.

Luvuille tuntuu siis olevan tuhlaavaista varata viisi kohtaa taulukossa, kun yksikin riittäisi:

| 5433120 | - | - | - | - |
|---|---|---|---|---|

Sama menetelmä tepsii mille tahansa taulukolle, jossa on $n$ kokonaislukua. Näyttää siltä, että alkuperäisen taulukon tilavaativuus on $\mathcal{O}(n)$, kun taas uuden taulukon tilavaativuus on vain $\mathcal{O}(1)$. Onko johtopäätös oikea?