Westerdals

Oslo School of Arts,
Communication and Technology

PG4200
Algoritmer og datastrukturer

Innlevering 1

Besvarelsen leveres i It's Learning innen
4. oktober 2015 klokken 23.55

Praktiske opplysninger

Besvarelsen leveres elektronisk i en `zip`-fil `innlevering1.zip` som skal inneholde nøyaktig tre filer:

- `IntegerStack.java`: Svar på oppgave 1.
- `analyse.pdf` og `Measurements.java`: Svar på oppgave 2.

Dere kan gjerne bruke filen innlevering1.zip som en modell for hvordan besvarelsen skal se ut.

Oppgave 1: Stakk-implementasjon

I denne oppgaven skal det lages to ulike implementasjoner av en stakk for `int`-verdier. Forskjellen mellom de to implementasjonene er at den ene implementasjonen lagrer elementene i et `ArrayList<Integer>`-objekt, mens den andre implementasjonen lagrer elementene i en `int[]`-tabell.

Den utleverte koden er organisert slik at all funksjonaliteten skal være tilgjengelig via den abstrakte klassen `IntegerStack`. Klassene `IntegerStackGeneric` og `IntegerStackPrimitive` er kun tilgjengelige via fabrikk-metoden `IntegerStack.create`. Denne løsningen er valgt *utelukkende* fordi den er praktisk i forbindelse med rettingen.

Med tanke på rettingen skal klassene ligge i `default`-pakken, d.v.s. at kode-filene ikke inneholder noen `package`-deklarasjon.

Den abstrakte klassen `IntegerStack`.

Vi kan tenke på denne klassen som et grensesnitt, selv om det strengt tatt er en abstrakt klasse. Denne klassen definerer kravene til de implementasjonene dere skal lage, gjennom 7 abstrakte metoder:

```
public abstract void push(int x);
public abstract int pop() throws NoSuchElementException;
public abstract int size();
public abstract boolean contains(int x);
public abstract void reverse();
public abstract Iterator<Integer> iterator();
```

Dokumentasjonen i filen `IntegerStack.java` gjør grundig rede for kravene som stilles til implementasjoner av disse metodene, d.v.s. hvilken kontrakt disse metodene skal tilfredsstille.

Klassen `IntegerStack` definerer også noen andre metoder, og ikke minst en statisk fabrikkmetode `create`. Andre klasser kan ikke instansiere `IntegerStackGeneric`- og `IntegerStackPrimitive`-objekter direkte, men må benytte `IntegerStack.create` for å få tak i slike objekter. Hvis man har importert klassen `IntegerStack`, kan man skaffe seg konkrete objekter på denne måten:

```
IntegerStack gStack = IntegerStack.create(IntegerStack.Type.GENERIC);
IntegerStack pStack = IntegerStack.create(IntegerStack.Type.PRIMITIVE);
```

OBS: Klassen `IntegerStack` skal **ikke endres**

Klassen `IntegerStackGeneric`.

Fullfør implementasjonen av denne klassen.

- Klassen skal implementere `IntegerStack`. Dette kravet avspeiles i deklarasjonen:

  ```
  class IntegerStackGeneric extends IntegerStack
  ```

- Klassen skal bruke et objekt av typen `ArrayList<Integer>` til å lagre objektene i stakken.

Hensikten med å bruke en `ArrayList` som underliggende datastruktur er at `ArrayList`-objektet selv tar ansvar for å utvide kapasiteten ved behov.

Klassen `IntegerStackPrimitive`.

Fullfør implementasjonen av denne klassen.

- Klassen skal implementere `IntegerStack`. Dette kravet avspeiles i deklarasjonen:

  ```
  class IntegerStackPrimitive extends IntegerStack
  ```

- Klassen skal bruke et en `int[]`-tabell til å lagre objektene i stakken.

Sjekkliste

- Skriv ferdig `IntegerStackGeneric`:
  - Implementere `IntegerStack`
  - Bruke en `ArrayList<Integer>` som underliggende datastruktur.
- Skriv ferdig `IntegerStackPrimitive`:
  - Implementere `IntegerStackPrimitive`
  - Bruke en `int[]`-tabell som underliggende datastruktur.
- Implementasjonene befinner seg i filen `IntegerStack.java`
- Alle klasser ligger i `default`-pakken.

Vurdering

Faglærer har skrevet en del tester som automatisk kontrollerer om de ulike metodene virkelig fungerer. Vurderingen av denne deloppgaven bygger på hvor mange tester implementasjonene klarer å tilfredsstille.

Faglærer kommer ikke til å offentliggjøre disse testene før etter at studentene har levert inn besvarelsene sine. Det å teste koden underveis mens man skriver den er en viktig del av programmeringsarbeidet, og det bør studentene gjøre selv.

Oppgave 2: Måling av kjøretid

I denne oppgaven skal dere sammenligne `IntegerStackGeneric` med `IntegerStackPrimitive`.

I `IntegerStackGeneric` vil det foregå en del *autoboxing* og *unboxing*. Dette kan bidra til at `IntegerStackGeneric` vil være tregere enn `IntegerStackPrimitive`, og spørsmålet i denne oppgaven er om vi kan merke denne effekten, og eventuelt hvor stor den er. Hvis du lurer på hva som menes med *autoboxing* og *unboxing* kan du lese denne artikkelen.

Hovedspørsmål

Vi ønsker å måle og sammenligne ytelsen til metodene `push` og `pop` i våre to `IntegerStack`-implementasjoner.

Vi kan spesifisere problemstillingen som følger: Definer

$T_G(n) =$ Kjøretid for n `push()` etterfulgt av n `pop()` i `IntegerStackGeneric`

og

$T_P(n) =$ Kjøretid for n `push()` etterfulgt av n `pop()` i `IntegerStackPrimitive`.

Oppgaven kan nå formuleres slik: - Finn mest mulig presise estimater for $T_G(n)$ og $T_P(n)$. - Sammenlig de to estimatene, f.eks ved regne ut gjennomsnittlig kjøretid pr. operasjon, det vil si $T_G(n)/n$ og $T_P(n)/n$. Kjøretiden kan også sammenlignes ved å regne ut forholdet mellom kjøretidene, altså $T_G(n)/T_P(n)$.

Her kan man bruke både teoretiske og empiriske metoder: Ved å studere programkoden kan man sette opp teoretiske estimater for $T_G(n)$ og $T_P(n)$. Ved å gjøre målinger av kjøretiden kan man så (i) vurdere de teoretiske estimatene og (ii) skaffe seg enda mer presise estimater for kjøretiden.

Sjekkliste

- Det er utført måling av kjøretid for ulike problemstørrelser.
- Målingene er fremstilt på en oversiktlig måte?
- Det er fremsatt teoretiske estimater.
- De teoretiske estimatene ersammenlignet med målingene.
- Man har reflektert kritisk over resultatene.

Vurdering

I denne oppgaven finnes det ingen fasit-svar og faglærer kan ikke på forhånd vite hva studentene bør komme fram til, blant annet fordi det dreier seg om ulike implementasjoner som kjøres med forskjellig utstyr.

Faglærer vil derfor ikke legge vekt på hvilke konklusjoner studentene har kommet frem til, men snarere at fremgangmåten og argumentasjonen er klar.

Tips til de som ønsker å granske mer

Her lister vi opp noen spørsmål som det kan være spennende å forfølge. Disse spørsmålene er *ikke* en del av oppgaven. Hvis noen likevel tar opp disse

spørsmålene, vurderes det som en del av refleksjonen over resultatene, og kan dermed ha en positiv innvirkning på vurderingen.

Just In Time-kompilering (JIT-kompilering)
Javas JIT-kompilering sies å ha stor betydning for kjøretiden til java-programmer. Det kan være interessant å se om det har noen betydning her. Man kan skru av JIT-kompileringen i programmet `Measurements.java` ved å starte programmet fra kommandolinjen med kommandoen

```
java -Xint Measurements
```

etter kompilering med kommandoen `javac Measurements.java`. Informasjon om hvordan man kjører programmer fra kommandolinjen finnes f.eks her.

Noen relevante spørsmål kan være:

(i) Hva er den prosentvise forskjellen i kjøretid med og uten JIT-kompilering?
(ii) Blir målingene mer regelmessige når vi slår av JIT-kompileringen?
(iii) Endres styrkeforholdene mellom de to implementasjonene når man slår av JIT-kompileringen?

Hvor mye koster det å bruke `ArrayList<Integer>`?
Det er veldig bekvemmelig å bruke en datastruktur som `ArrayList`, siden den selv tar ansvar for å utvide kapasiteten etter behov. Ulempen er at vi må bruke metodekall som `get`, `add` og `remove` for å håndtere objektene slik vi ønsker, og man kan spørre seg om dette påvirker kjøretiden nevneverdig. Det kan derfor være interessant å sammenligne en `IntegerStackGeneric` med en `IntegerStack`-implementasjon som benytter en `Integer[]`-tabell som underliggende datastruktur.

Et relevant spørsmål kan være: *Hva er den prosentvise forskjellen i kjøretid mellom to slike implementasjoner?*

Tips: Dersom man allerede har implementert `IntegerStackPrimitive`, er det meget enkelt å lage en implementasjon som baserer seg på en `Integer[]`-tabell istedenfor en `int[]`-tabell.

Forskjellen mellom *autoboxing* og *unboxing*:
Når en `int`-verdi *autoboxes* blir det opprettet et nytt `Integer`-objekt. Dette skjer i forbindelse med kall av `push` i `IntegerStackGeneric`.

Det som skjer når en `Integer`-verdi *unboxes* er at en `int`-verdi kopieres og samtidig som et `Integer`-objekt kan bli overflødig. Dette skjer f.eks i forbindelse med kall av `pop` i `IntegerStackGeneric`.

Ved å undersøke kjøretiden til push og pop hver for seg er det kanskje mulig å finne ut om autoboxing og unboxing kan ha ulik betydning for kjøretiden.

Totalvurdering av innleveringen

Karakterskala: A-F

Vekt: 20%

Grupper: 1-2 personer. I vurderingen blir det ikke tatt hensyn til gruppestørrelsen.

Karakterfastsettelse: Faglærer bedømmer oppgavene hver for seg. I totalvurderingen tillegges oppgave 1 og oppgave 2 like stor vekt, med mindre det forekommer opplagte følgefeil som det bør tas hensyn til.