Kræsjkurs INF101

Oppgavegjennomgang

2017

Oppg. 2 - H15 I

Et *sett* (eller *mengde*) er en uordnet samling av unike elementer, til forskjell fra f.eks. en list, hvor elementene har en rekkefølge, og eventuelt kan forekomme flere ganger. Det vil si:

- Lagringsrekkefølgen på elementene spiller ingen rolle, to sett er like om de inneholde de samme elementene uavhengig av hvordan de er laget.
- Å sette inn et element som allerede finnes i settet har ingen effekt.

I vedlegget finner du `ICollection`, et grensesitt for generelle samlinger av elementer, og en implementasjon av dette, `MyCollection`.

Oppg. 2 - H15 II

a) Vi skal nå lage grensesittet `ISet` for sett-samlinger, og klassen `MySet`, som implementerer en sett datastrukturer.

Aller først, gå igjennom `ICollection` og vurder om:

- metodene som er der også passer bra for sett
- om det er noen av metodene du mener ikke passer bra
- noe av dokumentasjonen burde være annerledes for `ISet`

Oppg. 2 - H15 III

b) La oss foreløpig anta at vi bruker `ICollection` mer eller mindre direkte som basis for `ISet`. Det vil si at vi har deklarert `ISet` slik:

```
public interface ISet<E> extends ICollection<E> { ... }
```

Implementer tre metoder:

```
public static void union(ISet<E> a, ISet<E> b) { ... }
public static void isect(ISet<E> a, ISet<E> b) { ... }
public static void diff(ISet<E> a, ISet<E> b) { ... }
```

Oppg. 2 - H15 IV

c) Vi kan også lage tester for `ISet` som er felles for alle klasser som implementerer `ISet`.

Lag fire metoder som uttrykker (ved `assertTrue`/`assertEquals`/etc) egenskaper ved metodene i `ISet`, inkludert

- sammenhengen mellom `contains` og en metode som endrer settet
- effekten av å legge til samme element flere ganger
- to andre egenskaper, f.eks. ved union/isect/diff fra forrige oppgave.

(Metodene vil typisk ta imot ett eller flere `ISet<E>`, og et eller flere elementer.)

Oppg. 5 - H16 I

Datastrukturen kø er en ordnet samling elementer som er slik at elementene hentes ut i rekkefølgen de settes inn i. Når vi henter et element fra køen blir det vanligvis også fjernet fra køen (slik at neste rykker frem).

For eksempel, hvis vi setter inn elementene "A", "B", "C" (i den rekkefølgen), vil "A" være det første elementet som blir hentet ut, og vi får ikke tak i "B" før vi har hentet "A".

Kø-datastrukturen er også kjent som FIFO-liste ? First-In, First-Out. Dette i motsetning til vanlige lister/tabeller, der elementer kan settes inn og hentes ut i vilkårlig rekkefølge (random access), og stack/stabel som er Last-In, First-Out (LIFO).

Oppg. 5 - H16 II

Lag et grensesnitt `interface IQueue` for køer. Du trenger ikke implementere noe (foreløpig). Velg navn etter hva du selv mener passer. Vi vil at `IQueue` skal ha følgende egenskaper:

- Vi må kunne velge elementtypen
- Vi må ha en metode for å fjerne og returnere neste element fra køen
- Vi må ha en metode for å legge til et element i køen
- Vi vil også kunne vite hvor mange elementer som er lagret i køen.
- Metodene må ha (kortfattet) dokumentasjon, så vi skjønner hvordan de skal brukes

Oppg. 6 - H16

Lag en klasse som implementerer grensesnittet `IQueue` fra forrige oppgave. Du velger selv hvordan du vil gjøre implementasjonen. Om du ønsker kan du gjøre bruk av klasser og metoder fra Java sitt klassebibliotek og/eller ting som vi har brukt i INF101.

Oppg. 7 - H16

En prioritetskø er en kø med en ekstra vri – elementene tilordnes en prioritet, og elementer med høyere prioritet 'sniker' seg frem i køen, slik at køen alltid er sortert slik at høyeste prioritet hentes ut først.

Prioriteten kan vi avgjøre ved f.eks. å sammenlikne elementene med compareTo. Vil det stille spesielle krav til elementtypen?

Lag et oppdatert grensesnitt for prioritetskø, `IPriorityQueue`. Metodene er de samme som før. Ta med eventuelle begrensinger på elementtypen.

Oppg. 8 - H16

Lag en klasse som implementerer `IPriorityQueue`.

Du kan regne med at om du har to elementer a og b, så har a høyere prioritet (skal foran b i køen) hvis `a.compareTo(b) < 0`.

Som før kan du bruke metoder og klasser fra Javas standardbibliotek og fra oppgaver/forelesninger/annet INF101-materiell. (Om du ikke helt husker bruken, beskriv kort hvordan du har tenkt.)

Oppg. 11 - V16

Lag en klassedeklarasjon for MyFraction som skal implementere `IFraction`.

Lag feltvariabler for teller og nevner (begge kan være av typen long - lange heltall), og lag en konstruktør som tar imot teller og nevner.

I likehet med f.eks. `Position`-klassen tenker vi at brøk-objektene ikke endres etter at de er opprettet.

Husk å sjekke parameterverdiene til konstruktøren før du bruker dem, om nødvendig.

Oppg. 12 - V16

En vanlig måte å lage equals-metoden på er på bare sammenlikne feltvariablene parvis - det er det Eclipse gjør hvis du ber den autogenerere equals.

Kan du se for deg et mulig problem med å implementere equals for MyFraction med en slik direkte sammenlikning av feltvariabler? Forklar kort, gjerne med eksempler.

Oppg. 13 - V16 I

Vi kjenner fra matematikken at hvis brøkens teller og nevner har en felles nevner (er delelig på samme tall), så kan den deles vekk og brøken forenkles. For eksempel, $4/8 = (2*2)/(2*2*2)$ som forenkles til $1/2$; $9/9$ forenkles til $1/1$, og så videre.

En annen forenkling er at å snu fortegnet over og under brøkstreken om dersom nevneren er negativ - da ender man opp med å alltid ha positiv nevner (nevneren kan som kjent ikke være 0). For eksempel kan man forenkle (-1)/(-2) til 1/2 og 1/(-2) til (-1)/2.

En enkel måte å finne felles nevnere på er Euclids algoritme, som finner den største felles nevneren til to heltall (dvs. det største heltallet som begge tallene kan deles på). Du kan anta at du har en implementasjon av denne i metoden long gcd(long a, long b) (gcd - greatest common divisor). framebreak

Oppg. 13 - V16 II

For eksempel:

```
long d = gcd(15, 75); // d = 5, dvs. 15/75 kan forenkles til (
d = gcd(3, 15); // d = 3, dvs. 3/15 kan forenkles til 1/5
d = gcd(1, 5); // d = 1, dvs. 1/5 kan ikke forenkles mer
```

Endre konstruktøren din for MyFraction slik at:

- Nevneren i brøken alltid er et positivt tall
- Telleren og nevneren ikke har noen felles nevnere (unntatt 1, som man alltid kan dele på)

Husk at det kan være flere fellesnevnere mellom teller og nevner, og at du ikke kan forenkle mer når gcd gir 1 som svar.

Oppg. 14 - V16 I

For et `MyFraction`-objekt er det nå mange kombinasjoner av verdier for feltvariablene som aldri skal forekomme (nevner=0 er et velkjent eksempel, forenkling av brøken i forrige deloppgave gjør enda flere kombinasjoner umulige).

Skriv ned en datainvariant (med ord) som beskriver hva som er gyldige tellere/nevnere for en `MyFraction` (gitt representasjonen med forenklet brøk).

Skriv også en metode `checkState` som sjekker at kombinasjonen av teller og nevner er gyldig / i samsvar med datainvarianten, og kaster en `IllegalStateException` hvis ikke.

Oppg. 15 - V16

Gitt at vi nå alltid har en forenklet brøk i MyFraction, blir problemene rundt equals-metoden noe annerledes enn tidligere? Forklar kort.

Oppg. 3 - V15 I

Det klassiske spillet *Space Invaders* ble utviklet av Tomohiro Nishikado i 1978. Spilleren styrer et lite romskip nederst på skjermen, som kan beveges til høyre og venstre, og skyte mot de invaderende romvesene, som kommer på rekke og rad nedover skjermen. Spillet er tapt når romvesene når helt ned på skjermen, eller spilleren blir truffet. Vi skal lage en noe forenklet utgave.

Oppg. 3 - V15 II

a) Vi skal modellere alle tingene i spillet ved hjelp av rektangler. Et rektangel har en posisjon (`IPosition`), en bredde og en høyde. Se kode for `IPosition`, `Position` og `IRectangle` i vedlegget. Lag en klasse `Rectangle implements IRectangle`, og definer feltvariabler og konstruktør som tar posisjon, bredde og høyde. Sjekk at bredden og høyden er ≥ 0, og kast en passende exception om det ikke er tilfelle.

Oppg. 3 - V15 III

b) Rektanglene våre skal være *immutable*, dvs. at de ikke kan endres etter at de er opprettet. Metodene `move()` og `moveTo()` må derfor returnere nye rektangler. Implementer disse to metodene.

Oppg. 3 - V15 IV

c) Implementer metodene contains(`IPosition`) og `contains(IRectangle)` i `Rectangle`.

Oppg. 3 - V15 V

d) Skriv minst to JUnit-tester for hver av de to `contains()`-metodene i `Rectangle`, som sjekker funksjonaliteten på noen enkle eksempler. Skriv også to tester for `overlaps()`.