Obligatorisk oppgave 3: Beholder

INF1010

Frist: mandag 13. mars 2017 kl. 12:00

Versjon 1.1 (`983bff0`). Endringer siden versjon 1.0

Innhold

1 Innledning

I denne oppgaven skal du lage en rekke beholdere som du vil få bruk for i de senere obligene. Det er derfor viktig å skrive god kode som er lett å lese, for du må komme tilbake til denne koden flere ganger.

Beholderne skal ha to forskjellige underliggende datastrukturer. Noen av beholderne lagrer elementene i et array, og de andre beholderne lagrer elementene i en lenkeliste.

Advarsel.

Det er ikke lov å bruke innebygde beholdere fra Java-biblioteket (med unntak av array) til å løse denne obligen. Du skal selv implementere beholderne fra bunnen av.

Vi trenger litt forskjellige varianter av lenkelister, så vi skal lage flere klasser som implementerer samme grensesnitt, men med litt forskjellig *semantikk* (virkemåte). Klassene dine kan selvfølgelig ha andre metoder i tillegg til dem som står i grensesnittet.

Når du løser denne oppgaven er det spesielt viktig at du tenker på hvilke *invarianter* som skal gjelde for beholderens tilstand[1]. Hvordan skal tilstanden være for en tom lenkeliste, og hvordan endrer dette seg når du setter inn et element? Blir tilstanden som tidligere når du tar ut det ene elementet i listen? Slike spørsmål bør du stille deg underveis.

1.1 Iteratorer og `Iterable`

Alle beholderne skal være iterérbare og implementere `Iterable<T>`. De trenger imidlertid ikke å gjøre dette eksplisitt ved `implements Iterable<T>`. Dette er fordi grensesnittene du får oppgitt for beholderne (`Tabell<T>` og `Liste<T>`) `extends Iterable<T>`, så deklarasjonen `class Stabel<T> implements Liste<T>` innebærer også at `Stabel<T>` implementerer `Iterable<T>`.

Grensesnittet Iterable ser slik ut:

```
interface Iterable<T> {
    Iterator<T> iterator();
}
```

`Iterator<T>` er et grensesnitt for **iteratorobjektet**, til forskjell fra `Iterable<T>` som er et grensesnitt for selve **beholderen**. Det er viktig at du skjønner forskjellen på disse to grensesnittene. `Iterator<T>` er gjengitt under:

```
interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove();
}
```

[1] Invariante tilstandspåstander er påstander som til enhver tid skal holdes sanne.

OBS.

Du trenger ikke å implementere `remove()`. Du kan bare la metoden stå tom.

`Iterable<T>` ligger i `java.lang` og trenger derfor ikke å importeres. `Iterator<T>` må importeres fra `java.util`.

Merk.

Alle klasser som implementerer `Iterable`, kan løpes gjennom med en for-each løkke.

Ofte vil en iterator kaste et `ConcurrentModificationException` hvis beholderen endres etter at iteratorobjektet er opprettet, men du trenger ikke å ta høyde for dette i denne obligen.

2 Tabell

Tabellene skal bruke et array som sin underliggende datastruktur. Vi sier at tabellens *kapasitet* er lengden på dette arrayet, mens tabellens *størrelse* er antall elementer som er lagret i tabellen.

Tabellens iterator skal stoppe etter den har returnert det siste elementet i tabellen – den skal altså stoppe før den kommer til de tomme plassene.

2.1 Grensesnitt

De array-baserte beholderne skal implementere følgende grensesnitt:

```
public interface Tabell<T> extends Iterable<T> {
    /**
     * Beregner antall elementer i tabellen
     * @return      antall elementer i tabellen
     */
    public int storrelse();

    /**
     * Sjekker om tabellen er tom
     * @return      om tabellen er tom
     */
    public boolean erTom();


    /**
     * Setter inn et element i tabellen
```

```
     * @param   element              elementet som settes inn
     * @throws  FullTabellUnntak     hvis tabellen allerede er full
     */
    public void settInn(T element);

    /**
     * Henter (uten aa fjerne) et element fra tabellen
     * @param  plass    plassen i tabellen som det hentes fra
     * @return          elementet paa plassen
     * @throws  UgyldigPlassUnntak  hvis plassen ikke er en gyldig
                                    indeks i arrayet eller plassen
                                    ikke inneholder noe element
     */
    public T hentFraPlass(int plass);
}
```

2.2 Unntak

Som det fremgår av javadoc-kommentarene til `Tabell<T>`, skal det kastes unntak hvis man prøver å sette inn et element i en full tabell, og det skal kastes unntak hvis man prøver å hente ut fra en plass som ikke er en gyldig indeks i arrayet eller som det ikke er satt inn noe på. Disse unntakene er definert under:

```
class UgyldigPlassUnntak extends RuntimeException {
    UgyldigPlassUnntak(int plass, int storrelse) {
        super(String.format("Plass: %d, storrelse: %d",
                            plass, storrelse));
    }
}

class FullTabellUnntak extends RuntimeException {
    FullTabellUnntak(int storrelse) {
        super(String.format("Storrelse: %d", storrelse));
    }
}
```

Merk.

Siden disse klassene arver `RuntimeException`, er det ikke nødvendig å håndtere disse unntakene for å få programmet til å kompilere. De vil likevel være til hjelp hvis programmet krasjer og vi lurer på hva som gikk galt, siden det vil skrives ut informasjon om størrelsen på tabellen og hvilken plass vi forsøkte å hente et element fra.

I programmene hvor du bruker denne tabellen, bør du når programmet er ferdig ikke trenge å ha noen håndtering av disse unntakene fordi programmet ditt selv gjør sjekker for å sørge for at de ikke oppstår.

2.3 Statisk tabell

Skriv en klasse `StatiskTabell<T>` som implementerer grensesnittet `Tabell<T>` som er gjengitt ovenfor. Konstruktøren skal ta inn lengden på arrayet (som også er tabellens kapasitet) som eneste parameter. Denne beholderen skal kaste et `FullTabellUnntak` hvis arrayet er fullt når et element blir forsøkt satt inn. Hvis man forsøker å hente fra en plass som ikke er en gyldig indeks i arrayet, eller som det ikke er satt inn noe på, skal det kastes et `UgyldigPlassUnntak`.

Merk.

Det er ikke mulig å opprette et array av en generisk type i Java. Du må derfor opprette et `Object[]` og *typekonvertere* (*caste*) dette til `T[]`

```
T[] tabell = (T[]) new Object[100];
```

Når du kompilerer programmet, vil du få en advarsel om at programmet "uses unchecked or unsafe operations", men det kommer av en begrensning i måten Java er implementert på, så det betyr ikke at du har gjort noe galt.

2.4 Dynamisk tabell (valgfri)

Det er valgfritt å lage denne klassen, men vi anbefaler at du gjør det hvis du har tid.

Skriv en klasse `DynamiskTabell<T>` som arver `StatiskTabell<T>` (og dermed også implementerer grensesnittet `Tabell<T>`). Denne beholderen skal ta høyde for at arrayet kan bli fullt og utvide kapasiteten om nødvendig. Dette kan gjøres ved å opprette et dobbelt så langt array og kopiere alle elementene over.

I tillegg kan det være hendig å ha en ekstra konstruktør uten parametre, som kaller den andre konstruktøren med en eller annen standardverdi for startkapasiteten, f.eks. 100.

Utover dette skal denne klassen virke på samme måte som `StatiskTabell<T>`.

3 Lenkelister

Vi skal implementere tre varianter av lenkelister:

1. En ordnet (sortert) lenkeliste.
2. En stabel (LIFO) hvor vi setter inn og tar ut elementer fra samme side.

3. En kø (FIFO) hvor vi setter inn elementer på én side og tar dem ut fra den andre siden.

Disse skal bygge på samme underliggende implementasjon, men nøyaktig hvordan du løser dette er opp til deg. Det viktige er at du klarer å gjenbruke koden du allerede har skrevet gjennom arv eller komposisjon. Hvis du leverer tre fulle implementasjoner av en lenkeliste med litt varierende metoder, vil du ikke få oppgaven godkjent.

3.1 Grensesnitt

De lenkeliste-baserte beholderne skal implementere følgende grensesnitt:

```
public interface Liste<T> extends Iterable<T> {
    /**
     * Beregner antall elementer i listen
     * @return      antall elementer i listen
     */
    public int storrelse();

    /**
     * Sjekker om listen er tom
     * @return      om listen er tom
     */
    public boolean erTom();

    /**
     * Setter inn et element i listen
     * @param   element     elementet som settes inn
     */
    public void settInn(T element);

    /**
     * Fjerner et element fra listen. Hvis listen er tom,
     * returneres null.
     * @return      elementet
     */
    public T fjern();
}
```

3.2 Stabel

Skriv en klasse `Stabel<T>` som implementerer grensesnittet `Liste<T>`. `settInn(T element)` skal sette inn `element` på starten av listen, og `fjern()` skal ta ut et element fra starten av listen.

3.3 Kø

Skriv en klasse `Koe<T>` som implementerer grensesnittet `Liste<T>`. `settInn(T element)` skal sette inn `element` på slutten av listen, og `fjern()` skal ta ut et element fra starten av listen.

3.4 Ordnet lenkeliste

Skriv en klasse `OrdnetLenkeliste<T extends Comparable<T> >` som implementerer grensesnittet `Liste<T>`. Listen skal være ordnet i stigende rekkefølge, slik at det minste elementet ligger først og det største elementet ligger sist, og `fjern()` skal ta ut det minste elementet.

`Comparable<T>` ligger i `java.lang` og ser slik ut:

```
interface Comparable<T> {
    int compareTo(T element);
}
```

`a.compareTo(b)` returnerer:

- Et negativt tall hvis `a` er mindre enn `b`
- 0 hvis `a` er lik `b`
- Et positivt tall hvis `a` er større enn `b`

Hva det vil si at objekt er mindre enn et annet, må vi selv velge hvordan vi skal definere. For `String`-objekter gjelder leksikalt sortering, så strengene blir sortert i alfabetisk rekkefølge. For andre objekter er det mindre åpenbart hvilket kriterium vi bør sortere etter.

4 Tester

Vi har skrevet en rekke tester du kan kjøre for å identifisere feil i koden din.

Advarsel.

Testene gir bare en indikasjon på om koden din er riktig – de finner ikke nødvendigvis alle feil.

Testene forutsetter at klassenavn er som beskrevet i obligteksten, at grensesnittene `Tabell<T>` og `Liste<T>` finnes, og at unntakene `UgyldigPlassUnntak` og `FullTabellUnntak` finnes. Disse grensesnittene og klassene er gitt tidligere i oppgaveteksten.

4.1 Nedlastning av prekode

Du kan laste ned en zip-fil eller en tarball med all prekoden og testene.

Du finner også koden på `https://github.uio.no/inf1010/oblig-prekode_v17/` sammen med instruksjoner for hvordan du pakker ut zip-filen og tarballen.

5 Oppsummering

Du skal levere alle beholderklassene i tillegg til de oppgitte grensesnittene (`Tabell<T>` og `Liste<T>`) og unntakene (`UgyldigPlassUnntak` og `FullTabellUnntak`), samt eventuelle andre klasser og grensesnitt som trengs for at programmet skal kompilere. Det er viktig at du **ikke** endrer på navnet på klassene og grensesnittene eller metodenes signatur, for da vil ikke testene virke.

Lykke til!
Stein Gjessing, Stein Michael og Kristian