LULEÅ TEKNISKA UNIVERSITET

Tentamen i
Objektorienterad programmering och design
Totala antalet uppgifter: 5
Lärare: Håkan Jonsson, 491000, 073-820 1700
Resultatet offentliggörs senast: Snarast..

Kurskod	D0010E
Datum	2018-05-23
Skrivtid	5 tim

Tillåtna hjälpmedel: Inga.

OBS! Lösningar som baseras på annat ur Javas standardbibliotek än `RuntimeException` (som fritt får kastas där så är lämpligt), `Object` och det som finns på den bifogade listan ger inga poäng. Eventuella undantag från detta anges i uppgiftstexten.

1 Teori

a) Vad är huvudpoängen med designmönstret *Iterator*? (0.5p)

b) Paketet `a` har klasserna `A1` och `A2`, paketet `b` klasserna `B1` och `B2` och paketet `c` klasserna `C1` och `C2`. `C1` ärver `B1` som ärver `A1`. `A2` ärver `B2` som ärver `C2`. `B1` innehåller en heltalsvariabel q deklarerad `protected`. Alla klasser är publika. Rita UML-diagrammet. (1.5p)

c) I vilken/vilka klasser i deluppgift b) syns q *inte*? (1p)

d) Vad returneras om vi gör anropet `ack(2,1)`? (1p)

```
public static int ack(int m, int n) {
  if (m == 0) {
    return n+1;
  } else if (n == 0) {
    return ack(m-1, 1);
  } else {
    return ack(m-1, ack(m, n-1));
  }
}
```

e) En heltalsarray kallas *palindrom* om ordningen mellan talen är densamma oavsett om man går igenom den från index 0 och uppåt eller åt andra hållet. Såväl $148, 37, 21, 37, 148$ som $42, 42$ och 330 är exempel på palindom.

Skriv en rekursiv metod `public static boolean isPalindrome(int[] a)` som avgör om en icke-tom array är ett palindrom. Lösningen får bestå av privata hjälpmetoder men varken bygga på iteration eller globala variabler. (2p)

2 Bokhyllor

a) Skriv en klass `Bok` för att representera en bok, som har en författare, en titel och ett förlag (alla är strängar) samt ett visst årtal då den publicerades (ett heltal). (2p)

b) Skriv med hjälp av `DEQ<E>` ur uppgift 4 en klass `Bokhylla` i vilken man kan ställa in (tillföra), söka efter och ta ut (avlägsna) böcker. Sökning och avlägsning görs genom att ange författare och titel. Internt ska böcker lagras i, och manipuleras med hjälp av, `DEQ<Bok>`-objekt. Arrayer får inte användas. (4p)

3 Bräckliga korgar

Till en diskret händelsestyrd simulering av ett snabbköp behövs korgar. En vara som kan läggas i korgarna är av den redan existerande typen `Vara`. Dess vikt ges av metoden `public double vikt()`.

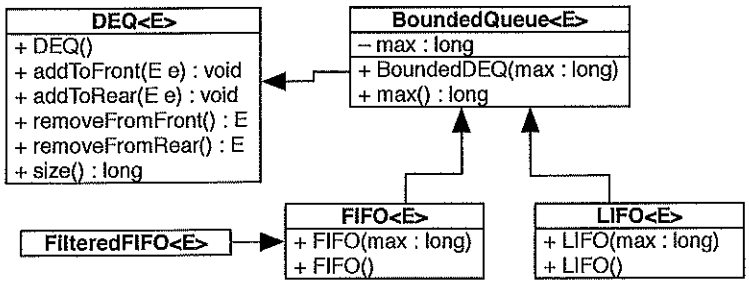
En korg har en konstruktor `public Korg(double maxVikt)`, där `mavVikt` är största vikten korgen håller för. En vara läggs till med metoden `public void stoppaIn(Vara vara)`. En korg kan dock som mest innehålla 50 varor vars sammanlagda vikt inte får överskrida gränsen. Om maximala antalet varor skulle överskridas stoppas inte varan in. Om en vara kan läggas till men viktgränsen då överskrids, går korgen sönder, töms på alla varor och kan sen inte tillföras nya varor.

Metoden `public double vikt()` returnerar *hur mycket mer* vikt som kan tillföras korgen utan att den går sönder medan `public int storlek()` ger antal varor i korgen. Metoden `public boolean ärSönder()` anger om den spruckit.

Uppgift: Skriv klassen `Korg` enligt ovan men lägg även till en möjlighet att skapa iteratorer till en korg genom att tillämpa designmönstret *Iterator*. Undantag ska kastas där så är lämpligt. `Vara` behöver inte skrivas. Iteratormetoden `remove` ska alltid kasta undantag. Som hjälp finns, på sidan 4, helt kort gränssnitten `Iterable<E>` och `Iterator<E>`. (6p)

4 Begränsade köer

Implementera klasserna för olika slags köer i figur 1 genom att använda arv i alla situationer det är lämpligt. OBS! UML-diagrammet innehåller inte allt som behöver implementeras. `DEQ<E>` är given; den behöver inte implementeras. (6p)



Figur 1: UML-diagram över bakverk.

- OBS! Överklassen `DEQ<E>` i UML-diagrammet behöver **inte** skrivas. Den beskrivs så här: Klassen `DEQ<E>` representerar en sekvens med element av typen `E` i vars båda ändar element kan läggas till och tas bort. Klassen har en konstruktor som skapar en tom kö samt följande metoder: `addToFront` och `addToRear` lägger till ett element i början respektive i slutet. Dessa metoder returnerar ingenting. `removeFromFront` och `removeFromRear` tar bort ett element från början respektive från slutet, och returnerar det borttagna elementet. Skulle kön vara tom kastar de istället undantaget `NoSuchElementException`. `size` returnerar slutligen antal element kön innehåller. (*Ja, detta är samma klass som skulle implementeras i tentan 2018-03-16.*)
- En `BoundedDEQ<E>` är en `DEQ<E>` med storleksbegränsning, dvs en `BoundedDEQ<E>` kan maximalt innehålla ett visst antal element. Detta antal (en `long`) ges som argument till konstruktorn. Metoden `max` returnerar storleksbegränsningen. Skulle storleksbegränsningen kommat att överskridas kastas undantag.

- En `FIFO<E>` (*First-In-First-Out*) är en slags `BoundedDEQ<E>` med två konstruktorer. Den med argument ger en kö med storleksbegränsning och den utan argument ger en kö som i praktiken saknar storleksbegränsning. Det senare uppnås internt genom att sätta storleksbegränsningen till `Long.MAX_VALUE`[1].

  I en `FIFO<E>` är `addToRear`, `removeFromFront` och `size` oförändrade. Metoderna `addToFront` och `removeFromRear` kastar däremot alltid endast undantag.

- En `LIFO<E>` (*Last-In-First-Out*) är en `BoundedDEQ<E>` med två konstruktorer som fungerar som konstruktorerna i `FIFO<E>`.

  I en `LIFO<E>` är `addToFront`, `removeFromFront` och `size` oförändrade. Metoderna `addToRear` och `removeFromRear` kastar däremot alltid endast undantag.

- En `FilteredFIFO<E>` är en `FIFO<E>`-kö som används för att tunna ut mängder. Dess två konstruktorer har en parameter p, $0 \leq p \leq 100$, som anger risken för att ett element som läggs till kastas bort av kön: Bortkastning sker om `Math.random()` $\leq p$.

5 Cirkulära köer

Implementera, med hjälp av en inre nodklass, på lämpligt sätt den publika och generiska klassen `CQ<E>` för *cirkulära köer* (*circular queues* på engelska) enligt nedan och utan att använda arrayer (och givetvis inte heller nåt ur Javas standardbibilotek). (6p)

Klassen representerar en mängd element av typen `E`. Om mängden har innehåll ordnas det så elementen följer på varandra runt i ring. Finns bara ett element så följer det efter sig självt.

Ett *finger* pekar på ett av elementen. För tomma mängden pekar dock inte fingret på nåt.

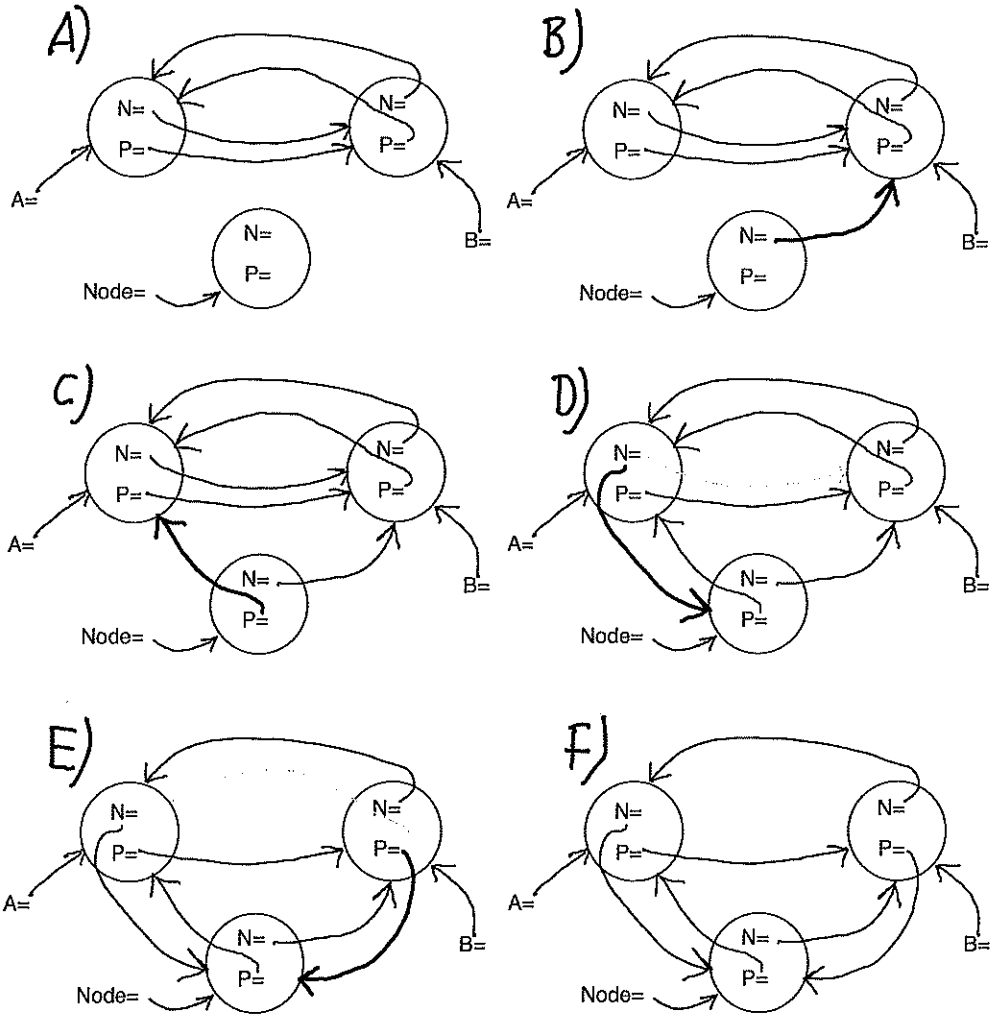
- Klassens konstruktor skapar en tom kö (med ett finger som inte pekar).
- Metoden `add(E item)` lägger till `item` i kön. Om kön var tom sätts fingret så det pekar på `item`. Annars sätts `item` in *efter* det element som fingret pekade på och utan att ändra fingret.
- Metoden `E remove()` tar bort elementet *efter* det som fingret pekar på och returnerar det. Skulle kön bli tom så pekar fingret sen inte. Om kön var tom vid anropet kastas istället ett undantag.
- Metoden `void forward()` flyttar, för köer med innehåll, fram fingret till nästa element. I en kö med bara ett element hamnar fingret på samma element som tidigare. Om kön var tom vid anropet kastas istället ett undantag.
- Metoden `size` returnerar slutligen antal element kön innehåller.

Viktigt: För poäng ska funktionen hos var och en av metoderna `add` och `forward` kortfattat *illustreras med skisser* som i steg, motsvarande satserna i metoderna, visar hur den interna nodkedjan och de interna nodernas referensvariabler förändras (precis som gjorts under kursen). Om en metod fungerar olika när kön är tom jämfört med då den inte är tom, så gör en skiss för respektive fall.

Figur 2 visar ett exempel på detta för en annan sorts nodkedjor (alltså inte säkert noder för `CQ`). Sekvensen A)-F) visar hur en nod, refererad av `Node`, länkas in mellan (dvs efter) `A`[2]

[1] `Long.MAX_VALUE`$= 2^{63} - 1$, eller nästan 10^{19}. Detta är mycket mer än vad konsumentdatorers primärminnen kan innhålla.

[2] OBS! Fel i figuren och texten - variabelnamnen ska förstås börja med små bokstäver, inte stora.



Figur 2: Exempel på hur ändringar av en intern nodkedja kan illustreras.

och (före) `B` i en dubbellänkad lista. Varje nod har här två referensvariabler `N` (som i "*n*ext node") och `P` (som i "*p*revious node"). Inlänkningen åstadkoms genom tilldelningssatserna B) `Node.N=B;`, C) `Node.P=A;`, D) `A.N=Node;` och E) `B.P=Node;`.

```
public interface Iterable<E> {
    Iterator<E> iterator(); // Returnerar en iterator över element av typen E.
}

public interface Iterator<E> {
    boolean hasNext() // Returnerar true om iteratorn har fler element.
    E next() // Returnerar nästa element i iteratorn.
    void remove() // Tar bort det element next() senast returnerade.
}
```

Figur 3: `Iterable<E>` och `Iterator<E>` till uppgift 3.

1) Följande ur Javas standardbiblioteket får fritt användas utan att förklaras, implementeras (eller ens importeras).

Metoder på strängar:

```
int length()                        antal tecken i strängen
char chatAt(int index)              "ABCDE".chatAt(2) == 'C'
String substring(int I, int j)      "ABCDE".substring(1,3) == "BC"
```

Utskrifter:

```
void System.out.print()             (Ny rad)
void System.out.print(…)            (Utskrift utan ny rad efteråt)
void System.out.println(…)          (Utskrift med ny rad efteråt)
```

Math:

```
double abs(double a)                absolutbeloppet (av a)
double sin(double a)                sinus av a (i radianer)
double cos(double a)                cosinus av a (i radianer)
double tan(double a)                tangens av a (i radianer)
double exp(double a)                exponentialfunktionen
double log(double a)                naturliga logaritmen
double pow(double a, double b)      upphöjt, a^b
double sqrt(double a)               kvadratroten ur a
double ceil(double a)               "tak" av a, avrunda uppåt
double floor(double a)              "golv" av a, avrunda neråt
double random()                     ger ett slumptal i, 0 <= i < 1
```

Översättning:

```
int Integer.parseInt(String s)
String Integer.toString(int i)

double Double.parseDouble(String s)
String Double.toString(double d)

double toDegrees(double a)          från radianer till grader
double toRadians(double a)          från grader till radianer
```

2) Några exempel på omtypningar:

```
"1234" + 99 == "123499"
Integer.toString(99) + "1234" == "99" + "1234" == "991234"

(int) 3.5 == 3
(int) -3.5 == 3

11 * 0.25 == 2.75
(int) 11 * 0.25 == 2.75
11 * (int) 0.25 == 0
(int) (11 * 0.25) == (int) (2.75) == 2
```