Använda prioritetsköer

Tema: Användning av Javas klass `PriorityQueue` och interfacen `Comparable` och `Comparator`.

Interfacen Comparable och Comparator

U 1. Javas klass `PriorityQueue` ska används för att hålla reda på patienter som väntar på en akutmottagning. Varje patient tilldelas en prioritet, som är ett positivt heltal. Ett lågt värde på detta attribut motsvarar hög prioritet. Patienter representeras av följande klass:

```
public class Patient  {
    private String firstname;
    private String lastname;
    private String personNbr;
    private int prio;

    public Patient(String firstname, String lastname, String personNbr, int prio) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.personNbr = personNbr;
        this.prio = prio;
    }

    public int getPrio() {
        return prio;
    }

    public boolean equals(Object other) {
        if (other instanceof Patient) {
            return personNbr.equals(((Patient) other).personNbr);
        }
        return false;
    }

    public int hashCode() {
        return personNbr.hashCode();
    }

    public String toString() {
        return firstname + " " + lastname + " " + personNbr + " " + prio;
    }
}
```

a) I en första variant av programmet skapar man prioritetskön så här:

```
PriorityQueue<Patient> pq = new PriorityQueue<Patient>();
```

Det innebär att klassen `Patient` antas implementera interfacet `Comparable` och att metoden `compareTo` används för att jämföra två element inuti `PriorityQueue`.
Se till att klassen implementerar gränssnittet `Comparable`. Det är patienternas prioritet som ska jämföras.

b) Efter att ha funderat vidare på problemet kommer man fram till att det är olämpligt att det finns två jämförelsemetoder i klassen `Patient` (`equals` och `compareTo`) som jämför olika attribut. Istället för att låta klassen `Patient` implementera `Comparable` vill man använda interfacet `Comparator`.
Skriv en klass som implementerar gränssnittet `Comparator` och ändra i koden för att skapa prioritetskön så att din nya komparatorklass används. *Eller* använd ett

lambdauttryck när du skapar prioritetskön.

c) På akutmottagningen vill man dessutom att patienter med lika prioritet ska behandlas i den ordning de kommit till mottagningen, dvs. den som väntat längst ska behandlas först. För att kunna garantera detta, oavsett vilken implementering av prioritetskö man använder, tänker man göra följande ändringar:

- Patientobjekt numreras. Det första objekt av klassen `Patient` som skapas får nummer 1, det andra nummer 2 etc. Tips! Använd ett statiskt attribut för att hålla reda på numret.
- Jämförelsen mellan patienter modifieras så att det vid lika värde på attributet prio är objektets nummer som avgör vilket objekt som är minst.

Gör de ändringar som krävs för detta i klassen `Patient` samt i komparatorklassen respektive ilambdauttrycket.

Tillämpning med prioritetskö

U 2. I den här uppgiften ska vi se närmare på några klasser som är tänkta att användas i ett program för att administrera aktiehandel. Programmet är under utveckling och till att börja med finns följande klasser:

```
public class Customer {
    private String id;

    /** Skapar en kund som kan handla med aktier. */
    public Customer(String id) {
        this.id = id;
    }
}

public class Order {
    private double price;
    private Customer customer;

    /** Skapar en köp- eller säljorder för en aktie med budpriset price och
        köpare/säljare customer. */
    public Order(double price, Customer customer) {
        this.price = price;
        this.customer = customer;
    }

    /** Returnerar köp/säljbudet för aktien. */
    public double getPrice() {
        return price;
    }
}
```

För enkelhets skull antar vi att en köp- respektive säljorder gäller en aktie.

a) För att hålla reda på köp- och säljordrar för ett visst aktieslag används en klass `OrderQueues`: Ett aktieslag representeras helt enkelt av en specifik kod, t.ex. ”ABCD”.

```
public class OrderQueues {
    private String shareId;
    private PriorityQueue<Order> buyOrders;     // sorterad efter avtagande pris
    private PriorityQueue<Order> sellOrders;    // sorterad efter växande pris

    /**
     * Skapar ett objekt som hanterar en kö för köpordrar och en kö för
```

```
     * säljordrar för aktien med id shareId.
     * @param shareId aktieslag
     */
    public OrderQueues(String shareId) {
        // Fyll i egen kod här.
    }

    /**
     * Lägger till en köporder ifall matchande säljorder inte finns.
     * Om matchande säljorder finns tas säljordern bort och returneras.
     * @param buyOrder köporder
     * @return matchande säljorder om sådan finns, i annat fall null
     */
    public Order addBuyOrder(Order buyOrder) {
        // Fyll i egen kod här.
    }

    /**
     * Lägger till en säljorder ifall matchande köporder inte finns.
     * Om matchande köporder finns tas köpordern bort och returneras.
     * @param sellOrder säljorder
     * @return matchande köporder om sådan finns, i annat fall null
     */
    public Order addSellOrder(Order sellOrder) {
        // Ingår ej i uppgiften att implementera.
        // Blir liknande kod som i addBuyOrder.
    }
}
```

Observera att:

- Köpordrarna ska vara sorterade efter avtagande pris.
- Säljordrarna ska vara sorterade efter växande pris.
- När någon vill lägga en köporder ska man först undersöka om den matchar en tidigare inlagd säljorder. Detta inträffar om köpbudet är större än eller lika med lägsta säljbud.

Implementera konstruktorn och metoden `addBuyOrder` i klasen `OrderqQueues`. Gör också de ändringar i befintliga klasser/tillägg av nya klasser som krävs för att köerna ska fungera som avsett.

b) Själva aktiehandeln sköts i klassen `ClearingHouse` där attributet `q` håller reda på orderköerna för alla aktieslag. Implementera metoden `buy`.

```
public class ClearingHouse {
    private Map<String, OrderQueues> q;

    /** Skapar ett objekt som hanterar aktiehandel. */
    public ClearingHouse() {
        q = new TreeMap<String, OrderQueues>();
    }

    /**
    * Låter kunden customer lägga en köporder av aktieslaget shareId till
    * budpriset price. Genomför köpet om matchande säljorder finns, i annat
    * fall lagras köpordern i motsvarande orderkö.
    * @param customer kunden
    * @param shareId aktieslag
    * @param price budpris
    * @throws NoSuchElementException om det inte finns någon orderkö för
    * aktieslaget shareId.
```

```
        */
        public void buy(Customer customer, String shareId, double price) {
            // Fyll i egen kod.
        }

        /** Genomför affären med ordrarna buyOrder och sellOrder. */
        private void execute(Order buyOrder, Order sellOrder) {
            //Färdig att använda
        }

        // övriga metoder i klassen
    }
```

c) Vad får metoden `addBuyOrder` för tidskomplexitet?