Lösningsförslag

Använda prioritetsköer

U 1. a)
```
public class Patient implements Comparable<Patient>  {

    ...

    public int compareTo(Patient other) {
        return Integer.compare(prio, other.prio);
    }
}
```

b) Lösning med explicit komparatorklass:

```
public class PrioComparator implements Comparator<Patient> {
    public int compare(Patient p1, Patient p2) {
        return Integer.compare(p1.getPrio(), p2.getPrio());
    }
}
```

Ett objekt av typen PrioComparator skickas med när prioritetskön skapas:

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

Om man istället använder lambdauttryck slipper man skriva komparatorklassen och beskriver istället hur jämförelsen ska gå till i lambduttrycket:

```
PriorityQueue<Patient> pq = new PriorityQueue<Patient>(
                        (p1, p2) -> Integer.compare(p1.getPrio(), p2.getPrio()));
```

c)
```
public class Patient {
    private static int total = 0;
    private String firstname;
    private String lastname;
    private String personNbr;
    private int prio;
    private int number;

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

    public int getNumber() {
        return number;
    }

    ...
}
```

Lösning med explicit komparatorklass:

```
public class PrioComparator implements Comparator<Patient> {
    public int compare(Patient p1, Patient p2) {
        if (p1.getPrio() == p2.getPrio()) {
            return Integer.compare(p1.getNumber(), p2.getNumber());
        } else {
            return Integer.compare(p1.getPrio(), p2.getPrio());
        }
    }
}
```

Prioritetskön skapas som förut:

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

Om man istället använder lambdauttryck kan man skapa prioritetskön så här:

```
PriorityQueue<Patient> pq = new PriorityQueue<Patient>((p1, p2) -> {
    if (p1.getPrio() == p2.getPrio()) {
        return Integer.compare(p1.getNumber(), p2.getNumber());
    } else {
        return Integer.compare(p1.getPrio(), p2.getPrio());
    }
});
```

Alternativ:

```
PriorityQueue<Patient> pq = new PriorityQueue<Patient>((p1, p2) ->
        p1.getPrio() == p2.getPrio() ? Integer.compare(p1.getNumber(), p2.getNumber())
        : Integer.compare(p1.getPrio(), p2.getPrio()));
```

U 2. a)

```
/**
 * 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) {
    this.shareId = shareId;
    buyOrders = new PriorityQueue<Order>((order1, order2) ->
        Double.compare(order2.getPrice(), order1.getPrice()));
    sellOrders = new PriorityQueue<Order>((order1, order2) ->
        Double.compare(order1.getPrice(), order2.getPrice()));
}

/**
 * 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) {
    if (! sellOrders.isEmpty() && buyOrder.getPrice() >=
                    sellOrders.peek().getPrice()) {
        return sellOrders.poll();
    }
    buyOrders.offer(buyOrder);
    return null;
}
```

Alternativ lösning:
Man kan skriva två klasser som implementerar `Comparator`:

```
public class PriceComparator implements Comparator<Order> {
    public int compare(Order order1, Order order2) {
        return Double.compare(order1.getPrice(), order2.getPrice());
    }
}

public class ReversePriceComparator implements Comparator<Order> {
    public int compare(Order order1, Order order2) {
        return Double.compare(order2.getPrice(), order1.getPrice());
    }
}
```

I så fall skapas köerna så här:

```
buyOrders = new PriorityQueue<Order>(new ReversePriceComparator());
sellOrders = new PriorityQueue<Order>(new PriceComparator());
```

b)
```
/**
* 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) {
    Order buyOrder = new Order(price, customer);
    OrderQueues share = q.get(shareId);
    if (share == null) {
        throw new NoSuchElementException();
    }
    Order matchingSellOrder = share.addBuyOrder(buyOrder);
    if (matchingSellOrder != null) {
        execute(buyOrder, matchingSellOrder);
    }
}
```

c) I metoden `addBuyOrder` utförs `peek` (som kostar $O(1)$) och sedan `poll` eller `offer` (som kostar $O(logn)$). Den totala tidskomplexiteten blir alltså $O(logn)$.