Programmering uppgift 2

1dt907 algoritmer

Emil Ulvagården (eu222dq)

How to compile and run the programs

Förflytta dig in i mappen lab2 och kör sedan:

```
cd .\Algorithm\

javac *.java

cd ..

javac *.java

java .\Main.java
```

Uppgift 1

Jag börjar med att skapa en array med storleken 10 där element sätts in i. Jag placerar totalt ut 1 000 000 personer i kön med slumpmässigt utvalda prioriteter från 0 till 1 000. När enqueue kallas och antalet element är den samma som storleken på arrayn växer arrayns storlek med en factor av 2 och sen sätts nästa element in. Det första som dequeue kollar är så att arrayn inte är tom för då kan inte ett elemet plockas bort. Om Arrayn har en eller fler element i sig tar dequeue bort ett slumpmässigt utvalt element för att sedan minska sedan kolla ifall antalet element är samma som 1/4 av arrayns storlek.

Enqueue ser ut på följande sett

```
public void enqueue(Obj element) {
    if (size == queue.length) {
        resize(2* queue.length);
    }
    queue[size++] = element;
}
```

Medan dequeue ser ut på följande sett

```
public Object dequeue() {
    if (isEmpty()) {
        throw new NoSuchElementException("No elements in the queue");
    }
    int randomIndex = random.nextInt(size);
```

```
    Object removedElement = queue[randomIndex];

    queue[randomIndex] = queue[size - 1];
    queue[size - 1] = null;

    size--;

    if (size > 0 && size == queue.length / 4) {
        resize(queue.length / 2);
    }

    return removedElement;
}
```

Uppgift 2

Skillnaden i tids kostnad mellan de olika träden är 10 miljoner noder är insättning billigare i BST trädet med 1974.76 milli sekunder i mitt testfall. Att söka upp en nod går snabbare i AVL trädet och skillnaden mellan träden är 11.2 micro sekunder för en slumpvis utvald nod. För delete är BST trädet snabbare i testfallet och skillnaden mellan träden ligger på 16.1 micro sekunder. BST trädet har en höjd på 49 medan AVL trädet har en höjd på 24. I det värsta fallet med 10_000 sorterad värden är BST 153.6 millisecunder långsammare än AVL trädet. För metoden Search är AVL trädet 628 micro sekunder snabbare än BST trädet. Delete är BST 4.2 microsekunder snabbare med. AVL trädet har en höjd på 14 medan BST har en höjd av 10_000. I det bästa fallet kommer BST vara snabbare och lika balanserat då trädet inte behöver kolla så det är ballanserat varje gång som en nod sätts in eller tas bort. Tiden för en search bör vara samma.

uppgift 3

De metoder som har implementerats är insertPerson, getPerson, deleteMaxPriority och swapPriority. För insertPerson Kollar vi först om storleken på vår array är samma som antalet personer i vår kö, ifall dem är samma så dubbleras storleken på arrayn. Sen sätts personen in i kön.

```
public void insertPerson(String name, int priority, int time) {
    if (size == queue.length) {
        resizeQueue();
    }

    Person person = new Person(name, priority, time);
    queue[size++] = person;
}
```

getPerson visar den person som står först i kön. Metoden börjar med att först sortera kön med en heapSort för att sedan returnera personen som har högst prioritet och har stått i kön längst.

```
public Person getPerson() {
    heapSort();
```

```
    return queue[0];
}
```

deleteMaxPriority kollar först ifall det är någon individ som står i kön och ifall det är någon i kön sorteras kön med heapSort för att sedan plocka bort personen som hamnar längst fram i kön. Efter borttagningen av personen längst fram sorteras kön om igen med heapSort.

```
public void deleteMaxPriority() {
    if (size > 0) {
        heapSort();
        swap(0, size-1);
        size--;
        heapSort();
    }
    else {
        System.out.println("Queue is empty.");
    }
}
```

swapPriority börjar med att leta igenom kön efter de angivna namnen där prioriteten ska bytas, om namnen inte hittas sker inget, om namnen hittas byts prioriteterna hos individerna. Kön sorteras sedan om för att individerna där prioriteten byttes ska hamna på rätt plats.

```
public void swapPriority(String name1, String name2) {
    int index1 = -1;
    int index2 = -1;
    for (int i = 0; i < size; i++) {
        if (queue[i].name.equals(name1)) {
            index1 = i;
        } else if (queue[i].name.equals(name2)) {
            index2 = i;
        }
    }

    if (index1 != -1 && index2 != -1) {
        int temp = queue[index1].priority;
        queue[index1].priority = queue[index2].priority;
        queue[index2].priority = temp;

    }
    heapSort();
}
```

heapSort börjar med att bygga en max heap och kallar heapify för att först kolla ifall prioriteten är större hos ena personen i kön. Ifall prioriteten är större byter personerna plats med varandra. Om deras prioritet är samma jämförs tiden som de har varit i kön, den som har varit i kön längst placeras före den andra individen.

```
    private void heapSort() {

        for (int i = size / 2 - 1; i >= 0; i--) {
            heapify(size, i);
        }

        for (int i = size - 1; i >= 0; i--) {
            swap(0, i);
            heapify(i, 0);
        }
    }

    private void heapify(int n, int index) {
        int l = 2 * index + 1;
        int r = 2 * index + 2;
        int s = index;

        if (l < n && queue[l].priority > queue[s].priority) {
            s = l;
        }
        else if (l < n && queue[l].priority == queue[s].priority) {
            if (queue[l].time > queue[s].time) {
                s = l;
            }
        }

        if (r < n && queue[r].priority > queue[s].priority) {
            s = r;
        }
        else if (r < n && queue[r].priority == queue[s].priority) {
            if (queue[r].time > queue[s].time) {
                s = r;
            }
        }

        if (s != index) {
            swap(index, s);
            heapify(n, s);
        }
    }
```

Tiden för en insättning för en person när 1 000 000 redan står i kön är 151.6 micro sekunder. Att hämta ut personen som står först tar 715.3 milli sekunder och att ta bort personen som tidigare stog först i kön tar 748.5 milli sekunder. Att byta prioriteten på två slumpmässigt utvalda personer tog 384.9 milli sekunder i testfallet.

Dem 10 första i en kön är:

```
Name: Person6022, Priority: 0, Time: 6022
Name: Person6602, Priority: 0, Time: 6602
```

```
Name: Person7407, Priority: 0, Time: 7407
Name: Person7614, Priority: 0, Time: 7614
Name: Person8699, Priority: 0, Time: 8699
Name: Person10018, Priority: 0, Time: 10018
Name: Person10687, Priority: 0, Time: 10687
Name: Person10862, Priority: 0, Time: 10862
Name: Person11545, Priority: 0, Time: 11545
Name: Person11657, Priority: 0, Time: 11657
```

Uppgift 4

Testning börjar med att skapa en array med 10 000 slumpmässigt utvalda tal och sedan köra med quickSort med djup från 0 till 200 med inkrement av 5. När det uppvalda djupet är nått skapas två koppior av arrayn för att sedan sortera med både heap- och insertSort. Den sorteringsmetod som är snabbast får sedan 1 poäng för att i slutet se vilken som gjorde bäst i totalen. Testningen fortsätter sedan med att ändra antalet element till 50 000 följt av 100 000 och 200 000 ocj slutligen följt av 500 000.

Tabellen nedan visar hur många element som har använts samt de rekomenderade djupet för de olika sorterings algoritmerna och hur många gånger de olika algoritmerna vann. Utifrån antalet vinster för de olika sorterings algoritmerna borde heap användas.

Antal Element	**Rek. Djup för Heap**	**Rek. Djup för Insertsort**	**vinster för heap**	**vinster för insert**
10 000	0	120	41	0
50 000	0	90	41	0
100 000	0	190	41	0
200 000	0	95	41	0
500 000	0	90	41	0