UNIVERSITEIT GENT

Algoritmen en Datastructuren 2

Project Wachtlijnen

Deadlines: 29 oktober en 26 november 2017

1 Inleiding

Wachtlijnen worden vaak gebruikt om bepaalde taken of objecten bij te houden samen met een bepaalde prioriteit. Meestal is het dan de bedoeling dat je heel efficiënt de taak kan opvragen die als volgende gedaan moet worden – in de cursus noemen we dit steeds het minimale element, dus hoe lager de waarde van de sleutel, hoe hoger de prioriteit. In de praktijk is zo'n prioriteit echter niet iets statisch: in de loop van de tijd kan een bepaalde taak dringender of minder dringend worden, of zelfs niet meer nodig. Wanneer dit gebeurt, zou je liefst de prioriteit kunnen aanpassen in de wachtlijn. De opgave van het project dit jaar gaat over de algoritmen die vereist zijn om deze aanpassingen te kunnen doorvoeren in de verschillende types wachtlijnen.

2 Opgave

Zoals hoger beschreven is de opgave van het project het opstellen en implementeren van algoritmen om de prioriteit van elementen in een wachtlijn aan te passen of elementen te verwijderen, en dit voor verschillende types wachtlijnen. We zullen de volgende types van wachtlijnen beschouwen in dit project: de binaire hoop, de binomiale wachtlijn, de leftist heap, de skew heap en de pairing heap. De eerste vier wachtlijnen heb je reeds gezien in deze of vorige cursussen. De laatste is echter een nieuw type dat we hieronder beschrijven.

2.1 Pairing heaps

Een niet-lege pairing heap is een boom met een eenvoudige recursieve structuur.

Definitie 1 *Een pairing heap is een boom waarin de wortel kleiner is dan al zijn kinderen, en voor elk kind y van de wortel is de maximale deelboom met als wortel y opnieuw een pairing heap.*

Net zoals bij sommige andere wachtlijnen die we reeds gezien hebben, kunnen we bij een pairing heap de meeste bewerkingen beschrijven aan de hand van een of meerdere merge-bewerkingen. Om twee pairing heaps H_1 en H_2 samen te voegen, gaan we als volgt te werk:

- Vergelijk de wortels van H_1 en H_2 en stel $H_{\min}$ de heap met kleinste wortel, en $H_{\max}$ de andere heap.
- Voeg de wortel van $H_{\max}$ toe als eerste kind van de wortel van $H_{\min}$.

Om een sleutel toe te voegen aan een pairing heap H, maak je dan gewoon een nieuwe pairing heap aan die enkel die sleutel bevat, en vervolgens voeg je deze heap samen met H. Dit kan in constante tijd.

Het kleinste element is altijd de wortel van de pairing heap, dus dit kan gevonden worden in constante tijd.

Om het kleinste element te verwijderen uit een pairing heap H, verwijder je de wortel en voeg je daarna alle kinderen samen. Aangezien de merge-bewerking enkel beschreven is voor twee heaps, zal je deze herhaaldelijk moeten toepassen. Let op want er zijn verschillende manieren om dit te doen, en de geamortiseerde complexiteit is hiervan afhankelijk. De standaard manier is om de kinderen van de oude wortel eerst in paren van links naar rechts samen te voegen, en vervolgens deze paren allemaal tot één boom samen te voegen van rechts naar links.

Voor de *decrease-key* operatie (het verlagen van de prioriteit van een element) verwijder je de deelboom die dit element als wortel heeft. Vervolgens kan je de prioriteit verlagen, en de deelboom mergen met de pairing heap.

De geamortiseerde kost van een reeks van n bewerkingen op een initieel lege pairing heap – bestaande uit toevoegbewerkingen, het verwijderen van het kleinste element en het verlagen van de prioriteit van een element – is $O(n \log n)$ of dus gemiddeld $O(\log n)$ per bewerking. De geamortiseerde kost van een reeks van n toevoegbewerkingen op een initieel lege pairing heap is $O(n)$ of dus gemiddeld $O(1)$ per toevoegbewerking.

2.2 Aanpassen en verwijderen van elementen

Als we de prioriteit willen aanpassen van een bepaald element, of een element willen verwijderen, dan moeten we een referentie hebben naar dit element. Om die reden zullen jouw implementaties van de verschillende wachtlijnen steeds een referentie naar het element moeten teruggeven wanneer je dit element toevoegt aan de wachtlijn. Deze referentie kan je dan gebruiken om het element later aan te passen. We helpen je hier reeds in door interfaces te voorzien waaraan je moet voldoen, maar hierover volgt later nog meer uitleg.

Let erop dat de prioriteit zowel verhoogd als verlaagd kan worden, en dat dit allebei efficiënt moet gebeuren.

Nadat de operatie is uitgevoerd, moet de heap eventueel geherstructureerd worden om te zorgen dat nog steeds aan de geldende voorwaarden voor de specifieke wachtlijn zijn voldaan.

Merk op dat voor een pairing heap het verlagen van de prioriteit een standaardbewerking is, en deze is dan ook reeds beschreven hierboven. Vergeet echter niet dat aanpassen niet hetzelfde is als verlagen.

2.3 Opdracht

Beantwoord onderstaande theoretische vragen. Implementeer de verschillende algoritmen zoals beschreven in Sectie 4. Let daarbij heel goed op de complexiteit en efficiëntie. Voer experimenten uit volgens Sectie 5 en schrijf een verslag volgens Sectie 6.

3 Theoretische vragen

Geef voor elk van de vijf types van wachtlijnen een zo efficiënt mogelijk algoritme voor het aanpassen van sleutels en verwijderen van elementen, en bepaal en bewijs de complexiteit van deze algoritmes waarbij je mag steunen op reeds genoemde resultaten in de cursus of de uitleg hierboven.

Het theoretische gedeelte is minstens even belangrijk als de implementatie, dus besteed er voldoende tijd aan. Geef voor elk algoritme een duidelijke beschrijving met pseudocode. Implementatiedetails en Java-specifieke dingen horen hier niet thuis. Zorg ervoor dat je bewijs volledig en correct is. Je mag gebruik maken van resultaten uit de cursus, of voor de pairing heap uit Sectie 2.1. Vermeld altijd welke eigenschappen je gebruikt.

4 Implementatie

Als startpunt krijg je van ons 4 Java-bestanden met respectievelijk twee interfaces en twee klassen.

Al je implementaties van wachtlijnen moeten de interface `heap.Heap` implementeren.

```
package heap;

public interface Heap<T extends Comparable<T>> {
    Element<T> insert(T value);
    Element<T> findMin() throws EmptyHeapException;
    T removeMin() throws EmptyHeapException;
}
```

De elementen die teruggegeven worden wanneer een element toegevoegd wordt aan een wachtlijn moeten allemaal de interface `heap.Element` implementeren.

```
package heap;

public interface Element<T extends Comparable<T>> {
    T value();
    void remove();
    void update(T value);
}
```

We voorzien ook een foutklasse `heap.EmptyHeapException` die kan gegooid worden als het kleinste element uit een lege wachtlijn wordt opgevraagd.

```
package heap;

public class EmptyHeapException extends Exception {}
```

Ten slotte geven we je ook de volgende klasse die aangeeft hoe jouw code aangeroepen moet worden. Let voornamelijk op de namen van packages en klassen die deze code impliceert. Natuurlijk plaats je ook andere klassen die je maakt op een logische plaats in deze klassehiërarchie.

```
package heap;

import heap.binary.BinaryHeap;
```

```java
import heap.binomial.BinomialHeap;
import heap.leftist.LeftistHeap;
import heap.skew.SkewHeap;
import heap.pairing.PairingHeap;

public class Heaps {
    public static <T extends Comparable<T>> BinaryHeap<T> newBinaryHeap() {
            return new BinaryHeap<>();
    }

    public static <T extends Comparable<T>> BinomialHeap<T> newBinomialHeap() {
            return new BinomialHeap<>();
    }

    public static <T extends Comparable<T>> LeftistHeap<T> newLeftistHeap() {
            return new LeftistHeap<>();
    }

    public static <T extends Comparable<T>> SkewHeap<T> newSkewHeap() {
            return new SkewHeap<>();
    }

    public static <T extends Comparable<T>> PairingHeap<T> newPairingHeap() {
            return new PairingHeap<>();
    }
}
```

Voorzie je code van voldoende commentaar en geef elke methode een correcte javadoc header. Geef bij elke niet-triviale methode in de javadoc header aan wat de bedoeling is van deze methode en wat eventuele neveneffecten zijn.

Implementeer JUnit tests om jouw implementatie op correctheid te testen en zorg dat je alle onderdelen van jouw implementatie goed test en niet alleen de functies voor het aanpassen van de sleutels. Schrijf je testen zo vroeg mogelijk tijdens het project zodat je geen tijd verliest met het optimaliseren en meten van een foute implementatie.

5 Experimenten

Beschrijf welke experimenten je hebt uitgevoerd om verschillende ideeën die je had te vergelijken, en de correctheid van je implementatie te testen. Ga ook de complexiteit van de algoritmen op experimentele wijze na, en vergelijk de prestaties van de verschillende types wachtlijnen.

6 Verslag

Schrijf een verslag van dit project. Beantwoord eerst de theoretische vragen en beschrijf dan jouw implementaties, waaronder mogelijke optimalisaties die je hebt doorgevoerd. Voeg de resultaten van je experimenten op overzichtelijke wijze toe. Wat kun je op basis van de experimenten concluderen? Komen je resultaten overeen met je verwachtingen? Probeer steeds om al je resultaten te verklaren.

7 Beoordeling

De beoordeling van het ingeleverde werk zal gebaseerd zijn op de volgende onderdelen:

- Werden de theoretische vragen goed beantwoord? Dit weegt zwaar door.
- Is de implementatie volledig? Is ze correct?
- Is er zelf nagedacht? Is alles eigen werk?
- Hoe goed presteert jouw algoritme?
- Hoe is er getest op correctheid?
- Zijn de experimenten goed opgezet?
- Is het verslag toegankelijk, duidelijk, helder, verzorgd, ter zake en objectief?

8 Deadlines

Er zijn twee indienmomenten:

1. Zondagavond 29 oktober 2017 om 23u59 moet de code en het verslag voor de binaire hoop, binomiale wachtlijn en leftist heap ingeleverd worden. Je mag hieraan nog wel wijzigingen doen tegen het volgende indienmoment.
2. We verwachten een volledig project tegen zondagavond 26 november 2017 om 23u59.

Code en verslag worden elektronisch ingediend. Van het verslag van het **tweede** indienmoment verwachten we ook een **papieren versie**. Nadien zal je mondeling je werk verdedigen. Het tijdstip hiervoor wordt later bekendgemaakt.

9 Elektronisch indienen

Op https://indianio.ugent.be/ kan elektronisch ingediend worden. Maak daartoe een ZIP-bestand en hanteer daarbij de volgende structuur:

- verslag.pdf is de elektronische versie van je verslag in PDF-formaat.
- De package `heap` bevat minstens de klassen gespecifieerd hierboven. Alle code bevindt zich onder deze package.
- De directory test bevat alle JUnit-tests.

Bij het indienen wordt gecontroleerd of je code voldoet aan de gevraagde structuur en interfaces, en of ze uitgevoerd kan worden. Wacht dus niet tot het laatste moment om een eerste keer in te dienen, enkel de laatste versie wordt verbeterd.

10 Algemene richtlijnen

- Zorg ervoor dat alle code compileert met Oracle Java 8.
- Extra externe libraries zijn niet toegestaan. De JDK en JUnit mogen wel.
- Niet-compileerbare code en incorrect verpakte projecten **worden niet beoordeeld**.
- Het project wordt gequoteerd op **4** van de 20 te behalen punten voor dit vak, en deze punten worden ongewijzigd overgenomen naar de tweede examenperiode.
- Projecten die ons niet bereiken voor beide deadlines worden niet meer verbeterd: dit betekent het verlies van alle te behalen punten voor het project.
- Dit is een individueel project en dient dus door jou persoonlijk gemaakt te worden. Het is niet toegestaan om code uit te wisselen of over te nemen van het internet. We gebruiken geavanceerde plagiaatdetectiesoftware om ongewenste samenwerking te detecteren. Zowel het gebruik van andermans werk, als het delen van werk met anderen, zal als examenfraude worden beschouwd.