2016

Verslag opdracht 2

VICTOR VAN DE RIET & ROB LOGTENBERG

Omdat onze vorige inlever poging niet helemaal correct was, hebben we besloten om alles helemaal opnieuw te doen. Want het aanpassen van de code leek ons te veel werk.

Ook hebben we besloten om een aparte .txt bestanden te gebruiken zodat we de invoer en uitvoer beter konden vergelijken.

Analyse

Onze verwachting is dat de snelheid van het invoegen van de data in de heap, O(log(N)) zal zijn. Dit komt omdat in de worst-case door de log van het aantal elementen van de boom gelopen moeten worden. Daarnaast worden ook veel elementen geinsert in de huildheap methode. Hierin is de snelheid O(N). De grootste delen van de inserts zullen ook de snelheid van O(N) hebben, omdat we eerst alles toevoegen en daarna in een heap “formaat” zetten.

FileSorter

De filesorter heeft als argumenten (File file, int heapSize), hiermee kun je makkelijk vanuit de mainclass een heap aanmaken met de gegenereerde file. Bij het aanmaken van een FileSorter wordt een een heap gemaakt met de heapsize + 1 omdat op de eerste positie het laatst verwijderde element wordt bijgehouden en het eigenlijk geen onderdeel is van de heap of van de deadspace.

Na het aanmaken van een nieuwe heap wordt er een scanner genaamd reader gemaakt die de input file inleest. De reader wordt vervolgens uitgelezen in de methode fillHeapFromReader(). Deze methode vult eerst de deadspace en vervolgens wordt de heap gevuld met de methode buildMinHeap().

Vervolgens wordt de methode doRun() aangroepen, deze methode zorgt ervoor dat zolang er nog elementen in de heap zitten deze elementen worden uitgeprint. Ook wordt er gekeken of er nog elementen in de inputfile staan, deze worden indien dit het geval is ook toegevoegd aan de heap. Het opvallende bij deze class is dat er een overflowException gegooid kan worden. Dit is het geval wanneer de array al vol zit en er toch nog geprobeerd wordt om een element toe te voegen.

BuildMinHeap

De methode die de heap gaat bouwen, deze methode haalt eerst alle elementen uit de deadspace naar voren, in de meeste gevallen zullen alle elementen al vooraan staan omdat de deadspace in dat geval vol staat. Deze methode gaat vervolgens ieder element af en probeert constant tryPercolateDown aan te roepen. Als alle elementen zijn geweest, dan is de heap klaar en staan de elementen op de juiste plek.

PercolateDown

Deze methode verwisselt het meegegeven item met zijn kleinste kind. Deze methode wordt vanuit de buildMinHeap aangeroepen om alle elementen op de juiste plek te krijgen. Roept ook tryPercolateDown aan, deze methode kijkt of hij nog verder naar beneden kan om de percolateDown methode nogmaals aan te roepen. In de percolateDown methode wordt constant gecontroleerd of er childs onder het meegegeven elementen zitten en wat de sleutelwaarde van deze elementen is.

RSHeap

De RSHeap is een heap gecombineerd met een deadspace. Als er items worden toegevoegd aan de heap, wordt er gekeken of de nieuwe items groter of gelijk zijn aan het laatst verwijderde item. Als

dit het geval is dan komt het vergeleken item in de heap te staan, is dit niet het geval dan komt dit item in de deadspace. De deadspace kan vervolgens weer worden omgezet in een heap.

Belangrijke code:

```
heap = new RSHeap<>(heapSize + 1);
```

Er wordt een een heap gemaakt met de heapsize + 1 omdat op de eerste positie het laatst verwijderde element wordt bijgehouden en het eigenlijk geen onderdeel is van de heap of van de deadspace. Dit element wordt gebruikt om te beslissen of het volgende element: in de heap of in de deadspace komt.

```
heapLength / 2
```

Het delen van de heapLength door 2 gebeurt in de buildMinHeap methode, hierdoor kom je op de helft van alle items terecht. Vanaf hier kan geloopt worden om alle elementen tryPercolateDown() te doen.

Implementatie

De implementatie wordt in de bijlage meegeleverd, de code zal ook Javadoc en een drietal JUnit-tests bevatten.