Sorting & Searching

Opdracht 1 – QuickSort

Naam (studentnummer)	Daan Befort (500727616)
Naam (studentnummer)	Kevin Huijzendveld (500727925)
Datum	30/11/18

Table of Contents

Inleiding *3*

QuickSort Array *4*

Sort 4

Partition 4

Less 5

Exch 5

QuickSort LinkedList *6*

Mogelijke verbeteringen *7*

Inleiding

Voor deze opdracht implementeren twee verschillende soorten van QuickSort. De eerste implementatie is de QuickSort die een Array kan soorten. De tweede implementatie van de QuickSort is voor een LinkedList. Deze twee implementaties zullen we grondig stap voor stap uitleggen en hierbij zullen code snippets aanwezig zijn om de uitleg te verduidelijken.

Naast de implementaties en de uitleg hiervan zullen we ook aangeven hoe efficiënt beide sorteringen zijn. Daarnaast zullen de twee implementaties ook met elkaar vergelijken en kijken wat de sterke en zwakke punten hiervan zijn. Ook zullen we kijken of het beter is om een LinkedList te vervangen voor een ArrayList,

We hebben onderzoek gedaan naar mogelijke implementaties om onze sortering te verbeteren hierbij zijn we uitgekomen op een techniek die de performence van de sortering zal verbeteren. Er hangt hier wel een nadeel aan, deze zullen we ook bespreken.

Vervolgens zullen we afsluiten met een kleine afsluiten over of deze opdracht onze visie op Arrays en ADT’s zoals LinkedList en ArrayList.

QuickSort Array

In dit onderdeel zullen we uitleggen hoe we QuickSort hebben geïmplementeerd om een Array te sorteren.

Om te beginnen kijken we natuurlijk of de lijst niet leeg is, als deze leeg is geven we de lijst gelijk terug want dan valt er niks aan te sorteren. Als dit niet het geval is roepen we de methode sort aan met de ongesorteerde Array, de waarde nul omdat dit de laagste waarde van de Array is en we sturen als laatste de hoogste waarde van de array mee.

Sort

De methode sort verwacht dus drie waardes de Array, de laagste waarde en de hoogste waarde. Als eerst wordt er gecheckt of de hoogste waarde net kleiner of gelijk is aan de laagste waarde want als dit het geval is dan is de hele array al gesorteerd en dan kan deze worden terug gegeven. Is dit niet het geval dan gaan we eerst door middel van de partition methode wordt de Array opgedeeld in twee kleine Arrays. Vervolgens gaan we door middel van recursie de twee kleine Arrays, ook wel de linker- en rechterkant, sorteren. Dit proces blijft zich herhalen totdat de hoogste waarde gelijk is of kleiner is dan de laagste waarde en je dus een Array hebt die volledig gesorteerd is.

```
public static <E extends Comparable<E>> E[] sort(E[] a, int
lo, int hi) {
        if (hi <= lo) return a;

        int j = partition(a, lo, hi);
        sort(a, lo, j - 1);
        sort(a, j + 1, hi);
        return a;
    }
```

Partition

De methode partition verwacht net als de Sort methode drie waardes. Ze verwachtte ook beide dezelfde waardes, namelijk de Array, de laagste waarde en de hoogste waarde. Eerst worden er twee variabele aangemaakt deze krijgen de laagste en hoogste waarde, I en J . Vervolgens wordt er nog een variabele aangemaakt en deze bevat de waarde uit de Array op de positie van de laagste waarde.

Met deze variabele genaamd V scannen we de variabele aan de linker en rechterkant. Hij scant de linkerkant tot er een waarde is gevonden die groter is dan V en de rechterkant tot er een waarde is kleiner dan V. Deze worden vervolgens verwisseld. Vanaf de plekken aan de linker en rechterkant waar we gebleven zijn gaan we vervolgens weer verder met scannen. Dit doen we tot I en J naast elkaar staan en dan ruilen we V met I. Op dit moment staan er geen waardes meer aan de linkerkant van V die groter zijn dan V en aan de rechterkant die kleiner zijn dan V.

```
private static <E extends Comparable<E>> int partition(E[] a,
int lo, int hi) {
        int i = lo;
```

```
        int j = hi + 1;
        E v = a[lo];
        while (true) {
            while (less(a[++i], v)) {
                if (i == hi) {
                    break;
                }
            }
            while (less(v, a[--j])) {
                if (j == lo) {
                    break;
                }
            }
            if (i >= j) {
                break;
            }
            exch(a, i, j);
        }
        exch(a, lo, j);
        return j;

    }
```

Less

De methode less vergelijkt de twee players waarmee de methode wordt aangeroepen. Het vergelijken gebeurt met de compareTo methode die wij hebben geschreven in de class Player. De methode verwacht twee players en vergelijkt dan dus de twee players en geeft daarna een waarde terug of de ene player onder of boven de andere player hoort.

```
private static <E extends Comparable<E>> boolean less(E v, E
w) {
        return v.compareTo(w) < 0;
    }
```

Exch

De methode exch wordt gebruikt om twee players van plaats te verwisselen. Bij het aanroepen worden er drie waardes verwacht, de array en twee players. Er wordt eerst een tijdelijke variabele aangemaakt waar de eerste player in wordt opgeslagen. Daarna wordt de andere player op de plek van de eerste player gezet. Vervolgens word de tijdelijke variabele gebruikt om de eerste player op de plek van de twee player te zetten.

```
private static <E extends Comparable<E>> void exch(E[] a, int
i, int j) {
        E t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
```

QuickSort LinkedList

In dit onderdeel zullen we uitleggen hoe we QuickSort hebben geïmplementeerd om een LinkedList te sorteren.

Voor het sorteren van een LinkedList hebben we dezelfde logica als voor het sorteren van een Array toegepast. Dit betekent dat we gebruik hebben gemaakt van dezelfde methodes maar hierbij alleen veranderd dat de methodes geen Array maar een LinkedList verwachten.

De less methode hebben we zelfs kunnen hergebruiken omdat deze alleen om twee items vraagt om te vergelijken. In de exch methode gebruiken we de functie get om de waarde uit de LinkedList te krijgen en de functie set om een niet waarde in de LinkedList toe te voegen. Deze functies komen vanuit de LinkedList zelf.

De uitleg en code snippets hierbij zijn dus terug te vinden onder het kopje QuickSort Array. Hier wordt alle logica die we gebruiken om te sorteren stap voor stap uitgelegd.

Mogelijke verbeteringen

In deze sectie zullen we uitleggen welke verbetering we nog aan onze sortering kunnen doen waarvan we van zijn dat deze ook een positief effect heeft op de efficiëntie.

Een verbetering op onze huidige sortering is het gebruiken van een median-of-three. Hierbij pak je de eerste, middelste en laatste waarde uit de Array of LinkedList en van deze deze drie waardes bereken je het gemiddelde, de median dus, deze gebruik je als pivot. Nu geef je dus aan de partition een extra waarde mee dat is dus de pivot. Je gaat nu dus vanuit de pivot uit vergelijken met de eerste van links.

Hierdoor verhoog je de snelheid van het sorteren. Dit je dus een beter partition maar dit gaat wel ten koste van extra bereken tijd. Er is een extra methode nodig om de median te vinden. Bij het kiezen zullen eerst ook de drie waardes gesorteerd moeten worden. Dit heeft geen effect op de median als deze voor het eerst wordt gekozen.

Als je dit proces herhaald voor het kiezen van de volgende median van de linkerkant Array of LinkedList heeft dit zeker wel effect. Dit komt omdat als je niet sorteert bij het kiezen van een median een grote waarde aan de linkerkant kan blijven staan en een kleine aan de rechterkant hierdoor wordt je pivot aan de linkerkant groter dan als er gesorteerd is en dit verpest het effect van de median of three.

Het is hierbij de bedoeling door het gemiddelde te nemen dat je dichter bij het midden zit. Doordat je dichter bij het midden zit zal je de Array of LinkedList dus ook sneller gesorteerd worden.

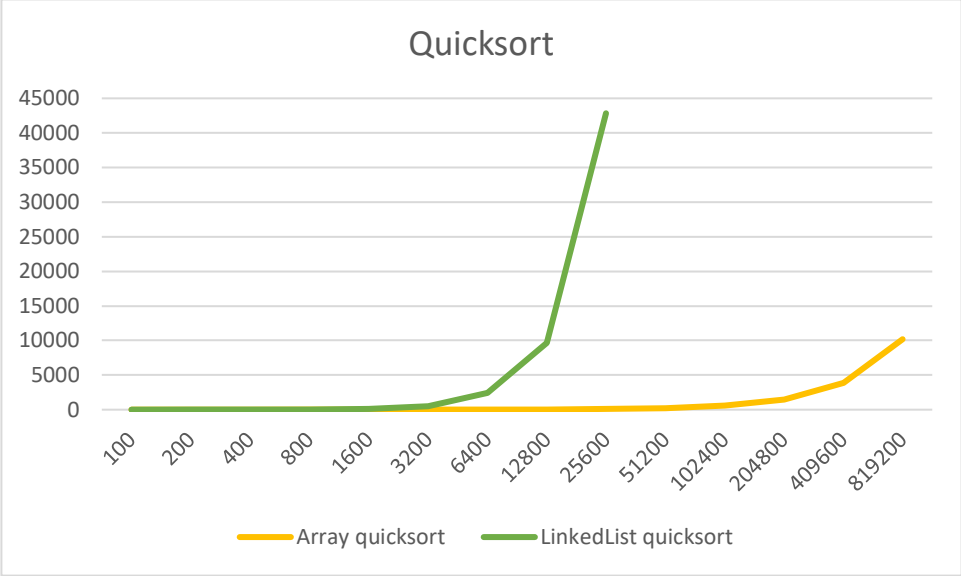
Metingen voor quicksort

Nadat we alles hebben geïmplementeerd zoals we hierboven hebben beschreven zijn we de quicksorts gaan testen op snelheid. We beginnen met 100 players en sorteren de players op highscore, achternaam, voornaam. Wanneer dit klaar is gaan we de aantal players verdubbelen. Bij elke sortering kijken we hoelang het duurt. Hieronder vind je een tabel met alle gegevens die uit deze testen naar voren zijn gekomen voor Array quicksort en LinkedList quicksort.

Aantal players	Array quicksort (in miliseconds)	LinkedList quicksort (in miliseconds)
100	0	0
200	0	2
400	0	9
800	1	33
1600	5	127
3200	9	539
6400	21	2453
12800	49	9673
25600	117	42840
51200	255	
102400	600	
204800	1505	
409600	3834	
819200	10183	

Zoals je hierboven kan zien hebben we bij Array quicksort meer players gesorteerd dan bij de LinkedList. Dit komt omdat de sort niet langer dan 20 seconden mag duren en maar maximaal 1.048.576 players sorteren. Wanneer dit wel het geval is zal erna niet nog een keer gesorteerd worden.

Ook kunnen we uit de bovenstaande metingen concluderen dat de Array quicksort sneller is dan de LinkedList quicksort. Array quicksort kan 819200 players sorteren onder de 20 seconden en de LinkedList quicksort kan maar 12800 sorteren binnen 20 seconden. Hieronder heb ik een tabel gemaakt waar je duidelijk het verschil kan zien.



The Big – O

Om de Big – O te bepalen moeten wij als eerst het aantal iteraties bij een sorteer poging tellen. Wij gberuiken meerdere keren een loop, hierdoor hebben vallen O(1) of O(N) weg. Deze zijn pas van toepassing als de efficiëntie van de toepassing constant of lineair stijgt wat met deze loops niet van toepassing is.

Wat wij in gedachten moeten houden is dat de sorteermethode die wij geschreven hebben recursief is, wat dus betekent dat wij in onze efficiëntie al een “loop” hebben, binnen deze methode doen wij twee while loops wat ons algoritme zou brengen op O(N^2), een kwadratisch verloop is het gevolg, dit zien wij ook terug als wij terugkijken naar onze grafieken.

Conclusie

De conclusie is dat een normale array veel sneller is dan een LinkedList. Zoals je kan zien in het hoofdstuk: *Metingen voor quicksort*. De normale array kan veel meer in veel kortere tijd sorteren.

Hieruit kan je ook herleiden dat ADT'S niet altijd beter zijn om te gebruiken. Ze doen wel wat ze moeten doen, maar niet altijd op de meest efficiënte manier die je nodig hebt.
Dit komt mede doordat bijvoorbeeld de ADT LinkedList allemaal extra functionaliteiten heeft die we niet nodig hebben maar wel extra tijd kosten. En een normale array heeft geen extra functionaliteiten, dus de sortering die je uitvoert is alles wat er gebeurt.

Dus mijn andere conclusie voor array vs ADT's is dat array in dit geval veel beter is.