Data Structures

Assignment 2

door

BURAK INAN
500740123

Dit document betreft het verslag van Assignment 2 van Data Structures.

HBO-ICT Software Engineering
Hogeschool van Amsterdam
Amsterdam, Nederland

6/12/2018

Inhoudsopgave

Inleiding

In dit verslag wordt beschreven op welke manier de sorteerfuncties, en de priority queue functie worden geïmplementeerd. Ook wordt de efficiëntie berekend van de verschillende implementaties. Hierbij wordt er gekeken naar hoeveel tijd het kost om bepaalde functies uit te voeren. Uiteindelijk worden alle berekeningen gevisualiseerd.

Approach 1 - Variant 1: Insertion Sort

Theorie

Dit simpele sorteer algoritme werkt door telkens 1 element te vergelijken met de elementen vóór het element. De tijd complexiteit is O(n^2) in het slechtste geval, en O(n) in het beste geval.

Mijn implementatie

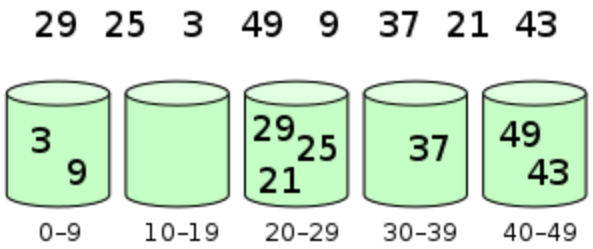
Stel we hebben een ongesorteerde lijst met de volgende highscores:
Speler1: 100, Speler2: 5, Speler3: 2, Speler4: 999, Speler5: 14000, Speler6: 365

- **We slaan twee dingen tijdelijk op: Het Player object, en de highScore.**
  ```
  long highScore;
  Player temp;
  ```
- **Vervolgens gaan we spelers met elkaar vergelijken:**
  - **We beginnen bij het tweede element in de lijst, omdat we deze met het element ervoor gaan vergelijken. Dus we beginnen bij Speler2.**
    ```
    for (int i = 1; i < list.size(); i++) {
        highScore = list.get(i).getHighScore();
        int j = i - 1;
    ```
  - **We kijken of de highScore van de speler waar we nu op staan groter is dan die daarvoor. De highScore van Speler2 is 5, dus deze is kleiner.**
    ```
    while (j >= 0 && highScore > list.get(j).getHighScore()) {
    ```
  - **Als dat zo is, wisselen we de positie van de spelers.**
    ```
    temp = list.get(j);
    list.set(j, list.get(j + 1));
    list.set(j + 1, temp);
    ```
  - **Zo niet, gaan we naar het volgende element in de lijst. In ons geval is dat dus niet zo, en gaan we naar de volgende speler.**
  - **Nu staan we op Speler3.**
  - **Deze gaan we vergelijken met Speler2, waaruit we concluderen dat we ze niet hoeven te wisselen. Dus we gaan naar het volgende element.**
  - **Speler4 heeft een hogere score dan Speler3, dus we wisselen de posities van de spelers met elkaar.**
  - **We gaan net zo lang door met het vergelijken van spelers, totdat we aan het einde van de lijst zijn aangekomen, of totdat we een speler tegen zijn gekomen met een highScore hoger dan Speler4.**

Approach 1 - Variant 2: Bucket Sort

Theorie

Dit algoritme sorteert elementen door ze in meerdere 'buckets' te plaatsen. Elke bucket wordt individueel gesorteerd. De tijd complexiteit is O(n^2) in het slechtste geval, en O(n+k) in het beste geval.



Zoals in het figuur hierboven te zien is, heeft elke bucket een bepaald bereik. Zo worden elementen 'gesorteerd' op, in dit geval, de highScore. Uiteindelijk worden deze elementen die in de buckets zijn geplaatst, gesorteerd van hoog naar laag.

Implementatie

Stel we hebben een lijst van 1000 spelers.

- **Eerst maken we een lijst aan die onze gesorteerde elementen zal opslaan.**
  ```
  List<Player> sortedList = new ArrayList<>();
  ```
- **Vervolgens wordt er gekeken of de lijst leeg is. Als dat zo is wordt de gegeven lijst teruggegeven.**
- **Vervolgens bepalen we de min. waarde en max. waarde van de de highScores. Deze zullen we gebruiken om het bereik van de buckets te bepalen.**

```
long maxScore = Integer.MIN_VALUE;
long minScore = Integer.MAX_VALUE;

for (Player p : list) {
    if (p.getHighScore() >= maxScore) {
        maxScore = p.getHighScore();
        minScore = maxScore;
    }

    if (p.getHighScore() < minScore) {
        minScore = p.getHighScore();
    }
}
```

- **Om het aantal buckets te bepalen nemen we de wortel van de grootte van de lijst.**
  ```
  int maxBuckets = (int) Math.sqrt(list.size());
  ```
- **Om het minimum bereik van een bucket te bepalen nemen we de laagste score, en om het hoogste bereik van een bucket te bepalen gebruiken we de maximum score en het maximum aantal buckets.**
  ```
  long bucketRange = maxScore / maxBuckets + 1;
  long minBucketRange = minScore;
  long maxBucketRange = bucketRange;
  ```
- **Bij het aanmaken van buckets, moeten we rekening houden dat in elke bucket een lijst van spelers bevindt.**
  ```
  private class Bucket {
      List<Player> bucket = new ArrayList<>();
  }
  Bucket[] buckets = new Bucket[maxBuckets];
  for (int i = 0; i < maxBuckets; i++) {
      buckets[i] = new Bucket();
  }
  ```
- **Nadat we buckets hebben aangemaakt, plaatsen we de elementen in de buckets. De elementen worden geplaatst in de buckets op basis van de waarde van de highScore. Als een speler een highScore heeft van 500, wordt deze bijvoorbeeld in de bucket geplaatst die een bereik heeft van 1-1000. Als een speler een waarde boven de 1000 heeft, wordt het bereik van de bucket opgeteld, en wordt er gekeken of de speler in de volgende bucket past.**
  ```
  for (int i = 0; i < maxBuckets; i++) {
      for (Player p : list) {
          if (p.getHighScore() >= minBucketRange && p.getHighScore() <=
  bucketRange) {
              buckets[i].bucket.add(p);
          }
      }
      minBucketRange = bucketRange + 1;
      bucketRange += maxBucketRange;
  }
  ```

- **Uiteindelijk wordt elke bucket gesorteerd met behulp van de Insertion Sort algoritme die hiervoor is geïmplementeerd. Na elke sorteer actie wordt de bucket vooraan in de lijst van gesorteerde elementen toegevoegd.**

```
for (Bucket b : buckets) {
    sort(b.bucket);
    sortedList.addAll(0, b.bucket);
}
```

- **Ten slotte retourneren we de gesorteerde lijst.**

```
return sortedList;
```

Approach 2: Priority Queue

Theorie

Een priority queue is een rij elementen, waarbij elk element een eigen prioriteit heeft. In een priority queue gaat altijd het element met de hoogste prioriteit er altijd eerst uit. De tijd complexiteit voor priority queue methodes die elementen toevoegen of verwijderen is O(log(n)), en voor het lezen van elementen is het constant.

Implementatie

Stel we hebben een lijst van 1000 spelers.

Een priority queue hoeft niet gesorteerd te worden. In dit geval heeft de speler met een hoger highScore, een hogere prioriteit, dus wordt deze speler verder bovenaan (vooraan in de lijst) gezet.
Elke keer wanneer er een speler wordt toegevoegd, wordt de toegevoegde speler vergeleken met de opeenvolgende spelers. Wanneer de queue ziet dat de toegevoegde speler een hogere highscore heeft, wordt deze speler op die plek toegevoegd.

- **Eerst wordt er een priority queue aangemaakt en er wordt daarin een methode meegegeven die spelers met elkaar vergelijkt:**

```
PriorityQueue<Player> priorityQueue = new PriorityQueue<>((p1, p2) -> {
    if (p1.getHighScore() > p2.getHighScore())
        return 1;
    if (p1.getHighScore() < p2.getHighScore())
        return -1;
    else
        return 0;
});
```

- **Wanneer de lijst van spelers wordt opgevraagd, worden de elementen die zich in de queue bevinden, een voor een (van kop tot staart) verplaatst naar een ArrayList:**

```
private List<Player> priorityQueueToList() {
    playerList.clear();
    PriorityQueue<Player> priorityQueue = new
PriorityQueue<>(this.priorityQueue);
    while (!priorityQueue.isEmpty()) {
        playerList.add(priorityQueue.poll());
    }
    Collections.reverse(playerList);
    return playerList;
```

FindPlayers

Theorie

In dit hoofdstuk worden de methodes toegelicht die het zoeken van spelers met een bepaalde naam implementeren, aangezien alle methodes op dezelfde manier werken voor alle sorteer algoritmen en de priority queue.

Implementatie

De implementatie van deze methodes is vrij simpel. Het enige wat er gedaan moet worden is:

- **Een nieuwe lijst aanmaken waar alle gevonden spelers in worden opgeslagen.**
  ```
  List<Player> foundPlayers = new ArrayList<>();
  ```
- **De methode kijkt of er een voornaam, achternaam of allebei is meegegeven, en op basis daarvan worden gevonden spelers toegevoegd aan de lijst van gevonden spelers.**
  ```
  for (Player p : priorityQueueToList()) {
      if (firstName.equalsIgnoreCase(p.getFirstName()) && lastName.equals("")) {
          foundPlayers.add(p);
      } else if (firstName.equals("") &&
  lastName.equalsIgnoreCase(p.getLastName())) {
          foundPlayers.add(p);
      } else if (firstName.equalsIgnoreCase(p.getFirstName()) &&
  lastName.equalsIgnoreCase(p.getLastName())){
          foundPlayers.add(p);
      }
  }
  ```
- **Uiteindelijk worden alle spelers die gevonden zijn geretourneerd:**
  ```
  return foundPlayers;
  ```

Efficiëntie & Conclusie

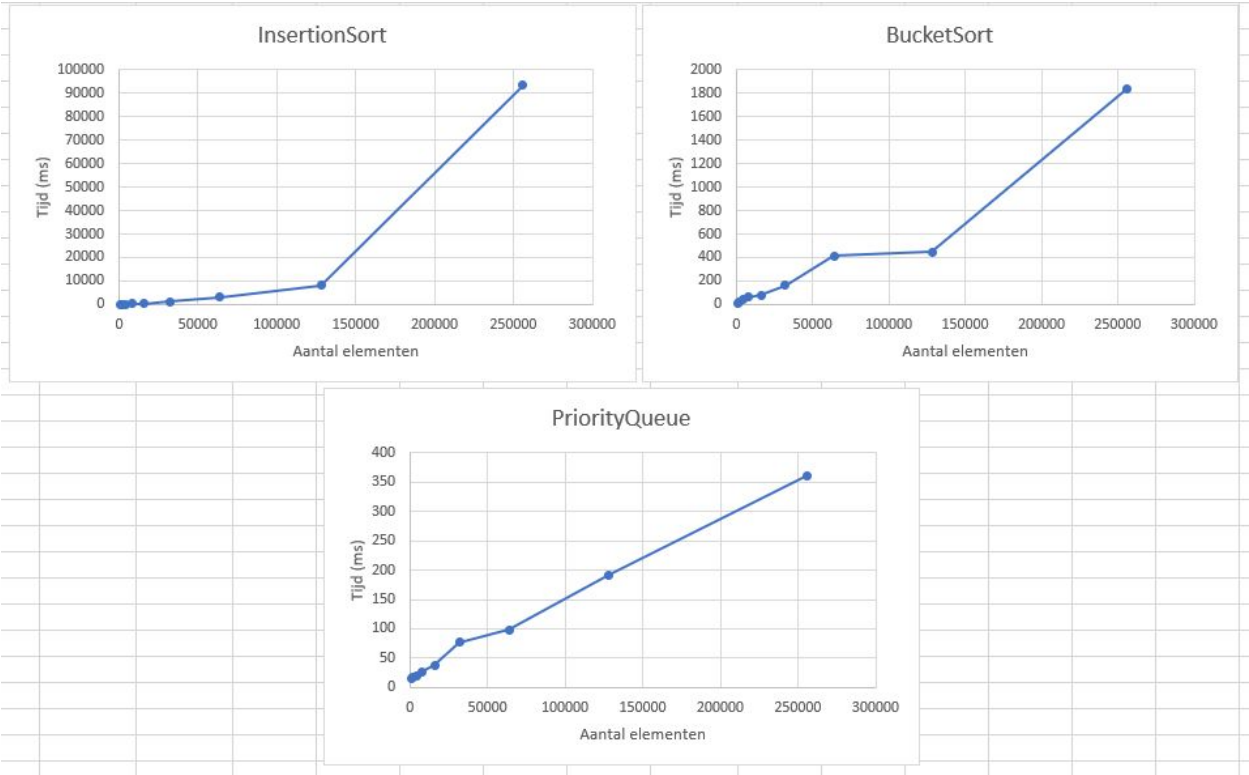
Uit tests blijkt dat het meest efficiënte algoritme de Priority Queue is.

Metingen

Elementen	Gemiddelde tijd	InsertionSort					
1000	14	13	14	13	14	14	
2000	54	50	53	60	56	53	
4000	108	34	160	107	116	122	
8000	177	28	273	189	199	197	
16000	209	214	198	213	204	214	
32000	1014	1042	1049	997	992	989	
64000	3204	5664	2720	2484	2662	2489	
128000	8232	10389	7808	7693	7625	7646	
256000	93348	91853	97131	93579	92048	92128	

Elementen	Gemiddelde tijd	BucketSort					
1000	13	12	13	13	13	13	
2000	24	13	27	26	26	26	
4000	38	25	42	41	40	42	
8000	64	36	65	75	77	68	
16000	80	73	79	92	81	77	
32000	165	160	157	173	162	172	
64000	415	412	433	420	394	416	
128000	444	418	451	453	443	454	
256000	1840	1674	2117	2003	1735	1670	

Elementen	Gemiddelde tijd	PriorityQueue					
1000	16	15	17	14	16	17	
2000	17	15	17	15	16	20	
4000	19	15	17	17	17	30	
8000	27	19	26	29	18	44	
16000	38	35	29	30	32	66	
32000	77	57	54	53	53	167	
64000	99	76	83	80	82	172	
128000	192	174	199	196	202	191	
256000	361	331	341	323	326	485	



Big O

Als je kijkt naar de lijnen van de grafieken kan je waarnemen dat bij Insertion sort de tijdscomplexiteit O(n^2) is, bij BucketSort en Priority Queue O(n).

Als we naar de tabellen kijken kunnen we bij Insertion sort constateren dat bij verdubbeling van aantal elementen tussen 1000 en 16000, de tijd ook verdubbelt. Maar bij alles daarboven neemt de tijd kwadratisch toe. Dus de waarneming klopt bij Insertion Sort.

Bij Bucket Sort is er telkens een constante toename (O(log n)), bij minder dan 16000 elementen. Maar bij meer elementen neemt de tijd met een verdubbeling toe. Dus bij dit algoritme klopt het ook dat de tijdscomplexiteit O(n) is.

Bij Priority Queue is er in (bijna) elk geval een verdubbeling van de tijd die er nodig is om alles te sorteren. Ook hier klopt de O(n) waarneming.

Waarneming

Priority Queue zou je kunnen gebruiken, maar deze zou ik alleen gebruiken wanneer je alleen spelers wilt toevoegen aan een lijst. Dit algoritme sorteert een bestaande lijst niet.

Insertion Sort zou ik echt alleen gebruiken als ik niet al te veel elementen wil sorteren, en een niet te complex algoritme wilt gebruiken. In dit geval zien we dat uit de metingen blijkt dat dit algoritme even snel is als Bucket Sort wanneer we minder dan 1000 elementen sorteren.

Mijn aanbeveling als je veel elementen wilt sorteren, en je zou moeten kiezen tussen deze drie algoritmen, dan zou ik gaan voor Bucket Sort. Wanneer de Insertion Sort klaar is met het sorteren van 4000 elementen, kan in dit geval Bucket Sort meer dan 32000 elementen sorteren in dezelfde tijd.