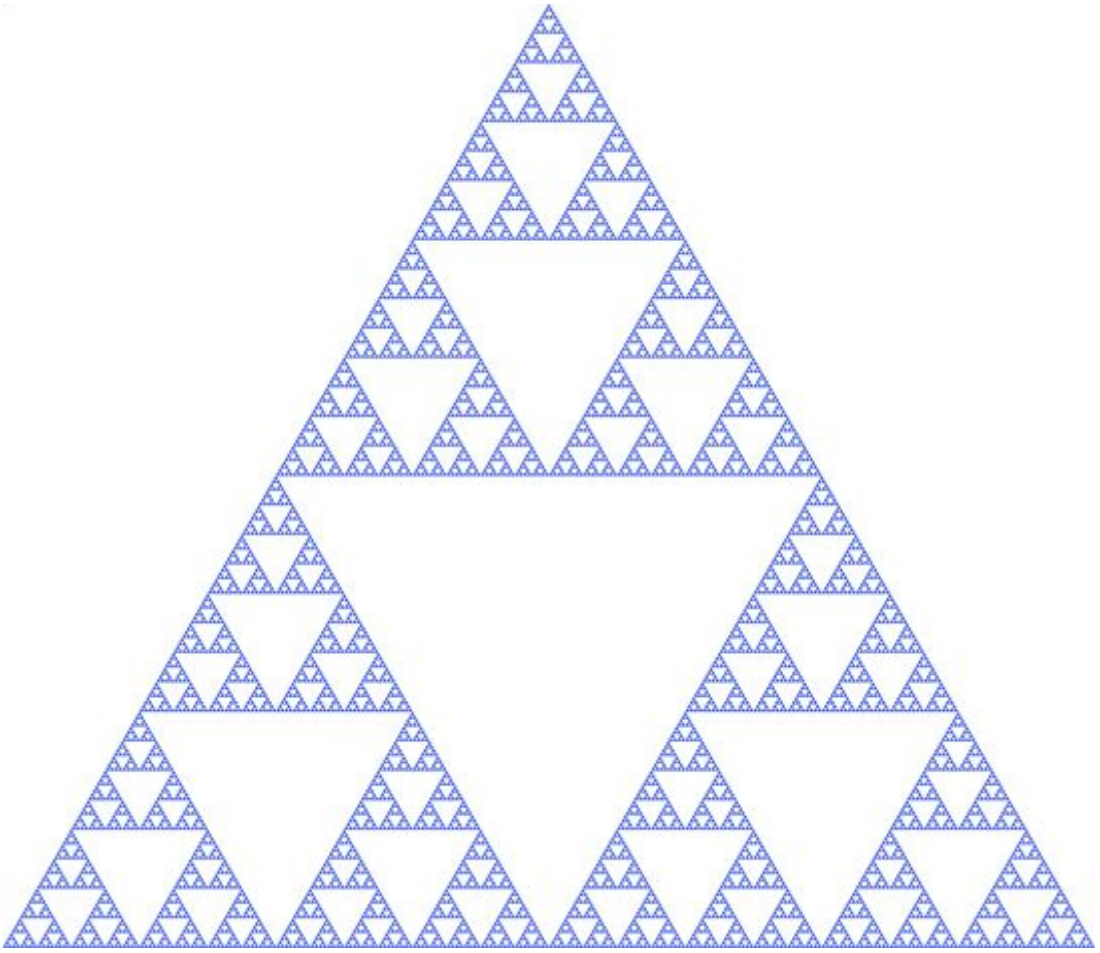
Data Structures Assignment 3 verslag (herkansing)



Naam en studentnummer	: Jeroen Hoff,	500731798
	Costa van Elsas,	500782594
Klas	: IS205	
Datum		
versie 1.0	: 9-11-2018	
versie 2.0	: 7-12-2018	
versie 3.0	: 18-1-2019	

1. Inleiding

In dit verslag zullen wij beschrijven hoe wij LiniarProbing, QuadraticProbing en DoubleHashing toegepast hebben om zo personen (spelers) te kunnen zoeken op een zo efficiënt mogelijke manier. Het doel van dit verslag is om een duidelijk beeld te geven hoe wij dit hebben aangepakt en informatie te geven over de code zelf. In dit verslag zullen een aantal code-snippets toegevoegd worden, deze worden uiteraard uitgelegd.

2. MultiValueSymbolTable

In de applicatie maken wij gebruik van een interface met de methodes put (toevoegen), get (ophalen) en remove (verwijderen). Een interface is een collectie van abstracte methodes. Deze methodes gebruiken wij vervolgens in de drie klassen LiniarProbing, QuadraticProbing en DoubleHashing. Een klassen is een soort blauwdruk voor de applicatie, een applicatie die bestaat uit meerdere klassen die samen het geheel vormen van de applicatie. In de klassen zelf implementeren wij de interface en overschrijven wij de methodes die gedefinieerd staan in de interface.

3. Linear Probing

Hashing is een techniek die gebruikt wordt om een object van een groep dezelfde objecten uniek te identificeren. We gebruiken dit zelf ook in het dagelijks leven. Zo krijgen wij op de hva allemaal een uniek studentnummer. Met dit unieke nummer kan informatie verkregen worden dat specifiek hoort bij dit unieke nummer. In de informatica werken we met objecten. Met behulp van een hash functie kunnen we een object aan een bepaalde index toewijzen. Vervolgens slaan we dit op in een datastructuur genaamd hash table. We lopen alleen tegen een probleem aan als de hashfunctie objecten dezelfde index toewijst. Dit heet een collision. Dit is waar Linear Probing ons gaat helpen. Met behulp van linear probing kunnen we collisions voorkomen. Stel je voor dat de gegenereerde index al bezet is, dan zal linear probing ons helpen een plek te vinden in de hash table die niet bezet is. We maken daarom gebruik van een startwaarde en een interval. De startwaarde is de originele gegenereerde waarde. Stel dat de interval gelijk is aan 1, dan wordt er telkens het getal 1 opgeteld bij de originele index. Dit gaat telkens zo door tot een lege plek gevonden is.
In de klasse LinearProbingMultiSymbolTable hebben we Linear Probing geïmplementeerd. We houden hierin het aantal collisions bij.

3.1. De put methode

```
public void put(String key, Player value) {
```

In de put methode kijken we eerst of een key waarde leeg is. Als dat zo is dan sturen we een bericht waarin staat dat het eerste argument in de methode put null is.

Als de value die opgegeven wordt null is dan gebruiken we de methode delete, waar ik later op terugkom.

Vervolgens kijken we of het aantal collisions dat we bijhouden groter of gelijk is aan de grootte van de tabel gedeeld door twee. Met andere woorden: is de tabel voor 50% vol, ja of nee. Als dat waar is, dan zorgen we ervoor dat de grootte van de tabel twee keer zo groot wordt met behulp van de methode resize.

4. Quadratic Probing

Quadratic probing lijkt erg op linear probing. Zo gebruiken we ook een hashfunctie om een index toe te wijzen aan een object zodat deze in de hashtable geplaatst kan worden. Net als linear probing is quadratic probing een techniek om een nieuwe plek te vinden als er een collision optreedt. De manier hoe die nieuwe plek gevonden wordt is anders dan linear probing. Deze keer gebruiken we een startwaarde en een interval op de andere manier. Als een collision optreedt dan is de nieuwe index de startwaarde (oude index) plus de interval vermenigvuldigd met de interval waarmee de interval steeds 1 ophoogt. Dus bij elke poging (probe) krijgen we een formule als: i = (i + h * h++) waarbij i de originele index is en h de interval.
In de klasse QudraticProbingMultiSymbolTable hebben we Quadratic Probing geïmplementeerd. We houden hierin het aantal collisions bij.

4.1. De put methode

```
public void put(String key, Player value) {
```

In de put methode kijken we eerst of een key waarde leeg is. Als dat zo is dan sturen we een bericht waarin staat dat het eerste argument in de methode put null is.

Als de value die opgegeven wordt null is dan gebruiken we de methode delete, waar ik later op terugkom.

Vervolgens kijken we of het aantal collisions dat we bijhouden groter of gelijk is aan de grootte van de tabel gedeeld door twee. Met andere woorden: is de tabel voor 50% vol, ja of nee. Als dat waar is, dan zorgen we ervoor dat de grootte van de tabel twee keer zo groot wordt met behulp van de methode resize.

5. Double Hashing

Double Hashing is een techniek dat wordt net als de andere gebruikt om in hash tables, hash collisions op te lossen. Hash tables (hashmap) zijn datastructuren die kunnen vastleggen wat de sleutel (key) en de waarde (value) is in een lijst met gegevens. Hash collisions zijn wanneer de berekening van de Hash hetzelfde is als een andere waardoor ze allebei op dezelfde index zouden komen te staan. Dit mag niet gebeuren.

5.1. De put methode

In de put methode van deze klasse kijken we eerst of het aantal collisions groter is dan de grootte van de lijst waar de spelers in zitten, zo ja dan stopt de methode. Er zijn twee hashes aangemaakt die we kunnen gebruiken om de sleutel te hashen. Zolang de waarde met de hash niet gelijk is aan nul wordt de eerste hash de tweede hash en gaat de eerste hash % lijst grootte. Vervolgens wordt de waarde en de sleutel aangegeven en in de lijst gezet. `values.set(firstHash, value);` Met deze code waarin firstHash de sleutel is. Worden de sleutel en waarde gezet in de lijst. Values is in dit geval de lijst met spelers zoals hier staat gedefinieerd: `private List<Player> values;`

5.2. De get methode

In de Get methode van deze klasse maken we eerst een lijst van een lijst genaamd getPlayers, dit is een lijst van alle spelers waar we nu in deze methode doorheen kunnen lopen en we later ook terug geven. We definiëren de hashes. Zoals de naam al zegt “Double” hashing hebben we twee verschillende hashes nodig. Wij hebben de hashes op onze eigen manier geïmplementeerd. Hash 1 wordt berekend over de grootte van de lijst. Hash 2 wordt berekend over de grootte van de lijst % primeSize. PrimeSize is een methode die een priemgetal berekend die kleiner is dan de grootte van de lijst.
Vervolgens lopen we door de lijst heen zo lang de index niet 0 is en kleiner dan de grootte van de lijst gaan we er doorheen. We kijken door de lijst heen en als de voornaam alleen, achternaam alleen of de voornaam + achternaam klopt. Dan voegen we de spelers toe aan de lijst getPlayers en die lijst geven we vervolgens terug aan de methode zodat hij op een andere plek opnieuw opgevraagd kan worden.

5.3. De remove methode

In de Remove methode doen we eigenlijk het hetzelfde als in de get methode. We stoppen namelijk alle spelers in een lijst getPlayers. Het enige wat nu veranderd is dat we de spelers die verwijderd moeten worden uit de lijst worden gehaald. Wanneer alle juiste waarde in de lijst zitten wordt deze lijst teruggegeven.

6. Collision detection

Elke keer wanneer er een succesvolle Put wordt gedaan in alle manieren (linearprobing, quadraticprobing, doublehashing) wordt er 1 collision bij opgeteld. Zie volgende afbeelding:

```
@Override
public void put(String key, Player value) {

    int firstHash = hash1(key);
    int secondHash = hash2(key);

    while(values[firstHash] != null) {
        firstHash += secondHash;
        firstHash %= tableSize;
        collisions++;
    }

    values[firstHash] = value;
```

Dit is de put van DoubleHashing. Elke keer als er een speler wordt toegevoegd krijgen we een collision er bij. collisions is een variabele die uit de volgende methode komt:

```
public int getCollisions() {
    return collisions;
}
```

Deze methode staat ook in elk van de manieren en geeft het aantal collisions terug. We hebben aan de interface HighScoresList een String collisions(); toegevoegd. Hierdoor kunnen we gemakkelijk in de tests kijken hoeveel collisions er zijn.

```
System.out.println(highscores.collisions());
```

```
@Override
public String collisions() {
    String firstNameCollisions = "First name collisions: " + firstNameFinder.getCollisions();
    String lastNameCollisions = "Last name collisions: " + lastNameFinder.getCollisions();
    String fullNameCollisions = "Full name collisions: " + fullNameFinder.getCollisions();

    return firstNameCollisions +  " " + lastNameCollisions +  " " +  fullNameCollisions;
}
```

Door dit stuk code zien we het aantal collisionsen wordt het als volgt vertoont bij het runnen van de test collisionsShouldHappen().

```
First name collisions: 445238 Last name collisions: 670204 Full name collisions: 44195
```

Zo hebben we later in dit verslag de aantallen in een tabel en grafiek kunnen zetten. Bij een lijst grootte van 10501, 11701, 13309, 15401. De array grootte wordt aangepast in de test collisionsShouldHappen().

7. Extra Tests

Voor de extra tests hebben wij tests geschreven voor het toevoegen met de Put methode `addPlayer()` en verwijderen met de Remove methode `removePotter()`.

In de `removePotter()` test zoeken we eerst met een bestaande methode FindPlayer() (methode om een speler te kunnen zoeken in een lijst met spelers op voornaam + achternaam) de speler met als achternaam “Potter” `List<Player> potters = highscores.findPlayer(null, "Potter");` in dit stukje code maken we eerst een eigen lijst met de personen die “Potter” in hun achternaam hebben. Nu is er dus een lijst genaamd potters met alle personen er in die “Potter” in hun achternaam hebben. Binnen deze lijst vertellen wij tegen de code dat het de potter met positie 1 in de lijst moet verwijderen `potters.remove(potters.get(1));`
In de eerste instantie weten wij dat er 3 potters zijn omdat die in de tests van de docent zijn toegevoegd. Wanneer wij deze “Potter” dus verwijderen zal dat moeten betekenen dat het aantal personen in de lijst niet 3 is maar 2. Dit wordt gecontroleerd met de volgende methode `assertEquals(2, potters.size());`
Hier wordt er dus gecontroleerd met assertEquals (controle of de meegegeven conditie gelijk is aan wat er verwacht wordt) of de lengte van potters 2 is. Zo ja dan wordt de test positief afgerond.

In de `addPlayer()` test maken we eerst zelf een speler aan genaamd testPlayer. Deze testPlayer geven we vervolgens de voornaam “player” en de achternaam “testing” met nog een highscore van 1000. Deze speler voegen wij toe aan de lijst met spelers die in de test van de docent wordt gedefinieerd namelijk highscores. Wanneer de speler wordt toegevoegd wordt dat met de Put methode gedaan zo kunnen we deze gemakkelijk testen. `highscores.add(testPlayer);`

Vervolgens doen we hetzelfde als hierboven bij de removePotter() test. We maken een eigen lijst met daarin de spelers die “player” als voornaam hebben en “testing” als achternaam. Daarna kijken we weer met de assertEquals methode of de lengte van de lijst 1 is omdat er maar 1 persoon die zo heet toegevoegd wordt. Ook hebben we nog een extra assertEquals met om te kijken of op plek 1 in de lijst ook echt daadwerkelijk de genoemde speler staat. `assertEquals(testPlayer, players.get(0));`

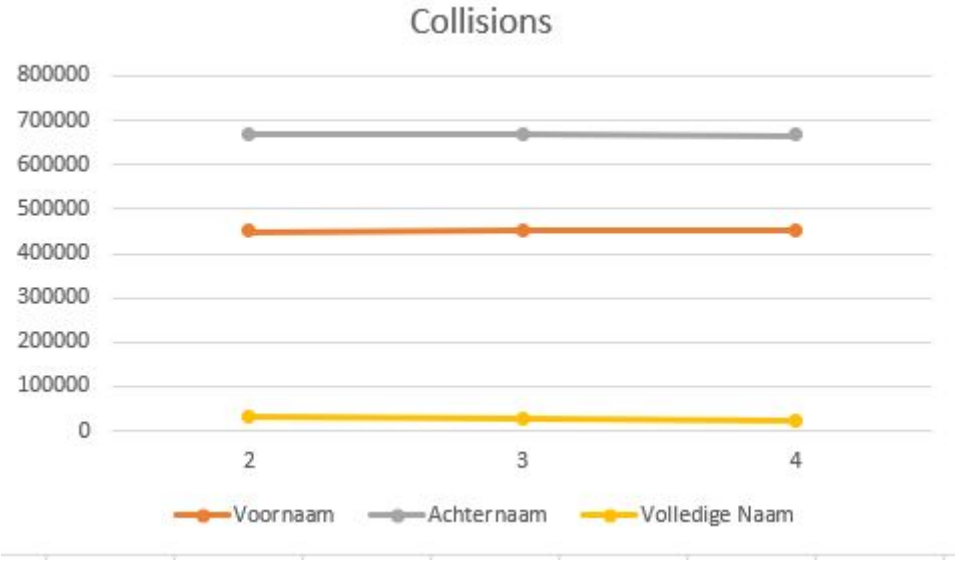
8. Results

8.1. Tabel

In de volgende tabel wordt het aantal collisions bijgehouden waarbij de voornaam collisions met LinearProbing gemeten wordt, de achternaam met QuadraticProbing en de volledige naam met DoubleHashing.

Grootte lijst	*Voornaam*	*Achternaam*	*Volledige Naam*
10501	441910	670034	47659
11701	451597	669068	32198
13309	453437	670948	25162
15401	452582	668516	21346

8.2. Grafiek



9. Vergelijkingen van de strategieën

De strategie waarvan wij denken dat het de beste strategie is is de double hashing. Hoe groter de array size hoe minder collisions. De manier waarop linear en quadratic probing geïmplementeerd zijn. Laten zien dat het bij onze implementatie niet de snelste manieren zijn dit wordt ook aangetoont in de grafiek. Wel zijn we van mening dat bij een klein aantal spelers je waarschijnlijk beter kunt gaan voor de linear probing omdat die met kleine hoeveelheden snel calculeert.

10. Extra opdrachten herkansing practicum 3

De vergelijking die we gedaan hebben in de originele opdracht was niet helemaal een eerlijke vergelijking, omdat er meer voornamen beschikbaar waren dan achternamen. Voor de extra opdracht bekijken we het effect van het gebruiken van de combinatie van de voor en achternaam bij alle collision strategieën. Dit hebben we gedaan door de add methode aan te passen in de klasse HighScorePlayerFinder.java.

De originele add methode zag er als volgt uit:

```
@Override
public void add(Player player) {
  firstNameFinder.put(player.getFirstName() , player);
  lastNameFinder.put(player.getLastName(), player);
  fullNameFinder.put(player.getFirstName() + player.getLastName(), player);
}
```

De vernieuwde versie ziet er zo uit:

```
@Override
public void add(Player player) {
  firstNameFinder.put(player.getFirstName() + player.getLastName() , player);
  lastNameFinder.put(player.getFirstName() + player.getLastName(), player);
  fullNameFinder.put(player.getFirstName() + player.getLastName(), player);
}
```

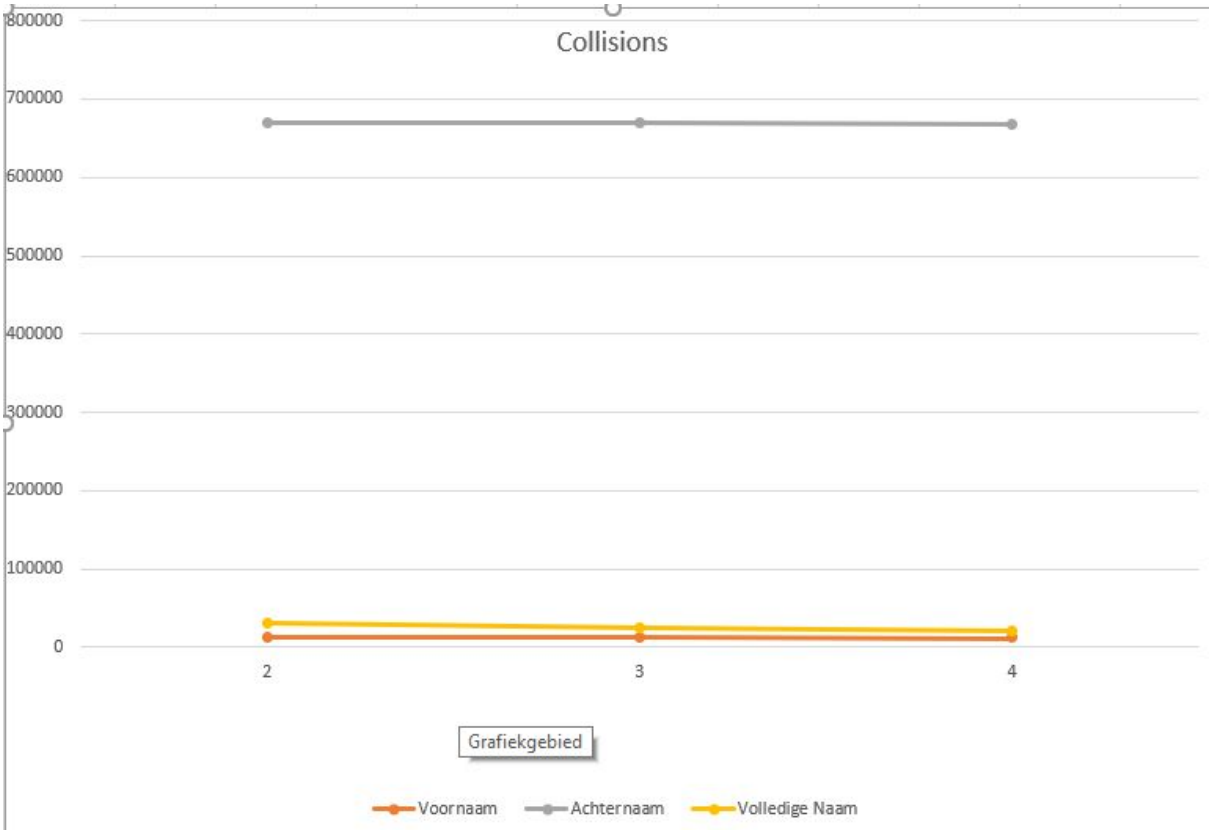
In de nieuwe versie wordt er dus bij de firstnamefinder, lastnamefinder en fullnamefinder allemaal op de volledige naam gezocht. Bij firstnamefinder wordt er gebruik gemaakt van linear probing, bij lastnamefinder wordt er gebruik gemaakt van quadratic probing en bij fullnamefinder wordt er gebruikt gemaakt van double hashing.

De resultaten van de nieuwe add methode zijn in de volgende tabel en grafiek weergegeven.

10.1. Tabel

Grootte lijst	*Voornaam*	*Achternaam*	*Volledige Naam*
10501	13769	668752	44427
11701	13685	669521	31932
13309	13921	668798	25262
15401	13571	668583	21930

10.2. Grafiek (aantal collisions voornaam + achternaam)



11. Conclusie

Als getoont in de grafiek 10.2 is er een grote kans achternaam die geïmplementeerd is met quadratic probing het aantal collisions niet goed wordt gemeten. Verder zien we bij de voornaam die met linear probing geïmplementeerd is dat het aantal collisions aardig gelijk blijft en met double hashing dat de waarde naarmate de array size groter wordt steeds een stukje daalt. Wanneer er dus een grote array is raden wij aan om voor double hashing te kiezen, in geval van een kleine array raden wij aan linear probing te kiezen. Deze keuzes zijn gebaseerd op de door ons gemeten statistieken.