Datastructures

Opdracht 3 – Finding players

# 

|  |  |
| --- | --- |
| Naam (studentnummer) | Daan Befort (500727616) |
| Naam (studentnummer) | Kevin Huijzendveld (500727925) |
| Datum | 24/10/18 |

Inhoudsopgave

[Inleiding 3](#_Toc528266299)

[Double hashing implementeren 4](#_Toc528266300)

[Hash functies 4](#_Toc528266301)

[Put functie 4](#_Toc528266302)

[Get functie 4](#_Toc528266303)

[Grafiek 5](#_Toc528266304)

# Inleiding

In dit document leggen we uit hoe we de methodes linear probing, quadratic probing en double hashing hebben geïmplementeerd in het project. Per methode leggen we de functies uit en geven we een grafiek met informatie of de methode. Aan het einde geven we een conclusie over welke methode het beste te gebruiken is.

# Double hashing implementeren

In dit project hebben we voor het zoeken naar voornaam + achternaam “double hashing” geïmplementeerd. Door double hashing te gebruiken zal het aantal collisions verminderen omdat we twee hash functies gebruiken. Wanneer er een collision is (dit betekent wanneer het toevoegen van een player niet kan op de uitgerekende positie) gaan we met de tweede hash methode berekenen hoeveel plekken verder we moeten gaan kijken. Dit gaat door totdat er geen collision meer is.

## Hash functies

Hieronder zie je de twee hash manieren die we hebben gebruikt. De eerste hash manier is om eerste plek te berekenen. En de tweede hash manier is om het aantal te bereken hoeveel stappen we verder moeten gaan kijken. We geven de key mee (dit is voornaam + achternaam) en krijgen halen daar de hashcode van op en doen dat modulo de arraysize in de eerste methode. In de tweede methode gebruiken we prime getal **5** om de hashcode te berekenen.

**private int** hashFunc1(String key) {  
 **return** (key.hashCode() & 0x7fffffff) % **ARRAY\_SIZE**;  
}  
  
**private int** hashFunc2(String key)  
{  
 **return** (**PRIME** - (key.hashCode() % **PRIME**));  
}

## Put functie

In de put functie voegen we de players toe aan onze player lijst. We een plek in de lijst die nog leeg is. Zolang we geen lege plek zijn tegen gekomen gaan we door met stappen de we hebben gekregen uit de tweede hash functie. Tot dat we wel een lege plek vinden en stoppen hier de player in. Per keer de plek bezet is slaan we de collision op.

**while** (**playerList**[hashVal] != **null**) {  
 **collisions**++;  
 hashVal += stepSize;  
 hashVal %= **ARRAY\_SIZE**;  
}

## Get functie

In de get functie halen we een specifieke speler op die gezocht wordt. We gaan door de lijst heen aan de hand van de hashvalues die we krijgen uit de key. Per gevonden player gaan we kijken of de voornaam + achternaam gelijk is aan degene die gezocht wordt. Zo weten we dat de player die we hebben de juiste is. En sturen deze player terug. Wanneer we helemaal niks vinden sturen we een lege lijst terug.

**while** (**playerList**[hashVal] != **null**) {  
 Player currentPlayer = **playerList**[hashVal];  
 String fullname = currentPlayer.getFirstName() + currentPlayer.getLastName();  
 **if** (fullname.equals(key)) {  
 players.add(**playerList**[hashVal]);  
 }  
 hashVal += stepSize;  
 hashVal %= **ARRAY\_SIZE**;  
}

## Grafiek

Hieronder de grafiek voor double hashing. Per grootte van de lijst kijken we hoeveel collisions er zijn.

# Conclusie

|  |  |  |  |
| --- | --- | --- | --- |
| **Array size** | **Linear probing** | **Quadratic probing** | **Double hashing** |
| 10501 | 0 | 0 | 63395 |
| 11701 | 0 | 0 | 29114 |
| 13309 | 0 | 0 | 23498 |
| 15401 | 0 | 0 | 14523 |