Mark Altorf, Quinnard Stewart

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

Rapport Opdracht 3

Data Structures

Inhoudsopgave

[Inleiding 2](#_Toc528359051)

[2. Linear Probing 3](#_Toc528359052)

[Constructor 3](#_Toc528359053)

[Hash 3](#_Toc528359054)

[Put (toevoeging) 4](#_Toc528359055)

[Get 4](#_Toc528359056)

[Botsingen(Collissions) 4](#_Toc528359057)

[3. Quadratic probing 6](#_Toc528359058)

[Contructor 6](#_Toc528359059)

[Hashing 6](#_Toc528359060)

[7](#_Toc528359061)

[Put (toevoeging) 7](#_Toc528359062)

[Check maximale grootte 7](#_Toc528359063)

[Check naar botsingen 7](#_Toc528359064)

[Add 8](#_Toc528359065)

[Get 9](#_Toc528359066)

[Collision check 9](#_Toc528359067)

[Resize 10](#_Toc528359068)

[Empty 10](#_Toc528359069)

[4. Double hashing 12](#_Toc528359070)

[5. Resultaten 13](#_Toc528359071)

[6. Conclusie 14](#_Toc528359072)

[7. Bronvermelding 15](#_Toc528359073)

# Inleiding

In vervolging van de vorige opdracht van onze zus heeft ze alweer onze hulp nodig. Dit keer wil ze zoekopdrachten implementeren op basis van de voor- en/of achternaam van de spelers. Hiervoor zou een ‘*symbol table*’ perfect zijn. Zo een tabel maakt gebruik van een sleutel en een waarde. Waarde wordt opgeslagen op de plek dat afhankelijk is van de sleutel, waardoor bij de zoekopdracht alleen de sleutel moet gevraagd worden om de waarden terug te kunnen vinden. Maar dit heeft wel een probleem voor toevoeging van meerder keer van een specifieke sleutel. In dit geval is er sprake van een botsing, waarvoor er verschillende methodieken zou kunnen worden aangepakt.

Bij dit opdracht werd er naar drie verschillende manieren gekeken. De eerste methode is de ‘*lineair probing*’ methode. (**1**)

Deze gaat zoeken naar de spelers met dezelfde voornaam.

De tweede methode is de ‘*quadratic probing*’ methode en zal naar spelers zoeken met dezelfde achternaam.(**2**)

En de ‘*double hashing*’ methode zal naar spelers zoeken met zowel dezelfde voor- en achternaam. (**3**)

Vervolgens zullen alle resultaten van de testen worden aangegeven. (**4**) Hiervoor wordt bij elke test *runtime* en aantal botsingen aangegeven.

En er wordt een conclusie getrokken van de test resultaten. (**5**)

Alle bronnen die waren gebruikt bij het ontwikkelen van dit opdracht bevinden zich in de bronvermelding. (**7**)

# Linear Probing

Linear probing is een methode om een bij botsing van sleutels, de nieuwe sleutel te kunnen verplaatsen naar een nieuwe locatie in de tabel. Deze doet het aan de hand van een formule:

* Index: positie in de tabel;
* Sleutel hashcodewaarde: numerieke waarde van alfanumerieke sleutel;
* q: toenemende waarde.

Bij een botsing moet er genoteerd worden dat er een botsing werd veroorzaakt. Ten eerst, bij elke klasse moeten de algemene variabelen vastgesteld worden. (zie Codesnippet 1)

private int M;  
private String[] keys;  
private Player[] vals;  
  
private int collisions;

Codesnippet 1 Linear Probing Algemene variabelen

Een constructor van een klasse is een belangrijke methode voor het initialiseren van een klasse. Deze methode staat parameters toe, in dit geval de maximale grootte van het tabel. (zie Codesnippet 2 )

Hiermee wordt de maximale grootte, ‘M’, de waarde van de toegevoegde parameter. Arrays kunnen niet geinitialiseerd worden zonder een maximale grootte. De maximale grootte van elke array wordt dan de waarde van ‘M’. Voor het tellen van de botsingen, wordt een waarde van nul aan het variabel ‘*collisions*’ toegepast.

Codesnippet 2 Linear Probing constructor

public LinearProbingMultiValueSymbolTable(int arraySize) {  
 M = arraySize;  
 keys = new String[M];  
 vals = new Player[M];  
 collisions = 0;  
}

## Hash

Hash is een methode om the numerieke hash waarde van de gegeven sleutel terug te geven. Deze doet het door middel van de ingebouwde ‘hashCode’ methode, 32-bit stelling van Java. Wanneer deze gereed is, wordt de modulo hiervan met betrekking tot de maximale grootte de index voor de waardes. (zie Codesnippet 3)

Codesnippet 3 Linear Probing hash

private int hash(String key) {  
 return (key.hashCode() & 0x7fffffff) % M;  
}

## Put (toevoeging)

De ‘*put*’ methode is om een sleutel en waarde aan te tabel te kunnen toevoegen. Hiervoor wordt de index met ‘*i*’ gedefinieerd. In een *for-loop* wordt bij elke *hash* waarde van de sleutel gekeken of de waarde bij de desbetreffende index geen waarde bevat. Indien juist, wordt het toegevoegd aan de array. Anders wordt er, volgens de formule, naar de eerstvolgende index gekeken. Indien er een lege plek gevonden is wordt de waarden daar toegevoegd en is de loop afgebroken. (zie Codesnippet 4)

@Override  
public void put(String key, Player value) {  
 int i;  
  
 for (i = hash(key); keys[i] != null; i = (i + 1) % M) {  
 collisions++;  
 if (keys[i] == null) {  
 keys[i] = key;  
 vals[i] = value;  
 return;  
 }  
 }  
 keys[i] = key;  
 vals[i] = value;  
}

Codesnippet 4 Linear Probing put methode

## Get

De klasse HighscorePlayerFinder roept de ‘*get*’ methode van deze klasse aan om een lijst van de spelers met de aangegeven sleutel te ontvangen. Om dit waar te kunnen maken moet eerst een lijst met de juiste type geïnitialiseerd worden. Na dit kan gelijk het aantal botsingen afgebeeld worden. Het belangrijkste gedeelte van deze methode is de *for-loop*, dat gaat door alle sleutels in de array. Als de waarde van de opgegeven sleutel gelijk is aan dat van in de array zit wordt de speler met dezelfde index in de lijst toegevoegd. Deze lijst wordt teruggestuurd naar waar het opgeroepen werd. (zie Codesnippet 5)

Codesnippet 5 Linear Probing get methode

@Override  
public List<Player> get(String key) {  
 List<Player> playerList = new ArrayList<>();  
 System.*out*.println("Collissions count: "+getCollisions());  
  
 for (int i = 0; i < M; i++) {  
 if (key.equals(keys[i])) {  
 playerList.add(vals[i]);  
 }  
 }  
 return playerList;  
}

## Botsingen(Collissions)

In het begin was het aantal botsingen een waarde van 0. Sindsdien bij elke botsing werd deze toegenomen. Om de aantal botsingen terug te kunnen vinden is er een get methode aangemaakt, ‘*getCollisions*’. Deze geeft aan de huidige waarde van de variabel ‘*collisions*’.

( zie Codesnippet 6)

Codesnippet 6 Linear Probing getCollissions

public int getCollisions() {  
 return collisions;  
}

# Quadratic probing

Quadratic probing is nog een methode om tegen een botsing te oplossen. Integendeel van het toevoegen bij een botsing op een lineaire wijze (zie vorige hoofdstuk), wordt deze met een kwadraat toegenomen. De formule voor de toename bij botsingen van zo een toename is:

* Sleutel: hashcode waarde van het gegeven sleutel waarde in cijfers (integer);
* q: Toenemende factor per botsing tijdens toevoeging.

Eerst, werd alle benodigde algemene variabelen gedefinieerd (zie Codesnippet 7). N is de variabel om het aantal toegevoegde waarden in de gaten te kunnen houden.

Omdat er met twee verschillende typen waarden moet werken, werd er voor elke type een array gedefinieerd. De sleutels hebben een alfanumeriek type, van ‘*String*’, en de waarden hebben een object type, ‘*Player*’. Verder moet ook de maximale grootte van de tabel ook gedefinieerd worden, met name de ‘*size*’ variabel.

Codesnippet 7 Algemene variabelen

private int N= 0;  
private Player[] players;  
private String[] keys;  
  
private int size; // initial size, collission variable

Elke klasse moet een ‘*constructor*’ hebben om de waarden van de algemene variabelen te kunnen vaststellen op basis van een parameter. In dit geval neemt het een parameter de initial maximale grootte van de tabel, ‘*arraySize*’. Met deze parameter wordt de maximale grootte van de tabel momenteel vastgesteld. Een array kan op twee manieren geïnitialiseerd worden: met new Type[max. grootte] of door een al gevulde array in de nieuwe array toevoegen. Bij deze klasse worden de maximale grootte voor elke array aangegeven bij de initialisatie. Voor het tellen van de botsingen, wordt een waarde van nul aan het variabel ‘*collisions*’ toegepast. (zie Codesnippet 8)

public QuadraticProbingMultiValueSymbolTable(int arraySize) {  
 size = arraySize;  
 players = new Player[size];  
 keys = new String[size];  
  
}

Codesnippet 8 Constructor

## Hashing

Bij de ‘*hash*’-methode wordt de sleutel omgezet naar een integer. Een van de belangrijkste notaties voor deze is om de ingebouwde ‘*hashcode()*’ functie te gebruiken om de waarde in een integer te laten omzetten. Hierna moet de waarde ook als een 32-bit waarde worden omgezet met de toevoeging van ‘*& 0x7fffffff*’. En volgens de formule met de waarde van ‘*i*’ opgeteld. In dit methode wordt de waarde van ‘*i*’ in eerste instantie nul, en als ‘*q*’ niet gelijk is aan nul, wordt ‘i’ aangepast voor de kwadratisch waarde (q\*q) van ‘*q*’ (zie Codesnippet 9).

## 

Codesnippet 9 Hash methode

private int hash(String k,int q){  
 int i=0;  
 if (q!=0){  
 i = (int) Math.*pow*(q,2);  
 }  
 return Math.*abs*(((k.hashCode()& 0x7fffffff) +(i)));  
}

## **Put (toevoeging)**

‘*Put*’ is de methode dat wordt opgeroepen door de klasse *HighscorePlayerFinder*. Deze klasse stuurt de parameters van de sleutel en de waarde. In eerste opzicht wordt een nieuwe variabel gedefinieerd, met name ‘*q*’, met een waarde van nul. Want als deze nul is wordt volgens de ‘*hash*’-functie alleen de *32-bit hashCode* waarde teruggegeven. Ten eerst wordt ernaar gekeken of de table leeg is. Indien wel, zijn de meegegeven sleutel en waarde de eerste waarden die toegevoegd moeten worden, maar hiervoor moet de sleutel wel gehashed worden en de modulo ten opzichte van de maximale grootte berekent worden. De toevoeging hiervan wordt in de ‘*add*’-methode toegepast. Indien de tabel niet leeg is wordt de ‘*AddToFilledArray*’-methode opgeroepen met de index (hashed sleutel), sleutel, waarde en toenemende factor als parameters meegestuurd. (zie Codesnippet 10)

Codesnippet 10 Put methode

public void put(String key, Player value) {  
  
 int q = 0;  
 if (isEmpty()){  
  
 add(hash(key,q)%size,key,value);  
 }else{  
 AddToFilledArray(hash(key,q)%size, key, value,q);  
 }  
}

De ‘*AddToFilledArray*’- methode bestaat uit twee delen:

1. check maximale grootte;
2. check naar botsingen.

### Check maximale grootte

Bij het checken van de maximale grootte wordt er naar de meegegeven index gekeken of het groter of gelijk is aan de helft van de maximale grootte. Bij het geval dat de stelling juist is wordt de ‘*resize*’-functie (**zie Resize**) opgeroepen met dubbel de maximale grootte als een parameter meegegeven. De reden hiervoor is om te voorkomen data als er een botsing ontstaat dat er automatisch meer ruimte is voor een nieuwe index.

### Check naar botsingen

Om te voorkomen dat een sleutel en waarde kwijt wordt geraakt wordt er gechecked of er een botsing ontstaat. Deze wordt gechecked met behulp van de ‘*checkCollission*’-methode (**zie Collisions check**). Indien er een botsing ontstaat wordt waarde van ‘*q*’ met 1 toegenomen, waardoor de voorbenoemde formule wordt toegepast en als waarde van ‘*t*’ aangegeven. En wordt de methode weer opgeroepen met een nieuwe index waarde. Indien er geen botsing ontstaat, worden de waarden via de ‘*add*’-methode toegevoegd aan de arrays. (zie Codesnippet 11)

private void AddToFilledArray(int index, String key, Player value, int q){  
 int t;  
  
 if (index>=(size/2)) {resize(size\*2);} //resizes table incase index is larger than half the size.  
  
 if (checkCollission(index, key)){  
 q++;  
  
 t = hash(key,q);  
 AddToFilledArray(t%size, key,value, q);  
 }else{  
 add(index,key,value);  
 }  
}

Codesnippet 11 AddToFilledAray methode

## Add

De ‘add’- methode heeft twee verschillende implementaties;

1. Met de basis parameters: index, sleutel, waarde;
2. Met de basis parameters en toenemende factor.

Dit is om te voorkomen dat er waarden ontbreken in de tabel. Het kijkt eerst of op de plek voor toevoeging leeg is. Indien wel, dan voegt het toe aan de arrays. Anders gaat het, met behulp van de ‘hash’-methode, naar de volgende waarde kijken via de tweede implementatie. (zie Codesnippet 12)

Codesnippet 12 Add met index, sleutel en waarde

private void add (int index, String key, Player value){  
 if(keys[index]==null){  
  
 keys[index] = key;  
 players[index] = value;  
 N++;  
  
 }else {  
 index = hash(key, 1);  
 add(index, key, value, 1);  
 }  
}

Bij de tweede implementatie gaat het bij de volgende index kijken of het leeg is. Indien niet roept het de methode weer op met een toename van q+1. Dit doet het totdat het een lege plek vindt in de array om de waarde toe te kunnen voegen. (zie Codesnippet 13)

private void add (int index, String key, Player value, int q){  
 if ( q>0){  
  
 if(keys[index]==null){  
  
 keys[index] = key;  
 players[index] = value;  
 N++;  
 }else {  
 index = hash(key, q++);  
 add(index, key, value, q);  
 }  
 }  
}

Codesnippet 13 Add methode met index, sleutel, waarde en toeneem factor

## Get

De ‘get’-methode is wat de HighScorePlayerFinder klasse oproept als er een lijst wil met spelers met alleen dezelfde achternaam wil hebben. Bij dit wordt de achternaam als parameter aangegeven en wordt ernaar als de sleutel gerefereerd. In eerste instantie moet er een lege lijst gedefinieerd worden. Hierna gaat door de array met de sleutels kijken, eerst als er een sleutel is, dan als de sleutel hetzelfde is met de meegegeven sleutel. Indien wel, wordt de speler aan de lijst toegevoegd, anders gaat het naar de eerstvolgende index kijken.

Aan het einde wordt deze lijst teruggestuurd naar de methode waarvan het opgeroepen was. (zie Codesnippet 14)

@Override  
public List<Player> get(String key) {  
 List<Player> l = new ArrayList<>();  
  
 for (int i = 0; i<keys.length ; i++){  
 if (keys[i]!=null){  
 if (keys[i].equals(key)){  
 l.add(players[i]);  
 }  
 }  
 }  
  
 return l;  
}

Codesnippet 14 Get methode voor aangevraagde sleutel

## Collision check

De ‘checkCollision’-methode is de methode dat constant gaat checken of de waarde van het gegeven index nul is. Indien het zo is, geeft het juist aan, anders onjuist. (zie Codesnippet 15)

private boolean checkCollission(int index, String key) {  
 if(keys[index]!=null){  
 collission+=1;  
 System.*out*.println("Collission for @oldkey: "+keys[index]+ " , @newkey: "+ key+ "@index: "+ index);  
 return true;  
 }else {  
 return false;  
 }  
}

Codesnippet 15 Collision check methode

## Resize

Om de maximale grootte te kunnen aanpassen wordt er gebruik gemaakt van de ‘*resize*’-methode. Deze neemt de nieuwe grootte als parameter. Eerst wordt een nieuwe instantie van de huidige klasse opgeroepen met als maximale grootte de nieuwe parameter. Hierna wordt door elke waarden in de sleutel array gekeken of er eerst, het niet leeg is, dan voegt hij de oude waarden met toe aan de nieuwe instantie van de klasse. En de grootte, sleutel- en spelers- array worden aangepast met de nieuwe maximale grootte. (zie Codesnippet 16)

Codesnippet 16 Resize methode

private void resize(int size){  
 QuadraticProbingMultiValueSymbolTable q = new QuadraticProbingMultiValueSymbolTable(size);  
  
 for (int i = 0; i<keys.length; i++){  
 if(keys[i] !=null){  
 q.put(keys[i],players[i]);  
 }  
 }  
  
 this.keys=q.keys;  
 this.players = q.players;  
 this.size = q.size;  
 this.collission = q.collission;  
}

­

## Empty

De ‘empty’ methode is het methode dat controleert of de tabel al waarden bevat. Waarvoor het vraagt naar de ‘*size*’ methode indien deze nul is geeft het aan dat de voorwaarde juist is. De ‘*size*’ methode geeft de waarde van de variabel ‘*N*’ aan (zie Codesnippet 17).

*/\*\*  
 \* Checks if Symbol table is empty.  
 \** ***@return*** *true if Symbol table is empty, false if not empty.  
 \*/*public boolean isEmpty(){  
 return size()==0;  
}  
  
*/\*\*  
 \* Returns amount of entries added to the table.  
 \** ***@return*** *N in amount of entries added to the table.  
 \*/*public int size(){  
 return N;  
}

Codesnippet 17 Empty & size methoden

# Double hashing

[inleading]

In eerste opzicht moeten alle algemene variabelen gedefinieerd worden. Deze zijn ongeveer hetzelfde als bij de voorafgaande methodieken. Waarvoor ‘M’ de maximale grootte, ‘keys’ een numerieke array, ‘vals’ een array van objecten is en ‘collisions’ het aantal botsingen wordt aangeeft. (zie Codesnippet 18)

Codesnippet 18 Double Hashing algemene variabelen

private int M; // number of key value pairs in the symbol table  
private String[] keys;  
private Player[] vals;  
private final int size; // size of linear probing table  
private int collisions;

Waarvoor de constructor ook hetzelfde inhoud als de voorafgaande intialisatie van deze variabelen. De waarde van ‘*M*’ is dat van de meegegeven parameter ‘*arraySize*’. Deze geeft aan de maximale grootte van beide te gebruikte arrays. En de variabel voor aantal botsingen, ‘*collisions*’, wordt met een waarde van nul geïnitialiseerd. (zie Codesnippet 19)

Codesnippet 19 Double Hashing contructor

public DoubleHashingMultiValueSymbolTable(int arraySize){  
 size = arraySize;  
 keys = new String[arraySize];  
 vals = new Player[arraySize];  
 collisions= 0;  
}

## Hash

De hash methode zet de sleutel waard om in en numerieke waarde. In tegendeel van die van de lineaire en kwadratisch methodieken moet er meer rekening houden met het aantal letters in de sleutel. Hierdoor kan de waarde een grotere waarde teruggeven dan de voorafgaande waarden. (zie Codesnippet 20)

Codesnippet 20 Double Hashing hash

Input code

## Put (toevoeging)

Bij de ‘put’ methode worden alle sleutels, met behulp van de ‘*hash*’ methode, omgezet naar een numerieke waarde. Deze om als een index te kunnen gebruiken bij het toevoegen van de sleutels en waarden in de tabel. Het check constant of de index geen waarde heeft in de sleutels array. Indien dit zo is, voegt het toe aan de array van ‘*keys*’ en ‘*vals*’. Anders zal het naar de eerst volgende beschikbare zoeken met behulp van de voorbenoemde formule. (zie Codesnippet 21)

Get

Codesnippet 21 Double Hashing put

[code]

Lerepfshnoalghl ( zie Codesnippet 22)

Codesnippet 22 Double Hashing get

[code]

# Resultaten

Hieronder worden de resultaten van elke test per klasse in een tabel uitgebeeld.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **LineairProbing** | **QuadraticProbing** | **DoubleHashing** |
| *albusIsUnique* | 89 ms, 6 botsing | nvt | nvt |
| *allPottersPresent* | nvt | 37ms, 1 botsing | nvt |
| *fancyPantsIsPresent* | nvt | nvt |  |
| *collisionsShouldHappen* |  |  |  |

Tabel 1 Test resultaten

# Conclusie

[placeholder]

# Bronvermelding

Sedgewick R, W. K. (2011). *Algorithms.* New Jersey: Pearson Education, Inc.