Hashing

Practicum 3

Bart Endhoven 500761169 & Peter Sabel (Zabel) 500759473

2017-2018

Hashing

Practicum 3

AUTEUR

Bart Endhoven 500761169 & Peter Sabel (Zabel) 500759473

Datum

6 oktober 2017

VAK

Datastructure

Versie

2.0

© 2016 Copyright Hogeschool Amsterdam

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand, of openbaar gemaakt, in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door print-outs, kopieën, of op welke manier dan ook, zonder voorafgaande schriftelijke toestemming van de Hogeschool Amsterdam.

Inhoudsopgave

[Inleiding. 4](#_Toc497244415)

[1. Opdracht 1.1 5](#_Toc497244416)

[2. Opdracht 1.2 7](#_Toc497244417)

[2.1 Hashtable 1, LinearProbingHashing 7](#_Toc497244418)

[2.2 SeperateChaningHashing 8](#_Toc497244419)

[2.2.1 Deze maakt gebruik van SequentialSearch 9](#_Toc497244420)

[2.3 QuadraticHashing 10](#_Toc497244421)

[Samenvatting. 12](#_Toc497244422)

[Bronnen 13](#_Toc497244423)

Inleiding.

In opdracht 3 gaan we aan de slag met Hashing, er wordt van ons verwacht dat wij de Ldap van een student (het studentnummer) om te zetten in een hashcode. Hiervoor hebben wij wel een Long gebruikt i.p.v. een Int omdat onze hash te lang werd voor een int. Deze hash wordt de key in de hashtabels die gemaakt gaan worden, het automatisch gegenereerde Ect’s worden de values. Er zullen 4 testen uitgevoerd worden in 3 verschillende hashtabels. Dit zorgt voor een totale test van 12 onderdelen.

# Opdracht 1.1

Er werd gevraagd een fuctie te schrijven om de Ldap van een student om te zetten in een hashcode. Wij hebben gekozen voor een Long omdat de door ons bedachte hash niet in een Int past.

/\*\*

**\*** creer een hashcode van 9 cijfers waarbij het eerste cijfer de lengte is

**\*** de string van de letters worden omgezet in een cijfer afhankelijk van de

**\*** plek in het alfabet. a = 01**,** b = 02 etc.

**\*** **@param** ldap

**\*** **@return**

  \*/

**public** **static** long hashing(String ldap) {

long hash;

int lengte = ldap.length();

**List**<**String**> t = **new** ArrayList<>();

int cijfer1 = 0;

int cijfer2 = 0;

int cijfer3 = 0;

int cijfer4 = 0;

**for** (int i = 0; i < ldap.length(); ++i) {

int n;

**try** {

char ch = ldap.charAt(i);

n = (int) ch - (int) 'a' + 1;

} **catch** (**Exception** e) {

n = 0;

}

t.add(**String**.valueOf(n));

}

**String**[] cijfers = **new** **String**[t.size()];

t.toArray(cijfers);

**try** {

cijfer1 = **Integer**.parseInt(cijfers[0]);

cijfer2 = **Integer**.parseInt(cijfers[1]);

cijfer3 = **Integer**.parseInt(cijfers[2]);

} **catch** (**Exception** e) {

**System**.out.println("somthing went wrong");

}

**try** {

cijfer4 = **Integer**.parseInt(cijfers[3]);

} **catch** (**Exception** e) {

}

hash = (lengte \* 100000000) + (cijfer1 \* 1000000) +

(cijfer2 \* 10000) + (cijfer3 \* 100) + (cijfer4);

**return** hash;

}

Daarna werd er gevraagt om alle Ldap’s van de studenten te hashen met onze geschreven functie. De modulo 97 hiervan te nemen en deze om te zetten in een grafiek. Dit is gedaan via Excel.

Hier zijn we dat de modulo 9 maar 6 keer voorkomt en de modulo 64 zelfs 23 keer voorkomt.

# Opdracht 1.2

Voor deze opdracht moet je drie hash tables (of eigenlijk drie collision resolution strategieën) onderzoeken, hiervoor hebben wij 3 classes aangemaakt die een hashtable bevatten. Wij vullen deze tables met de hiervoor omschreven gehashde Ldap als Key en den Ects als Value.

De key is de plek van de array waar wij onze value in willen stoppen. Mocht er nou al iets zitten op deze plek van de array dan betekend dit dat we een collision hebben. Deze willen wij graag bijhouden en dat doen we simpelweg door deze op te tellen iedere keer als we er 1 tegenkomen.

## Hashtable 1, LinearProbingHashing

**package** nl.hva.dmci.ict.datastructures;

/\*\*

**\***

**\*** **@author** **Bart en Peter**

**\*** **@param** **<Key>**

**\*** **@param** **<Value>**

 \*/

**public** **class** LinearProbingHashST<Key, Value> {

**private** int N; // number of key-value pairs in the table

**private** **final** int M = 10501; // size of linear-probing table

**private** **final** **Key**[] keys; // the keys

**private** **final** Value[] vals; // the values

**private** long Collisions = 0; // keeps track of the number of collisions

**public** LinearProbingHashST() {

keys = (**Key**[]) **new** **Object**[M];

vals = (Value[]) **new** **Object**[M];

}

**private** int hash(**Key** key) {

**return** (key.hashCode() & 0x7fffffff) % M;

}

**public** void put(**Key** key, Value val) {

int i;

**for** (i = hash(key); keys[i] != **null**; i = (i + 1) % M) {

**if** (keys[i].equals(key)) {

vals[i] = val;

**return**;

}

Collisions++;

}

keys[i] = key;

vals[i] = val;

N++;

}

**public** long getCollisions() {

**return** Collisions;

}

**public** int getN() {

**return** N;

}

**public** int getM() {

**return** M;

}

**public** Value get(**Key** key) {

**for** (int i = hash(key); keys[i] != **null**; i = (i + 1) % M) {

**if** (keys[i].equals(key)) {

**return** vals[i];

}

}

**return** **null**;

}

}

## SeperateChaningHashing

**package** nl.hva.dmci.ict.datastructures;

/\*\*

**\***

**\*** **@author** **Bart en Peter**

**\*** **@param** **<Key>**

**\*** **@param** **<Value>**

 \*/

**public** **class** SeparateChainingHashST<Key, Value> {

**private** int N; // number of key-value pairs

**private** **final** int M = 10501; // hash table size

**private** int Collisions = 0; // keeps track of the number of collisions

**private** **final** SequentialSearchST<**Key**, Value>[] st; // array of ST objects

**public** SeparateChainingHashST() { // Create M linked lists.

st = (SequentialSearchST<**Key**, Value>[]) **new** SequentialSearchST[M];

**for** (int i = 0; i < M; i++) {

st[i] = **new** SequentialSearchST();

}

}

**public** int getCollisions() {

**return** Collisions;

}

**public** int getM() {

**return** M;

}

**private** int hash(**Key** key) {

**return** (key.hashCode() & 0x7fffffff) % M;

}

**public** Value get(**Key** key) {

**return** (Value) st[hash(key)].get(key);

}

**public** void put(**Key** key, Value val) {

Collisions = Collisions + st[hash(key)].put(key, val);

}

**public** Iterable<**Key**> keys() {

// See Exercise 3.4.19.

**return** **null**;

}

}

Deze maakt gebruik van Sequential Searching

**package** nl.hva.dmci.ict.datastructures;

/\*\*

**\***

**\*** **@author** **Bart en Peter**

**\*** **@param** **<Key>**

**\*** **@param** **<Value>**

 \*/

**public** **class** SequentialSearchST<Key, Value> {

**private** Node first; //first node in the linked list

**private** int Collisions = 0; //keeps track of the number of collisions

**private** **class** Node { // linked-list node

**Key** key;

Value val;

Node next;

**public** Node(**Key** key, Value val, Node next) {

**this**.key = key;

**this**.val = val;

**this**.next = next;

}

}

**public** Value get(**Key** key) { // Search for key, return associated value.

**for** (Node x = first; x != **null**; x = x.next) {

**if** (key.equals(x.key)) {

**return** x.val; // search hit

}

}

**return** **null**; // search miss

}

**public** int put(**Key** key, Value val) { // Search for key. Update value if found; grow table if new.

**for** (Node x = first; x != **null**; x = x.next) {

**if** (key.equals(x.key)) {

x.val = val;

**break**;

} // Search hit: update val.

Collisions++;

}

first = **new** Node(key, val, first); // Search miss: add new node

**return** Collisions;

}

**public** int getCollisions() {

**return** Collisions;

}

}

## QuadraticHashing

**package** nl.hva.dmci.ict.datastructures;

/\*\*

**\***

**\*** **@author** **Bart en Peter**

**\*** **@param** **<Key>**

**\*** **@param** **<Value>**

 \*/

**public** **class** QuadraticHashST<Key, Value> {

**private** int N; // number of key-value pairs in the table

**private** **final** int M = 10501; // size of linear-probing table

**private** **final** **Key**[] keys; // the keys

**private** **final** Value[] vals; // the values

**private** long Collisions = 0; // keeps track of the number of collisions

**public** QuadraticHashST() {

keys = (**Key**[]) **new** **Object**[M];

vals = (Value[]) **new** **Object**[M];

}

**private** int hash(**Key** key) {

**return** (key.hashCode() & 0x7fffffff) % M;

}

**public** void put(Key key, Value val) {

int i;

int j = 1;

**for** (i = hash(key); keys[i] != **null**; i = (i+(j\*j)) % M) {

**if** (keys[i].equals(key)) {

vals[i] = val;

**return**;

}

Collisions++;

j++;

}

keys[i] = key;

vals[i] = val;

N++;

}

**public** long getCollisions() {

**return** Collisions;

}

**public** int getN() {

**return** N;

}

**public** int getM() {

**return** M;

}

**public** Value get(Key key) {

int j = 1;

**for** (int i = hash(key); keys[i] != **null**; i = (i + (j\*j)) % M) {

**if** (keys[i].equals(key)) {

**return** vals[i];

}

j++;

}

**return** **null**;

}

}

***Testen***

Er werd gevraagd om aan te tonen dat alle studenten terug gevonden konden worden, hier onze gebruikte code en het resultaat:

for(Student student : studenten){

**System**.out.println(tab1.get(hashing(student.getLdap())));

**System**.out.println(tab2.get(hashing(student.getLdap())));

**System**.out.println(tab3.get(hashing(student.getLdap())));

}

LinearProbingHashing: 124

SeparateChainingHashing: 124

QuadraticHashing: 124

LinearProbingHashing: 9

SeparateChainingHashing: 9

QuadraticHashing: 9

LinearProbingHashing: 212

SeparateChainingHashing: 212

QuadraticHashing: 212

LinearProbingHashing: 8

SeparateChainingHashing: 8

QuadraticHashing: 8

LinearProbingHashing: 19

SeparateChainingHashing: 19

QuadraticHashing: 19

LinearProbingHashing: 51

SeparateChainingHashing: 51

QuadraticHashing: 51

LinearProbingHashing: 235

SeparateChainingHashing: 235

QuadraticHashing: 235

LinearProbingHashing: 218

SeparateChainingHashing: 218

QuadraticHashing: 218

LinearProbingHashing: 66

SeparateChainingHashing: 66

QuadraticHashing: 66

LinearProbingHashing: 123

SeparateChainingHashing: 123

QuadraticHashing: 123

LinearProbingHashing: 86

SeparateChainingHashing: 86

QuadraticHashing: 86

LinearProbingHashing: 164

SeparateChainingHashing: 164

QuadraticHashing: 164

LinearProbingHashing: 118

SeparateChainingHashing: 118

QuadraticHashing: 118

Hieronder het aantal collisions per manier

LinearProbingHashing

number of collisions: 15 With a M size of 13

SeparateChainingHashing

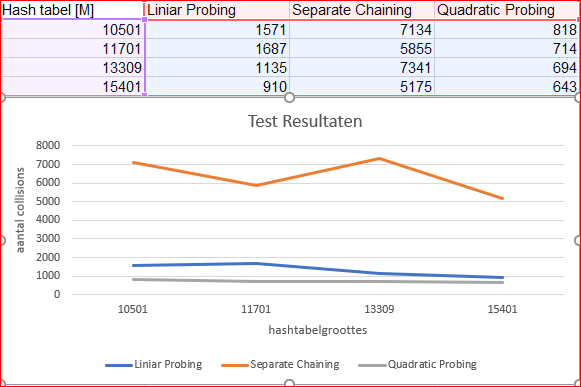
number of collisions: 4 With a M size of 13

QuadraticHashing

number of collisions: 9 With a M size of 13

# Testen

Om de verschillen in de verschillende manieren van *collision resolution* te testen hebben we voor iedere manier vier testen gedaan met verschillende grootte hash tabellen(10501,11701, 13309, 15401).  
Het bijhouden van het aantal collisions hebben we gedaan door per manier een variabele bij te houden, en die te verhogen per collision. De code hiervoor is per class in de bijlage te zien.  
Hieronder staan de testresultaten:



Samenvatting.

Bronnen

## 