Data Structures

Assignment 3

door

BURAK INAN
500740123

Dit document betreft het verslag van Assignment 3 van Data Structures.

HBO-ICT Software Engineering
Hogeschool van Amsterdam
Amsterdam, Nederland

7/12/2018

Inhoudsopgave

Inleiding

In dit verslag wordt beschreven hoe de collisions worden gedetecteerd door het programma, per resolution strategy. Uiteindelijk worden de verschillen in collision detections vergelijkt tussen alle drie de strategieën, en gevisualiseerd in een tabel en grafiek.

Een collision houdt in dat wanneer je bij het invoeren van een element in een HashTable, het kan voorkomen dat er op de plek van de toevoeging al een element bestaat. Soms kan je het element die op die plek staat, gewoon verwijderen. Wanneer je dit element niet kan verwijderen kunnen de volgende methodes gebruikt worden om deze collisions te verhelpen:

- **Linear probing**
- **Quadratic probing**
- **Double hashing**

Implementaties

In dit hoofdstuk worden de strategieën beschreven die zijn gebruikt om collisions te berekenen en te verhelpen.

Linear probing

Bij Linear probing is het zo dat wanneer er een collision wordt veroorzaakt bij het invoeren van een element in een HashTable, dat de positie met 1 wordt opgeschoven.

De stap die ik hiervoor gebruik is (slides):

- **Bij alle strategieën wordt de positie op dezelfde manier berekend:**

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

put():

- **Met behulp van de positie die hiervoor berekend is, wordt er gekeken of er op die plaats al een element bestaat, en dit wordt net zo lang gedaan totdat we die vrij is:**

```
int i = hash(key); // Index
while (keys[i] != null) {
    i = (i + 1) % maxSize;
    collisions++;
}
```

- **Elke keer dat we op zo'n plek komen die niet vrij is wordt er een collision bij opgeteld, en wordt de positie verhoogd met 1, en vervolgens wordt daar de modulo van berekend, zodat we niet buiten de lijst komen.**
- **Wanneer we een plekje hebben gevonden, wordt het element op dat plekje geplaatst:**

```
keys[i] = key;
values[i] = value;
```

add():

- **Deze methode werkt op dezelfde manier, alleen wordt bij elke collision gekeken of we de speler hebben gevonden:**

```
int i = hash(key);
while (keys[i] != null) {
    if (keys[i].equals(key)) {
        if (!foundPlayers.contains(values[i])) {
            foundPlayers.add(values[i]);
        }
    }
    i = (i + 1) % maxSize;
}
```

- **Als dat zo is, wordt die aan de lijst toegevoegd en wordt de lijst geretourneerd.**

Quadratic probing

Bij Quadratic probing wordt bij een collision de positie verhoogd met een kwadratische functie:

- Collision at x →
  - $x + 1^2$
  - $x + 2^2$
  - $x + 3^2$
  - .....

put():

- **Bij elke collision wordt er een collision opgeteld, en wordt de positie verhoogd met een kwadratische functie:**

```
while (keys[i] != null) {
    i = (i + (int) Math.pow(power++, 2)) % maxSize;
    collisions++;
}
```

- **In dit geval wordt onze x berekend in de functie die de positie berekent. Wanneer er een collision wordt veroorzaakt, wordt deze kwadratisch verhoogd. In de slides wordt bij de positie een getal in het kwadraat opgeteld. In mijn geval is dat getal 'power', bij elke botsing wordt dit getal verhoogd met 1, en dit getal in het kwadraat is de nieuwe positie waar we gaan zoeken.**
- **We gaan net zo lang door totdat we een lege positie hebben gevonden.**

add():

- **Werkt op dezelfde manier als bij Linear probing, alleen in plaats van dat we net zo lang doorgaan totdat we een positie hebben gevonden, gaan we net zo lang door totdat we buiten de lijst zitten.**

```
while (i > 0 && i < maxSize) {
    if (keys[i].equals(key)) {
        if (!foundPlayers.contains(values[i])) {
            foundPlayers.add(values[i]);
        }
    }
    i = (power * (power++)) % maxSize;
}
```

Double hashing

Bij double hashing is de techniek om een tweede hash functie te gebruiken. Bij deze functie wordt een priemgetal gebruikt om de positie te berekenen.
De formule hierbij is: hash2 = priem - (key % priem).

Bij de vorige twee strategieën zijn er telkens dezelfde stapgrootten. Bij double hashing hangt de stapgrootte af van de key die wordt meegegeven.

put():

- **Net zoals bij Linear probing gaan we kijken of er een collision ontstaat. Als dat zo is voeren we de tweede hashfunctie uit en tellen we deze op bij onze huidige positie. Vervolgens pakken we hier de modulo van, zodat we niet buiten de onze lijst komen:**

```
while (values[i] != null) {
    this.collisions++;
    i += hash2(key); // Nieuwe index bij collision
    i %= maxSize;
}
```

- **Zodra er een plekje is gevonden voor het element, wordt deze toegevoegd:**

```
keys[i] = key;
values[i] = value;
```

add():

- **Werkt gewoon op dezelfde manier, net zoals bij alle andere functies:**

```
while (keys[i] != null) {
    if (keys[i].equals(key)) {
        if (!foundPlayers.contains(values[i])) {
            foundPlayers.add(values[i]);
        }
    }
    i += hash2(key);
    i %= maxSize;
}
```

- **Wanneer een zoekopdracht gelijk is aan wat we zoeken (een collision), wordt deze toegevoegd aan de lijst, en wordt de nieuwe positie bepaald.**

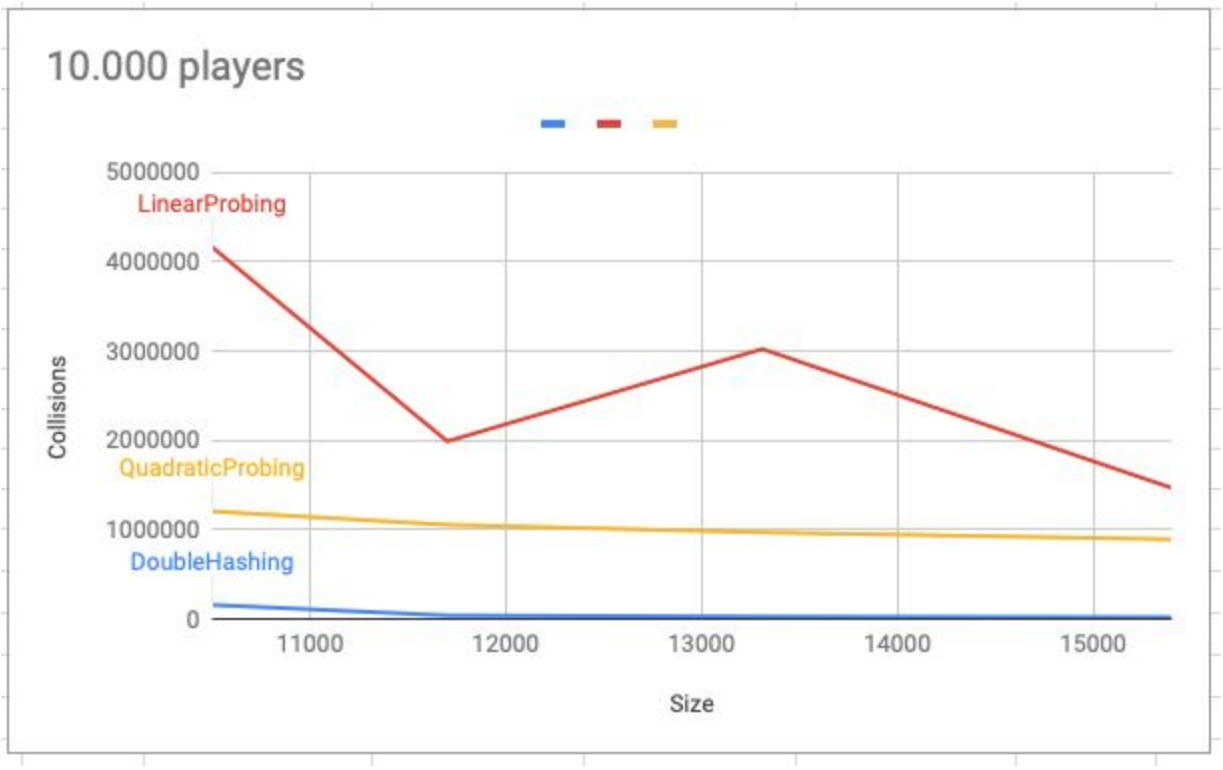
Metingen

De onderstaande getallen zijn de gemiddelden uit 10 tests. Het blijkt dat bij het gebruik van Linear probing bij 13309 elementen, meer collisions ontstaan dan bij 11701 elementen. De minste collisions ontstaan bij Double hashing.

Tabel

10000 Players	DoubleHashing	LinearProbing	QuadraticProbing
10501	156708	4162018	1203999
11701	38472	1988870	1053788
13309	26342	3022992	969158
15401	19134	1466855	891028

Grafieken



Fair comparison

De verschillende strategieën worden niet allemaal op dezelfde manier getest in de kleinere tests. Want bij quadratic probing wordt er bijvoorbeeld gekeken of de 3 players met dezelfde achternaam "Potter" zich bevinden in de hashtable. Deze drie players hebben allemaal dezelfde key, en in een hashtable hoort elke key uniek te zijn, daarom is de implementatie van de get() methode anders dan hij normaal hoort te zijn, want deze moet meerdere elementen teruggeven.

Wat betreft de test voor het aantal collisions, denk ik dat deze wel gewoon gelijk getest wordt bij elke strategie. Het klopt dat linear probing zo veel collisions veroorzaakt, doordat je maar 1 stap zet bij elke collision.

Wat ik aan de opdracht zou verbeteren is dat elke key in de hashtable uniek moet zijn, en dat de get() methode dan gewoon 1 element teruggeeft.