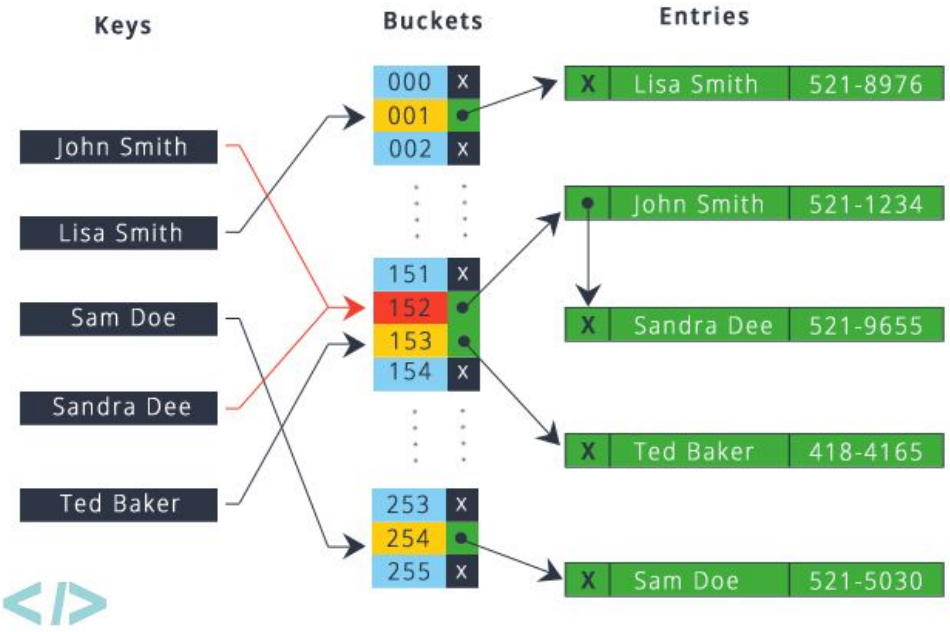
Practicum 3 Datastructures

Yacob Ben Youb en Youssef Ben Youb
500672040 - 500775494



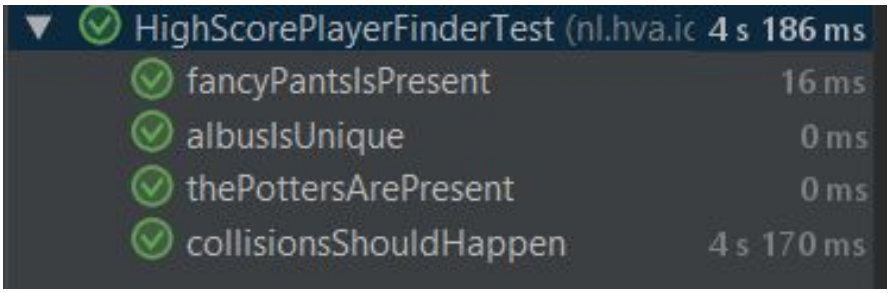


Table of contents:

The table of contents contains the structure of our report

Introduction:

This chapter contains a quick summary of the report and what subjects will be addressed

This is a report about the third datastructures practical assignment. In this report we will discuss the differences between three different types of hashtable implementations: Linear probing, Quadratic probing and Double Hashing. We will also analyse how each of these deals with collisions and the impact it has on their effective speed

Hashing Algorithms:

Collision detection code:

*(*The text that is marked in grey is code to track how many collisions have found place and is unnecessary for the functioning of the code)*

Linear Probing collision detection:

Linear probing increases the index by one if the resulted hash already contains an object.

The while loop checks whether the list at the number of index is not empty. If the list at that index already contains an element, it will move to the next index. Once it finds an empty index it will add the record into that index.

```
int collisionCounter;
   public void put(String key, Player value) {
      int index = keyHash(key);
      while (!linearProbeList.get(index).isEmpty()) {
         index++;
      collisionCounter++;
         if (index >= linearProbeList.size()) {
            index = 0;
         }
      }
           System.out.println("LinearProbing collision counter: "+ collisionCounter);
      linearProbeList.get(index).add(value);
   }
```

Hashing algorithm:

```
public int keyHash(String key) {
 int hash = 0;
 for (int i = 0; i < key.length(); i++) {
  hash = hash + key.charAt(i);
 }
 hash = hash % linearProbeList.size();
 return hash;
}
```

Quadratic probing collision detection

The while loop checks whether the list to which the record will be added is not empty and whether the first record of the list does not have the same last name as the key. It does this until both these statements are true using recursion.

```
int collisionCounter;

public void put(String key, Player value) {
   int counter = 0;
   int index = counterKeyHash(key, counter);
 while ((!quadraticProbleList.get(index).isEmpty()) &&
!quadraticProbleList.get(index).get(0).getLastName().equals(key)) {
      collisionCounter++;
      index = counterKeyHash(key, counter);
   }
   quadraticProbleList.get(index).add(value);
   System.out.println("quadratic Probing collision count: "+ collisionCounter);
}
```

If a collision has found place, increase the counter and increase the hash with the counter squared.

```
public int counterKeyHash(String key, int collisionCounter) {
 int hash = 0;
 for (int i = 0; i < key.length(); i++) {
  hash = hash + key.charAt(i);
 }
 //Nadat collision is opgetreden is wordt dit uitgevoerd
 if (collisionCounter != 0) {
  hash += (collisionCounter * collisionCounter);
 }
 hash = hash % quadraticProbleList.size();
 return hash;
}
```

Double Hashing collision detection

The collision detection with this method is a lot like the quadratic probing method. First the while loop checks if the list to add it to is not empty and then it checks whether the first record of that list doesn't have the same full name as the record that is being added.

```
int tweedeCollisionCounter;
public void put(String key, Player value) {
  int counter = 0;
  int index = doubleCounterKeyHash(key, counter);
  System.out.println(key);

  while ((!doubleHashingList.get(index).isEmpty()) &&
(getFullName(doubleHashingList.get(index).get(0))) != key) {
     counter++;
     tweedeCollisionCounter++;

     index = doubleCounterKeyHash(key, counter);
  }
  doubleHashingList.get(index).add(value);
  System.out.println("DoubleHashing CollisionCounter: "+ tweedeCollisionCounter);
}
```

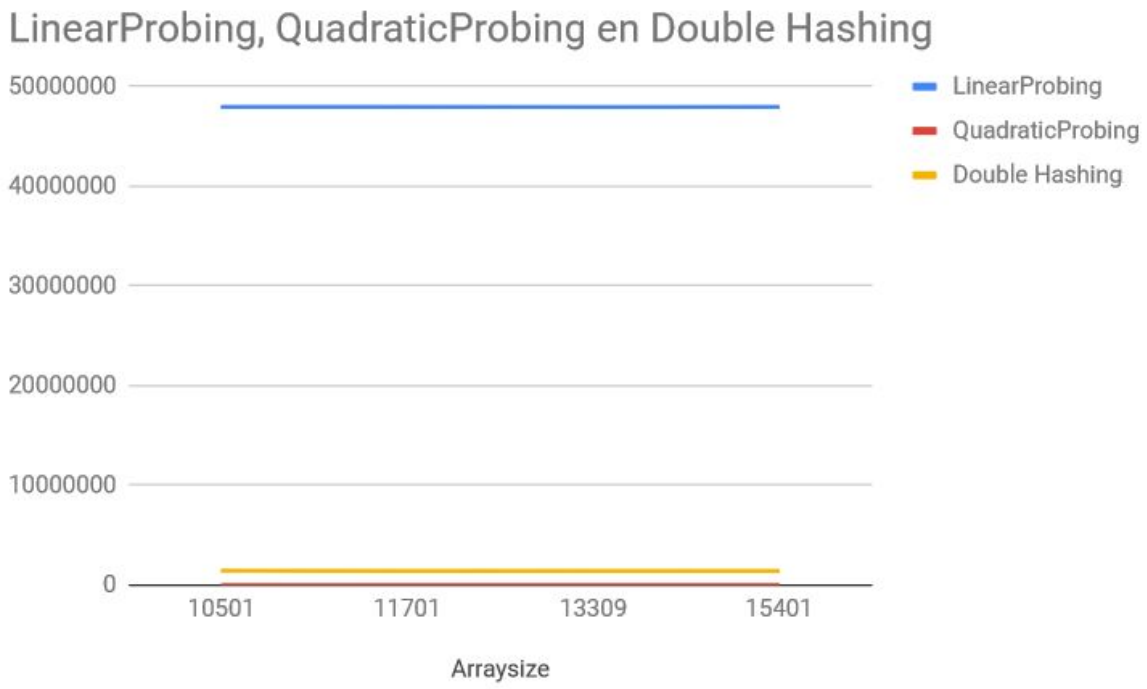
If detects collision however, it will multiply the counter with a prime number instead of squaring it. This is done because multiplying with a prime number will never result a modulo at zero.

```
public int doubleCounterKeyHash(String key, int collisionCounter) {
      int hash = 0;
      for (int i = 0; i < key.length(); i++) {
         hash = hash + key.charAt(i);
      }
      //Nadat collision is opgetreden is wordt dit uitgevoerd
      if (collisionCounter != 0) {
         hash += collisionCounter*31;
      }
      hash = hash % doubleHashingList.size();
      return hash;
   }
```

CollisionShouldHappen() test results.

Amount of collisions:

Arraysize	LinearProbing	QuadraticProbing	Double Hashing
10501	47967059	931	1452409
11701	47966450	894	1427401
13309	47953435	1026	1428905
15401	47965417	950	1429808



As visible in this graph the array size doesn't make much of a difference in the amount of collisions.

The reason linear Probing has such a bizarre amount of collisions is because it does not efficiently spread out names that have an equal hash value. This means that when names end up with an equal value after hashing, it will not only collide with the 1 other name that has an equal hash value, but all names that had an equal hash value. Names with the same value can undergo collision with each other in linear probing. Also there are only 126 different names which means that collision is going to happen at least once each time after all 126 different names have been added. And after that collision is going to happen at least

twice each time when they have all been added twice. The amount of collisions that will happen steadily rises the more records are added.

The reason quadratic probing is so efficient is because it adds names with the same value to the same list thus avoiding collision. Also there are only 75 different names, meaning that even if every name had the exact same hash value, it couldn't possibly undergo more than 75 collisions per added record.

The reason double hashing is not nearly as efficient as quadratic probing is because double hashing makes use of a lot more different names. Although it adds names that are the same to the same list avoiding collisions, there are a lot more different names. Quadratic probing only makes use of 75 different names, whilst double hashing makes use of(126*75) 9450 names. This makes it so that a lot of different names can undergo collision.

Fixed tests:

This chapter is about the tests we have edited to run. Initially they were not able to run because they asked for input that did not exist (more names than available).

shouldReadAllLastNames()

The test expected there to be 100 lastnames in the lastname.txt file, however there are only 75 available.

Therefore it has been changed to expect 75 lastnames.

shouldReadAllFirstNames()

This test expected there to be 192 firstnames in the firstname.txt file, however there are only 126 available.

Therefore it has been changed to expect 126 firstnames.

shouldContainKnownLastNames()

In this test case the lastnames.txt file is searched through for certain names. One of them being "van der pol". Unfortunately this name does not exist in the lastnames.txt file causing the test to fail. Therefore it has been changed to no longer expect "van der pol".

Conclusion:

It is quite apparent that this was not a fair comparison at all. The different hashing methods had very different conditions not making it transparent how they would perform in the same environment:

Linear probing was at a huge disadvantage: It did not add to the same list like with quadratic probing and double hashing but instead made a new list for each name. On top of that it only used 126 different names which means a lot of collisions were bound to happen anyway

Quadratic probing also had an unfair advantage: Quadratic probing added records with the same name to the same list avoiding collision. On top of this there were only 76 different names so it was avoiding a lot of collisions this way.

Double Hashing: As with quadratic probing it added records with the same name to the same list. However, double hashing made use of a potential 9450 different names because it combined the first and last name. Meaning it didn’t avoid nearly as much collisions as quadratic probing did.