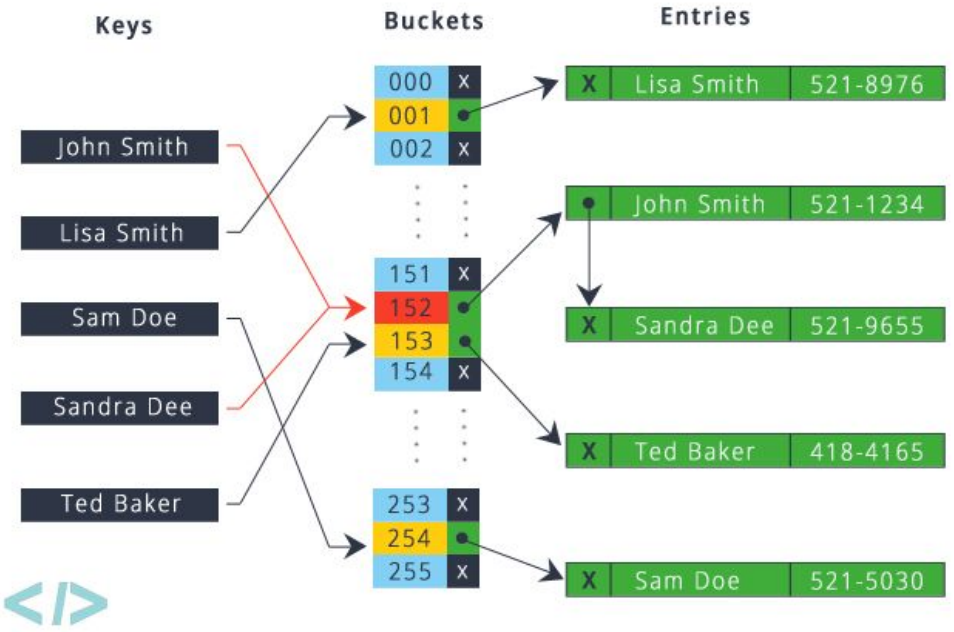
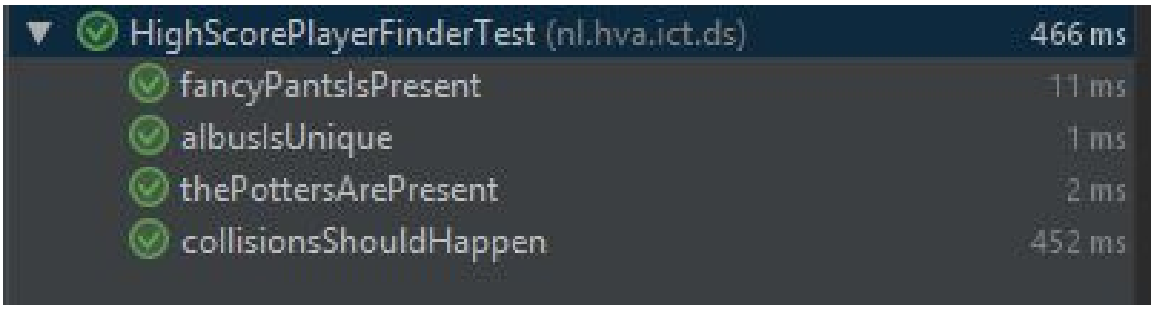
Practicum 3 Datastructures

Yacob Ben Youb en Youssef Ben Youb
500672040 - 500775494





1 Table of contents:

The table of contents contains the structure of our report

2 Introduction:

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

This is a report about the thirdDatastructures 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.

3 What is a hashtable?

A hashtable is a type of data structure that stores its elements at an index with a certain key. This index then contains the element value. What makes the hashtable unique and different than normal array types, is that the index an element is stored at is correlated to the element itself. By performing a calculation on a chosen type of data of the element, we can generate a number and then store our element in this index number of the array. If this index is already occupied, we perform the calculation again.

This method heavily decreases the amount it takes to find a certain object, because the computer can calculate the index of the element again in order to find it.

This project is about comparing different algorithms to calculate our our hash index with, in order to find an empty slot for our item as fast as possible.

4 Classes explanation:

In this chapter a short summary of each class is given.

4.1 General methods:

A short explanation of methods which recur in all hashing classes

4.1.1 The HashTables:

Each class contains a list (HashTable). Because for our tests some HashTables can have multiple elements in one hash index, each HashTable index contains yet another list, and this list then contains our element(s).

4.1.2 The keyHash() methods:

These methods generate hashes for our Player. Each hash class uses a different unique way to generate hashes for our player.

4.1.3 The put() methods

These use the Keyhash() method of their respective class to generate an index for our player in the list of playerlists, and checks whether it can add the player to that index or needs to generate a new hash.

4.2 Hashing Algorithms:Linear probing:

4.2.1 Put()

First we calculate an index number (hash) for our element by getting the number of each letter in the alphabet, multiplying it by the prime number 461, and adding them all up. Then we use module to cut off numbers above the maximum length of our list.

The method then checks whether the index of this array is empty. If there is nothing our element will be placed there. If the index of the array already contains an element, our algorithm will simply increment the index by one until it finds an empty index. Once it finds an empty index it will add the record into that index.

```
public void put(String key, Player value) {
   int index = keyHash(key);
   while (!(linearProbeList[index] == null)) {
      index++;
      linearCollisionCount++;
      if (index >= linearProbeList.length) {
         index = 0;
      }
   }
   linearProbeList[index] = (value);
   System.out.println("linear probe collision count: " + getLinearCollisionCount());
}
```

Hashing algorithm:

```
public int keyHash(String key) {
      int hash = 0;
      for (int i = 0; i < key.length(); i++) {
         hash = hash + key.charAt(i)*461;
      }
      hash = hash % linearProbeList.length;
      return hash;
   }
}
```

4.2.2 get()

The get method hashes the key that is inputted and looks for it at the resulting index. If it is not there it will continue to add 1 to the index until it finds the inputted key. If it finds a match it will continue until it finds a mismatch. It adds all the matches into a list and returns that list.

```
public List<Player> get(String key) {
    List<Player> returnList = new ArrayList<>();
    int index = keyHash(key);
    while (!linearProbeList[index].getFirstName().equals(key)) {
        index++;
        linearCollisionCount++;
        if (index >= linearProbeList.length) {
            index = 0;
        }
    }
    while (linearProbeList[index].getFirstName().equals(key)) {
        returnList.add(linearProbeList[index]);
        index++;
        linearCollisionCount++;
        if (index >= linearProbeList.length) {
            index = 0;
        }
    }
    return returnList;
}
```

4.3 Quadratic probing collision detection

As its name suggests, quadratic probing squares a number during the hash. This is done so indexes which would result in similar secondary hashes become completely different after collisions find place.

First we calculate an index number for our element. This hash is similar to the one used in linear probing. But then it becomes different. The big difference between quadratic and linear probing is that instead of incrementing the index on collision, quadratic probing multiplies the number with itself. For the initial index this does not matter very much, but after finding collisions this drastically speeds up the time to find a new index.

```
public void put(String key, Player value) {

   int collisionCounter = 0;
   int index = counterKeyHash(key, collisionCounter);
   while (!(quadraticProbleList[index] == null)) {
      collisionCounter++;
      quadraticCollisionCount++;

      index = counterKeyHash(key, collisionCounter);
   }
   quadraticProbleList[index] = (value);
   System.out.println("quadratic probe collision count: " + getQuadraticCollisionCount());
}
```

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)*461;
      }
      //Nadat collision is opgetreden is wordt dit uitgevoerd
      if (collisionCounter != 0) {
         hash += collisionCounter + 3 + (collisionCounter * collisionCounter);
      }
      if (hash < 0) {
         hash = 1;
      }

      hash = hash % quadraticProbleList.length;
      return hash;
   }
```

4.3.1 Get()

This method hashes the given key using the hash method and then looks for it at the resulting index. If it is not there it will increment the collisioncounter by one and try again. It continues until it has found a match. After it has found a match it will continue until it finds a mismatch.

```
public List<Player> get(String key) {
    List<Player> returnList = new ArrayList<>();
    int collisionCounter = 0;
    int index = counterKeyHash(key, collisionCounter);
    while (!(quadraticProbleList[index].getLastName().equals(key))) {
        collisionCounter++;
        quadraticCollisionCount++;
        index = counterKeyHash(key, collisionCounter);
    }
    while (quadraticProbleList[index].getLastName().equals(key)) {
        returnList.add(quadraticProbleList[index]);

        collisionCounter++;
        quadraticCollisionCount++;
        index = counterKeyHash(key, collisionCounter);
        printAllNamesInArray(returnList);
    }
    return returnList;
}
```

4.4 Double Hashing collision detection

First double hashing makes use of a normal hash to calculate an index. If the index is filled in the array it will then make use of a different hash.

If the index is filled again it will continue to use the second hash in combination with quadratic probing until it finds an empty index in the array.

```
public void put(String key, Player value) {
      int collisionCounter = 0;
      int index = doubleCounterKeyHash(key, collisionCounter);
      while (!(doubleHashingList[index] == null)) {
         collisionCounter++;
         doubleHashCollisionCount++;
         index = doubleCounterKeyHash(key, collisionCounter);
      }
      doubleHashingList[index] = (value);
      System.out.println("double hash collision count: "+getDoubleHashCollisionCount());
   }
```

```
public int doubleCounterKeyHash(String key, int collisionCounter) {
   int hash = 0;
   //Nadat collision is opgetreden is wordt dit uitgevoerd
   if (collisionCounter == 0) {
      for (int i = 0; i < key.length(); i++) {
         hash = hash + key.charAt(i) * 457;
      }
   }
   //Nadat collision is opgetreden is wordt dit uitgevoerd
   if (collisionCounter >= 1) {
      for (int i = 0; i < key.length(); i++) {
         hash = hash + key.charAt(i) * 509;
      }
      hash += collisionCounter * collisionCounter;
   }

   if (hash < 0) {
      hash = 0;
   }

   hash = hash % doubleHashingList.length;
   return hash;
}
```

Get()

This method works similarly to the get method from quadratic probing:
First it hashes the key, looks for a match at the resulting index, and if it's not there it increments the collision counter by one and tries again. It keeps going until it finds a match. After it has found a match it will keep going until it gets a mismatch.

```
public List<Player> get(String key) {
    List<Player> returnList = new ArrayList<>();
    int collisionCounter = 0;
    int index = doubleCounterKeyHash(key, collisionCounter);
    while (!(getFullName(doubleHashingList[index]).equals(key)))
        System.out.println("doublehashing");
        collisionCounter++;
        doubleHashCollisionCount++;
        index = doubleCounterKeyHash(key, collisionCounter);
    }
    while ((getFullName(doubleHashingList[index]).equals(key))) {
       System.out.println(getFullName(doubleHashingList[index]));
        System.out.println("adding target to list");
        returnList.add(doubleHashingList[index]);
        collisionCounter++;
        doubleHashCollisionCount++;
        index = doubleCounterKeyHash(key, collisionCounter);
        printAllNamesInArray(returnList);
    }
    return returnList;
  }
```

5 Tests

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

5.1 Fixed tests:

5.1.1 thePottersArePresent()

According to Weiss, Mark Allen (2009) in *Data Structures and Algorithm Analysis in C++. Pearson Education.* "*For linear probing it is a bad idea to let the hash table get nearly full, because performance is degraded as the hash table gets filled. In the case of quadratic probing, the situation is even more drastic. With the exception of the triangular number case for a power-of-two-sized hash table, there is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions.*"

ThePottersArePresent is a test with 6 names in a hashtable of 7. This caused an infinite loop of collisions. There are two solutions to this problem

1) Remove a name from the list index as soon as it is found, remember the index and reinstate the names on the indices when the get method is finished.
2) Make sure any list which is hashed is at least twice the size of the total amount of players in them.

For this test, the size of the list has been adjusted to 13 to support more players. This caused the amount of collisions to go from ~65000 to 4.

5.1.2 shouldReadAllLastNames()

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

5.1.3 shouldReadAllFirstNames()

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

5.1.4 shouldContainKnownLastNames()

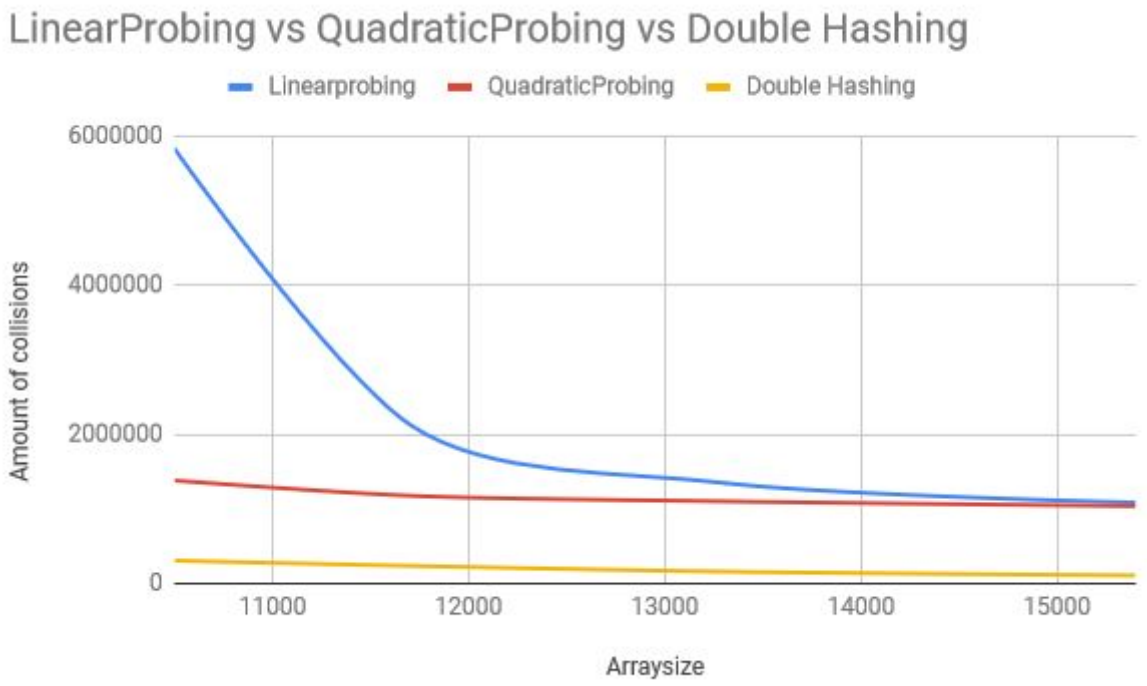
In this test case the lastnames.txt file is searched through for certain names. One of them being "van der pol". Unfortunately this name does not exist in the lastnames.txt file causing the test to fail. Therefore the test has been changed to no longer expect "van der pol".

5.2 CollisionShouldHappen() test results.

Amount of collisions:

Arraysize	Linearprobing	QuadraticProbe	Double Hashing
10501	5825256	1381251	306954
11701	2129162	1177144	237151
13309	1346759	1101791	159511
15401	1085365	1037985	108094

Amount of collisions VS Array size



The test results show Linear Probing to have the highest amount of collisions, Quadratic Probing the second highest amount of collisions and double hashing to have the least amount of collisions. This remains true regardless of the array size used.

5.3 Linear Probing

The reason Linear probing has the highest amount of collisions is because Linear probing doesn't spread out its records if it encounters collision. This causes it to create a big amount of records in succession in the array. Because of this, a new record that encounters collision might very well have to undergo a large amount of collisions before finding a place. On top of this there are only 126 different first names so there are bound to happen quite a few collisions.

A larger array size decreases the chances of an initial collision and cause the records to be spread out better. A larger array size drastically increases the performance of Linear Probing.

5.4 Quadratic Probing

Quadratic Probing is different from linear probing in that it is more likely to find an empty space faster after a collision. This is because quadratic probing looks for empty spaces further and further apart from the collision depending on the amount of collisions. This causes the records to be spread out better and causes the records to be placed faster after collision than with linear probing.

A problem with this test is that it makes use of the “last names file” which only contains 75 last names. This means a lot of collisions will happen because identical keys will result in identical hashes. Despite this is still results in fewer collisions than the Linear Probing algorithm

5.5 Double Hashing

Double hashing results in the least amount of collisions. This is because double hashing makes use of a second hash if the first hash encounters collision. This causes records to be spread out very well and allows records to find empty spaces very quickly after an initial collision.

On top of this it makes use of first names and last names to determine collisions. This causes fewer collisions to happen because of records with the same name.

6 Conclusion

According to the testing method provided by the UUAS Double hashing is the best hashing method for the high score list, as it results in the fewest amount of collisions in all cases.

6.1 Discussion:

A problem found with the testing method is that the tests could be considered unfair. Each hashing method worked with a different amount of unique keys for their respective test. For example double hashing had a higher variety of unique keys in comparison to linear probing and quadratic probing.