Hyperiondev

TASK

Algorithms - Hashing, Shuffling and Composition

Visit our website

Introduction

WELCOME TO THE ALGORITHMS - HASHING, SHUFFLING AND COMPOSITION TASK!

In this task, you will be looking at hashing algorithms, as well as some other useful algorithms that are used within the Java Collections Framework.

HASHING

Before we look at what hashing is, we need to understand what a map is. A map is a data structure that is implemented using hashing. As you may recall from JavaScript, a map is a container object that stores entries, each of which contains two parts; a key and a value. The key is used to search for the corresponding value. An example of a map is a dictionary, where the words are keys and their definitions are the values. In fact, another name of a map is actually a dictionary. A map is also called a hash table or an associative array.

Hashing is a very efficient way to search for an element. Hashing can be used to implement a map to search, insert and delete an element in O(1) time (You will learn about what that means in the Big O Notation task next!). If you know what the index of an element is, you can retrieve that element in O(1) time. So that means we are able to store the values in an array and use the key as the index to find the value. That is, of course, if you can map a key to an index.

The array that stores the values is called a hash table. The function or algorithm that maps a key to an index in the hash table is called a hash function. Hashing is a technique that retrieves the value using the index obtained from the key without performing a search.

As an example to help you understand how hashing works, suppose that we want to map a list of String keys to String values. In this example, we are going to map a list of countries to their capital cities. Let's say that the data in the table below are stored in a map.

Key	Value
England	London
Australia	Canberra
France	Paris
Spain	Madrid

Now, let's suppose that our hash function simply determines the length of the String. We have two arrays; one to store the keys and another to store the values. Our hash function converts a key to an integer value called a hash code. To put an item into the hash table, we need to compute the hash code, which in this case, is to count the number of characters and put the key and value in the arrays at the corresponding index.
The table below shows how a hash function obtains an index from a key and uses the index to retrieve the value for the key. Spain, for example, has a length (hash code) of 5, so it is stored in position 5 in the keys array, and Madrid is stored in position 5 of the values array. In the end, we should end up with the following:

Position	Keys array	Values array
1		
2		
3		
4		
5	Spain	Madrid
6	France	Paris
7	England	London
8		
9	Australia	Canberra

Notice that our arrays must be able to accommodate the longest String, which in this case is Australia. Some space is inevitably wasted because there are no keys that are fewer than 5 letters, for example. Nor is there an 8-letter key. This is not a big disaster in this case. However, you can easily see that this method would not work for storing arbitrary Strings. Imagine if one of your String keys were 10 000 characters long but the rest were under 100 characters long. The majority of the

space in the array would go to waste. Also, what would happen if you have more than one key with the same hash code? For example Egypt and Spain (they both have a length of 5).

Ideally, we would like to design a function that maps each key to a different index in the hash table. These functions are known as perfect hash functions. Perfect hash functions are difficult to find, however. A collision occurs when two or more keys are mapped to the same hash value (for example, Egypt and Spain). There are ways to deal with collisions. However, it is better to just avoid them in the first place. You should aim to design fast and easy to compute hash functions that minimise collisions.

HASH FUNCTIONS

The `hashCode` method, which returns an integer hash code, is found in the Object class. This method returns the memory address for the object by default. The `hashCode` method has different implementations in different classes. For example, the following code:

```
Integer object1 = new Integer(1234);
String object2 = new String("1234");
System.out.println("hashCode for an integer 1234 is " + object1.hashCode());
System.out.println("hashCode for a String 1234 is " + object2.hashCode());
```

Will print out:

```
hashCode for an integer 1234 is 1234
hashCode for a String 1234 is 1509442
```

You should override the default hash functions to provide one that better handles your data.

HASH CODES FOR PRIMITIVE DATA TYPES

If your keys are of type byte, short, int, or char, you can simply cast them into an int. Two different keys of any one of these types will, therefore, have different hash codes.

If your key is of type float, you can use `Float.floatToIntBits(key)` as the hash code. **floatToIntBits(float f)** returns an int value whose bit representation is the

same as the bit representation for the floating number f. Two different keys of type float will, therefore, have different hash codes.

If your key is of the type long, you cannot simply cast it to int. This is because all keys that differ in only the first 32 bits will have the same hash code. You, therefore, need to divide the 64 bits into two halves and perform the exclusive-or operation (^) to combine the two halves in order to take the first 32 bits into account. This is called folding. The hash code for a long keyword is:

```
int hashCode = (int)(key ^ (key >> 32));
```

You might be a little unfamiliar with some of the symbols in the hash code above. Let's take a closer look. >> is known as the right shift operator. It shifts the bits 32 positions to the right. ^ is the exclusive-or operator. It takes two-bit patterns of equal length and performs the **exclusive-or** operation on each pair of corresponding bits. For example, 1010110 ^ 0110111 will give you 1100001.
For a key of type double, you need to first convert it to a long value using Double.doubleToLongBits and then perform a folding like with any long value. The hash code for a double keyword is:

```
long bits = Double.doubleToLongBits(key);
int hashCode = (int)(bits ^ (bits >> 32));
```

HASH CODES FOR STRINGS

Most often, keys are String values. It is therefore important that you design a good hash function for them. One approach is simply to add the Unicode of all the characters in the String together as the hash code. This can work if no two keys contain the same letters. However, it will produce collisions if they do contain the same letters. An example of this is the words coin and icon. Both of these words contain exactly the same letters.

A better hash function is one that generates a hash code that takes the position of each letter into account. This hash code is:

$s_0 * b(n - 1) + s_1 * b(n -2) + ... + s_{n-1}$

where, si is s.charAt(i). This is called a polynomial hash code as it is a polynomial for some positive value b. You can use **Horner's rule** to calculate the hash code efficiently:

$(\ldots((s_0 * b + s_1)b + s_2)b + \ldots + s_{n-2})b + s_{n-1}$

An appropriate value for b should be chosen to minimise the number of collisions. According to experiments that were conducted, good choices for b are 31, 33, 37, 39, and 41. The hashCode is overridden using the hash code above with b being 31 in the **String class**.

COMPRESSING HASH CODES

It is possible for the hash code for a key to be an extremely large integer that is out of the range for the hash-table index. You, therefore, need to scale it down to fit within the range of the index. Let us assume that the index of a hash table is between 0 and n - 1 (there are n different indices). The most common way to ensure an integer is between 0 and n - 1 is to use the following:
h(hashCode) = hashCode % n
n should be a prime number which is greater than 2 to make sure that the indices are spread evenly.

HANDLING COLLISIONS

As explained previously, when two keys are mapped to the same index in a hash table a collision occurs. There are two ways of handling collisions:

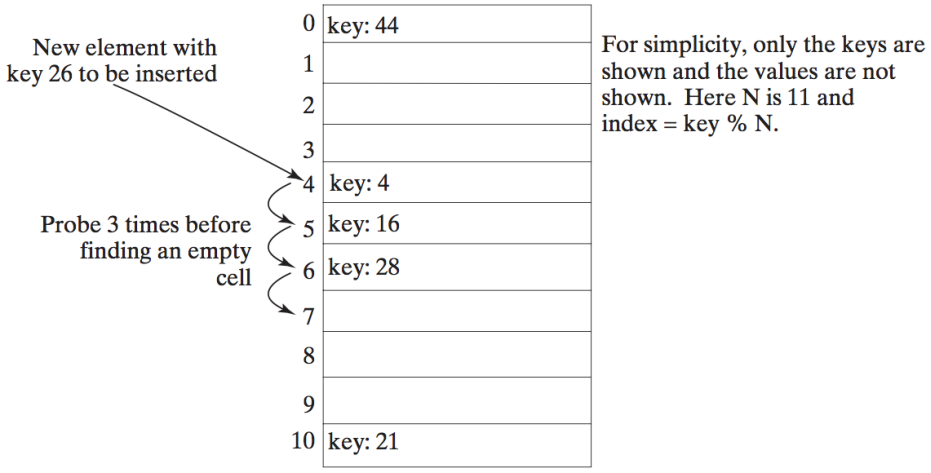
- open addressing
- separate chaining

OPEN ADDRESSING

Open addressing handles collisions by finding an open location in the hash table if a collision occurs. There are several variations of open addressing, namely: linear probing, quadratic probing, and double hashing. In this task, we will look at linear probing.

Linear probing finds the next available location when a collision occurs sequentially. If a collision occurs at `hashTable[key % n]`, for example, `hashTable[(key + 1) % n]` is checked to see if it is available and if not `hashTable[(key + 2) % n]` is checked and so on, until an available location is found. When probing reaches the end of the table, it goes back to the beginning of the table. The hash table is treated as if it were circular.

For example, take a look at the diagram below. Let's say we would like to store an element with the key 26 into a hash table that has an index between 0 and 10. Therefore, n is 11 and the index is key % n. The index for the key 26 will, therefore, be 26 % 11 = 4, however, as you can see, there is another element stored there with a key of 4. The hash table is probed three times before an empty cell is found.



In order to search for an entry in the hash table, we need to obtain the index, in this case, `k`, from the hash function for the key. Then check whether `hashtable[k % n]` contains the entry and if not check whether `hashTable[(k+1) % n]` contains the entry, and so on, until it is found or alternatively, an empty cell is reached.

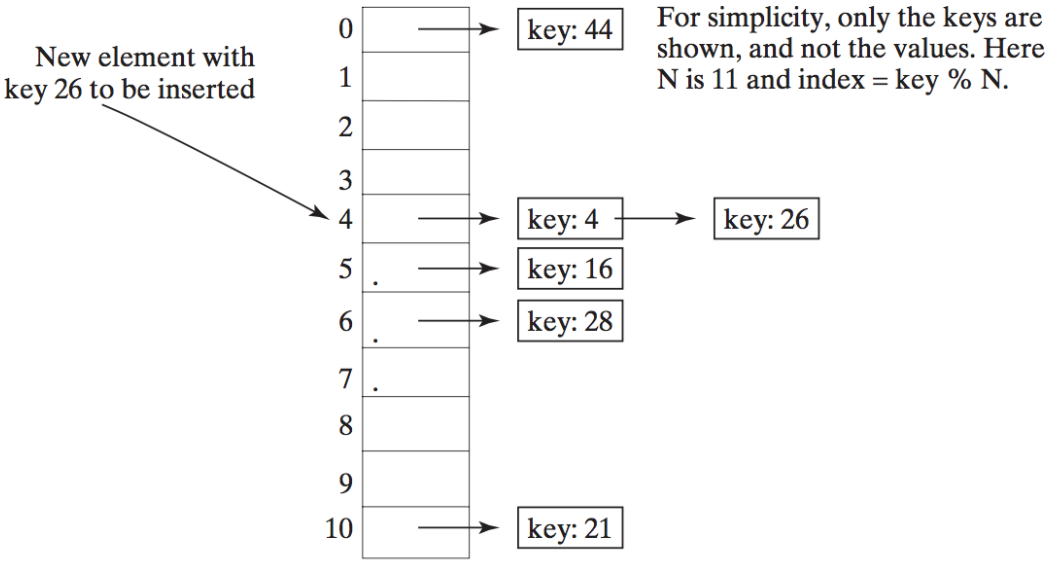
To remove an entry from the hash table, you need to search for the entry that matches the key. A special marker is placed to denote that the entry is available if it is found. In the hash table, each cell has three possible states, namely, occupied, marked or empty. A marked cell is also available for insertion.

With linear probing, groups of consecutive cells tend to be occupied. These groups are known as clusters. Each of these clusters is a probe sequence that you need to search when retrieving, adding, or removing an entry as clusters may merge with each other to create even bigger clusters as they grow. This can slow down the search time further and is a major disadvantage of linear probing.

SEPARATE CHAINING

Rather than finding new locations, separate chaining places all entries with the same hash index in the same location. Each location uses a bucket to hold multiple

entries. You can create a bucket using an array ArrayList, or LinkedList. For this task, we will be using the LinkedList. In the diagram below, you can see that each cell in the hash table is the reference to the head of a LinkedList, starting from the head, elements are chained in the LinkedList.



SHUFFLING

The shuffle algorithm destroys all traces of order that may be present in the list. In other words, the shuffle algorithm gives us a random permutation of the elements in a list. This is useful if you would like to create a program that implements a game of chance, such as a card game. It can also be useful for generating test cases.

There are two shuffle methods provided in the Java Collections Framework:

- `shuffle(List list)` — takes in a List object as an argument and uses a default source of randomness
- `shuffle(List list, Random rnd)` — requires a **Random** object to use as a source of randomness

The following code is used to shuffle a number of words in a list:

```
import java.util.*;

public class Shuffle {
   public static void main(String[] args) {

      List<String> list = new ArrayList<>();
```

```
        list.add("Java");
        list.add("Programming");
        list.add("Is");
        list.add("Fun");

        System.out.println("Original List: " + list);

        Collections.sort(list);
        System.out.println("Sorted List: " + list);

        Collections.shuffle(list);
        System.out.println("Shuffled List: " + list);

        Collections.shuffle(list);
        System.out.println("Shuffled List: " + list);
    }
}
```

Output:

```
Original List: [Java, Programming, Is, Fun]
Sorted List: [Fun, Is, Java, Programming]
Shuffled List: [Programming, Is, Fun, Java]
Shuffled List: [Is, Fun, Java, Programming]
```

Notice that the two calls of shuffle give you two different shuffled versions of the list.

COMPOSITION

The Collections Framework contains two algorithms to test some aspect of the composition of one or more collections:

- `int frequency(Collection c, Object o)` — returns the number of times a specific object occurs in the given collection.
- `boolean disjoint(Collection c1, Collection c2)` — determines whether the two specified Collections contain no elements in common.

FINDING EXTREME VALUES

The Collections Framework contains two algorithms, min and max, that allow you to find the minimum or maximum element in a Collection.

- `max(Collection c)` — Takes in a collection and returns the maximum element according to the elements' natural ordering.
- `min(Collection c)` — Takes in a collection and returns the minimum element according to the elements' natural ordering.

Compulsory Task 1

Answer the following questions:

- Create a Java file called **linearProbing.java**. Inside, write the algorithm for removing entries using linear probing.

Compulsory Task 2

Answer the following question:

- Create a diagram that shows the hash table of size 9 after entries with the keys 55, 22, 19, 1, 111, 39, 72, and 3 are inserted, using separate chaining.

Compulsory Task 3

Answer the following question:

- Create a Java file called **colours.java**. Inside, write a Java random-assignment program that allows the user to enter a list of names and a list of colours (the lists must be equal lengths).
- The program should assign each person a random colour.
- Print out the pairs (i.e. name + colour)

Optional Bonus Task

Answer the following question:

- Create a text file called **hashing.txt**. Inside, outline an algorithm that hashes a simple object with at least 2 attributes. For example, a fruit with a name and a colour.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.