HashMap Collision Avoidance Using Linked list With Java

author : gasbaoui mohammed al amine

Contents

1 Hashmap Collision Avoidance

Hashmap collision avoidance using a linked list is a technique employed in Java (and other programming languages) to handle situations where two or more keys map to the same bucket in a hashmap. When such a collision occurs, instead of overwriting the existing key-value pair, we use a linked list to store multiple key-value pairs within the same bucket. This way, we can maintain all the key-value pairs associated with the same hash value.

2 The Proposed Technique Data Structure

- ▶ We want to store a pair of data identification, description in a table with length v by avoiding the collision using a linked-list
- ▶ The position of the pair objects is calculated according to the summation order of identification property

Example pair1={"abc", "desc of abc"}

$$Key(pair1) = ord(a) + ord(b) + ord(c) = 0 + 1 + 2 = 3$$

the proposed data structure is as follows:

- create a static table with a predefined length
- the content of each cell table is a LinkedList
- the linked list is about a Pair class object
- a Pair object with the two properties identification and description

in this project, we create all required methods to implement this data structure :

- ⋆ add an element to the table.
- ⋆ delete an element from the table.
- ⋆ print the whole table.
- ⋆ checking if an object exists in the table.
- ⋆ getting the number of empty slots.

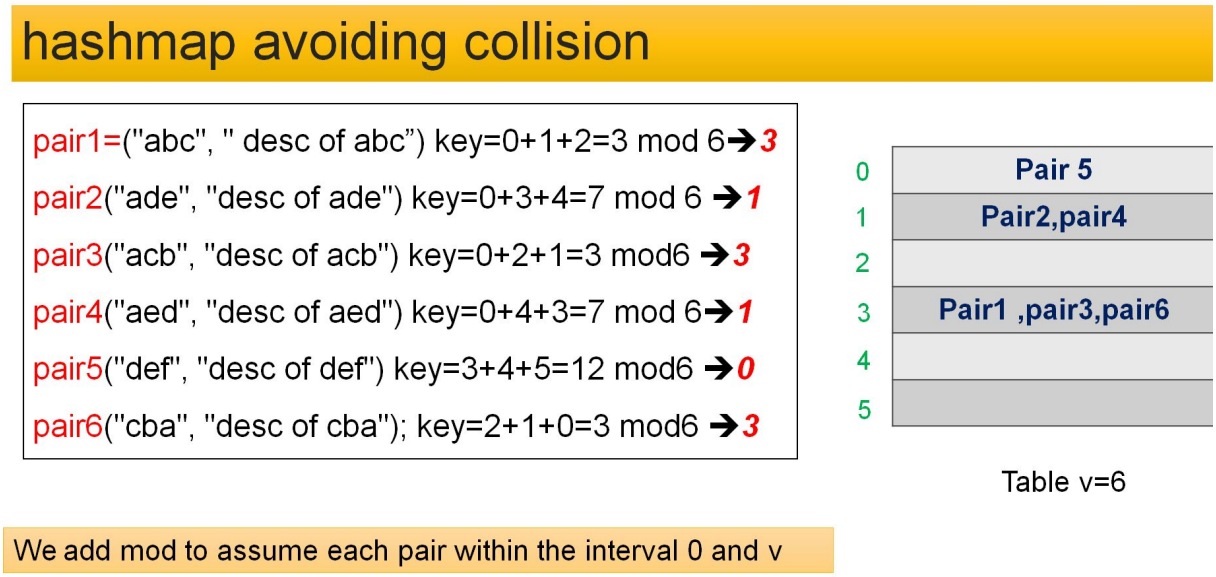

Figure 1: distributing the pair objects in the table according to the key

3 Implementation Technique Using Java

the following table shows the role of each class

The Class	The Role
interface HashMap	all abstract methods have to implement
class ServiceHashMap	this class implement all methods of the interface HashMap
class Pair	a pair object class with two properties id and desc
class ExceptionPairNotExist	throws an exception if a pair doesn't exist In the table
class ExceptionPairExist	throws an exception if a pair exist before In the table
class Main	for the testing purposes

Table 1: the role and significance of each class project

3.1 interface HashMap source code

```
public interface HashMap {
    void addElement(Pair pair);
    void deleteElement(Pair pair);
    Boolean isExist(Pair pair);
    void printHashMap();
    //for printing the whole table
    int getEmptySlot();
}
```

3.2 class Pair source code

```
public class Pair {
     private String identification;
     private String description;

    public Pair(String identification, String description) {
        this.identification = identification;
        this.description = description;
    }

    public String getIdentification() {
        return identification;
    }

    public String getDescription() {
        return description;
    }

    @Override
    public String toString() {
        return "{" +
                "ident='" + identification + '\'' +
                ", desc='" + description + '\'' +
                '}';
    }
}
```

3.3 class ServiceHashMap source code

```
import exceptions.ExceptionPairExist;
import exceptions.ExceptionPairNotExist;
import java.util.LinkedList;


public class ServiceHashMap implements HashMap {
    private Integer maxValue;
    //max value table length
    LinkedList<Pair>[] table;
    //table contains linked-list data structure  of Pair class

    public ServiceHashMap(Integer maxValue) {
        this.maxValue = maxValue;
        table = new LinkedList[maxValue];
    }

    @Override
    public void addElement(Pair pair) {
```

```
    if (isExist(pair))
        //we don't add a pair exists before
        throw new ExceptionPairExist("this pair exist in the table");
    int key = hashValue(pair.getIdentification());
    //get key of pair using its identification
    if (table[key] == null) {
        // if the object of linked-list pair doesn't exist
        LinkedList linkedList = new LinkedList();
        linkedList.add(pair);
        table[key] = linkedList;
    } else {
        // the object of linked-list was created before
        // we add just the pair object to the list
        table[key].add(pair);
    }

}

@Override
public void deleteElement(Pair pair) {
    if (!isExist(pair))
        throw new ExceptionPairNotExist(" pair not exist in the table");
    int key = hashValue(pair.getIdentification());
    LinkedList<Pair> linkedList = table[key];
    for (int i = 0; i < linkedList.size(); i++) {
        if (linkedList.get(i).getDescription()
                .equalsIgnoreCase(pair.getDescription()))
            linkedList.remove(i);
    }

}

@Override
public Boolean isExist(Pair pair) {
    // checking from the key and the description pair if exists
    int key = hashValue(pair.getIdentification());
    if (table[key] == null)
        return false;
        //if the key doesn't exist
    LinkedList<Pair> linkedList = table[key];
    for (int i = 0; i < linkedList.size(); i++) {
        if (linkedList.get(i).getDescription()
                .equalsIgnoreCase(pair.getDescription()))
            return true;
        //if the pair exist (means the key and the description)
    }
    return false;
}
```

```
    @Override
    public void printHashMap() {
        for (int i = 0; i < table.length; i++) {
            // printing just the occupied value of table
            if (table[i] != null)
                System.out.println("index " + i + " " + table[i]);
        }
    }

    @Override
    public int getEmptySlot() {
       // counting the empty slot of the table
        int count=maxValue;
        for (LinkedList<Pair>p:table) {
            if(p!=null )
                count--;

        }
   return count; }

    private Integer hashValue(String identification) {
        int order_a = 'a';
        //get the ASCII code of small letter a
        int sum = 0;
        String lowerCase = identification.toLowerCase();
        for (char ch : lowerCase.toCharArray()) {
            sum += ch - order_a;
           // summing the order of each letter
           // starting with order a which is 0
        }
        return sum % maxValue;
        //assuming each pair within the interval 0 and max-value
    }
}
```

3.4 class ExceptionPairNotExist source code

```
package exceptions;
public class ExceptionPairNotExist extends RuntimeException{
    public ExceptionPairNotExist(String message) {
        super(message);
    }
}
```

3.5 class ExceptionPairExist source code

```
package exceptions;

public class ExceptionPairExist extends RuntimeException{
    public ExceptionPairExist(String message) {
        super(message);
    }
}
```

3.6 class Main source code

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Pair> listPair = List.of(
                new Pair("abc", "desc of abc"),
                new Pair("ade", "desc of ade"),
                new Pair("acb", "desc of acb"),
                new Pair("aed", "desc of aed"),
                new Pair("def", "desc of def"),
                new Pair("cba", "desc of cba"));

        int maxValue = 6;
        ServiceHashMap serviceHashMap = new ServiceHashMap(maxValue);
        for (int i = 0; i < listPair.size(); i++) {
            serviceHashMap.addElement(listPair.get(i));
        }
        System.out.println("\n\t\t___result after adding elements____");
        serviceHashMap.printHashMap();

        Pair pairToDelete=listPair.get(2);
        serviceHashMap.deleteElement(pairToDelete);
        System.out.println("\n\t\t___ deleting a pair element "+pairToDelete);
        serviceHashMap.printHashMap();

        int getEmptySlot= serviceHashMap.getEmptySlot();
        System.out.println("\n\t\t___ get empty slot "+getEmptySlot);

    }
}
```

3.7 Result output

```
___result after adding elements____
index 0 [{ident='def', desc='desc of def'}]
index 1 [{ident='ade', desc='desc of ade'}, {ident='aed', desc='desc of aed'}]
index 3 [{ident='abc', desc='desc of abc'}, {ident='acb', desc='desc of acb'}
, {ident='cba', desc='desc of cba'}]

___ deleting a pair element {ident='acb', desc='desc of acb'}
index 0 [{ident='def', desc='desc of def'}]
index 1 [{ident='ade', desc='desc of ade'}, {ident='aed', desc='desc of aed'}]
index 3 [{ident='abc', desc='desc of abc'}, {ident='cba', desc='desc of cba'}]

___ get empty slot 3

Process finished with exit code 0
```

4 Time Complexity

1. Time complexity to add a pair object is O(1) (keeping track of the last cell of the linked list)
2. Time complexity to search an object in worst cases is O(n) (length of the linked list)
3. Time complexity to delete an object in worst cases is O(n) (length of the linked list) because we need to find the pair object before deleting.

5 Refrences

⇒ the whole URL java project github.

⇒ video youtube URL explanation.