Perfect Hashing

Names :

Ahmed Reda Amin ()

Ahmed Ezzat Elmaghawry (11)

Arsanuos Essa Attia(18)

Problem statement :

you’re required to implement a perfect hashing data structure. We say ahash function is perfect for S if all lookups involve O(1) work.
In section 2, background about universal hashing is provided. Sections 3 and 4 describe twomethods for constructing perfect hash functions for a given set S. You’re requied to design,analyze and implement a perfect hash table as described in sections 3 and 4.

Implementation details:

Order N

1- Constructing array of Size N of pairs of Hashed Integers and number of collisions
2- Hash all elements and if a collision happened then rehash the small array in the pair using another hash function.
3- if the number of collisions exceeds N2 Rehash using another hash function using the universal hash function which implemented.

Order N^2

1- Constructing array of Size N2 of pairs of Hashed Integers and number of collisions
2- Hash all elements and if a collision occurred rehash all.

Code snippet :

The constructor of the main Class :

```
public PerfectHash(int[] array, boolean flag) {
            u = 0;
            hashFunction = new HashFunction();
            this.array = array;
            inputArray = new String[array.length];
            findU();
            findB(flag);
            list = new LinearHashingPair[m];
      }
```

To get The U :

```
private void findU() {
            for (int i = 0; i < array.length; i++) {
                  String temp = Integer.toBinaryString(array[i]);
                  inputArray[i] = temp;
                  int len = temp.length();
                  if (u < len) {
                        u = len;
                  }
            }
      }
```

To get b :

```
private void findB(boolean flag) {
            if (flag == true) {
                  // order n
                  // assume m = n
                  // 2^b = 2 *n
                  // b = 2 * log(n,2) base 2 of course.
                  m = (int) (2 * array.length);
                  b = log(m, 2);
                  intializeList();
                  orderN = true;
            } else {
                  // order 2 * n^2
                  m = (int) (array.length * array.length);
                  b = log(m, 2);// 2 * 2
                  result2 = new int[m];// chain of 5;
                  Arrays.fill(result2, -1);
                  orderN = false;
            }
      }
```

Generating the hash function :

```
private void constructArrayOfSizrBU() {
            mainHashFunction = hashFunction.getHashFunction(b, u, 0);
      }
```

Hashing Function :

```
public int hash() {
            // choose hash function
            constructArrayOfSizrBU();
            // hash
            boolean collision = hashAll();
            int count = 1;
            while (collision) {
                  if (count > 3) {
                        // throw new RuntimeException();
                  }
                  clearAll();
                  constructArrayOfSizrBU();
                  collision = hashAll();
                  count++;
                  // System.out.println(count);
            }
            // if collision repeat and count number of repetition.
      System.out.println(count);
            return count;
      }
```

Search of the hashing table :

```
public boolean search(int num) {
            if (orderN) {
                  String binary = completeSizeU(Integer.toBinaryString(num));
                  if (binary.length() <= u) {
                        int index = multiply(hashFunction.getHashFunction(b,
u, 0), binary);
                        if (index < list.length) {
                              LinearHashingPair l = list[index];
                              if (l != null) {
                                    int[] array = l.num;
                                    if (array.length == 1) {
                                          if (array[0] == num) {
                                                return true;
                                          }
                                    } else {
                                          // for second level hashing
                                          int index2 =
multiply(hashFunction.getHashFunction(log(l.num.length, 2), u, index),
binary);
                                          if (index2 < array.length &&
array[index2] == num) {
                                                return true;
                                          }
                                    }
```

```
                    }
                }
            }
            return false;
        } else {
            String binary = completeSizeU(Integer.toBinaryString(num));
            if (binary.length() <= u) {
                int index = multiply(hashFunction.getHashFunction(b,
u, 0), binary);
                if (index < result2.length && result2[index] == num)
{
                    return true;
                }
            }
            return false;
        }
    }
```

Multiplying the matrix :

```
private int multiply(int[][] hashFunction, String binaryOfNumber) {
        StringBuilder out = new StringBuilder(hashFunction.length);
        for (int i = 0; i < hashFunction.length; i++) {
            int num = 0;
            for (int j = 0; j < binaryOfNumber.length(); j++) {
                if (binaryOfNumber.charAt(j) == '0') {
                    num += 0;
                } else {
                    num += hashFunction[i][j] * 1;
                }
                if (num >= 2) {
                    num %= 2;
                }
            }
            num = num % 2;
            out.append(num);
        }
        return Integer.parseUnsignedInt(out.toString(), 2);
    }
```

Hash function generator :

```
private int[][] generateHashFuction(int b, int u, int index) {
        // index of the row required to be hashed and it will be zero if
it was
        // the main hashFunction.
        int[][] hashFunction = new int[b][u];
        Random r = new Random();

        for (int i = 0; i < b; i++) {
            for (int j = 0; j < u; j++) {
                // number will be even or odd so
                // taking mod 2 will make it equal 0 or 1
                int num = r.nextInt(10) % 2;
                hashFunction[i][j] = num;
            }
        }
        hashFunctions.put(index, hashFunction);
        return hashFunction;
    }
```

Note : We assume that the hashing didn’t accept only the -1 where we initially fill the first building array with -1 where if we compare the key with it and the key was -1 will say that there is a collision although there isn’t .