LAB 4 REPORT (Perfect Hash Map)

Members :

Basel Ahmed Awad Ayad	19015513
Ali Hassan El Sharawy	19016013

Table of contents:

Problem Statement:

Implementation of a perfect hashing data structure. We say a hash function is perfect for S if all lookups involve O(1) work.

There are two methods for constructing perfect hash functions for a given set of keys S.

O(N^2)-Space Solution:

In this method the size of the hash table is quadratic in the size N of our dictionary S. Then, here is an easy method. Let H be universal and M= N^2. Pick a random h from H and try it out, hashing everything in S. So, we just try it, and if we have any collisions, we just try a new h. On average, we will only need to do this twice.

O(N)-Space Solution:

The main idea for this method is to use universal hash functions in a 2-level scheme.The method is as follows. We will first hash into a table of size N using universal hashing. This will produce some collisions. However, we will then rehash each bin using Method 1, squaring the size of the bin to get zero collisions. So, the way to think of this scheme is that we have a first-level hash function h and first-level table A, and then N second-level hash functions h1, ..., hN and N second-level tables A1, ..., AN . To look up an element x, we first compute i = h(x) and then find the element in Ai[hi(x)].

Data Structure Used:

- Array
- Set
- List
- MapEntry:
  Class containing two attributes key and corresponding value with setters and getters

```
public class MapEntry {
   private int key;
   private int value;

   public MapEntry(int key) {
       this.key = key;
       this.value = 0;
   }

   public MapEntry(int key, int value) {
       this.key = key;
       this.value = value;
   }
```

Important Algorithm:

This function is used to generate random function to hash key to the table

```
public int[][] get_random_hash_fn(int b, int u) {
   int[][] random_hash_fn = new int[b][u];
   Random random = new Random();
   for (int i = 0; i < b; i++) {
       for (int j = 0; j < u; j++) {
           random_hash_fn[i][j] = (random.nextInt() % 2 + 2) % 2;
       }
   }
   return random_hash_fn;
}
```

For $O(N^2)$ Solution:

hash function multiply key and random hash function to get hash value to insert element to hash table.

```
protected int hash(int key, int[][] hash_fn) {
   int[] key_binary = util.decompose_into_n_bits(key, 32);
   int[] hash_value = new int[hash_fn.length];
   for (int i = 0; i < hash_fn.length; i++) {
       for (int j = 0; j < hash_fn[0].length; j++) {
           hash_value[i] += hash_fn[i][j] * key_binary[j];
       }
       hash_value[i] %= 2;
   }
   return util.decimal_value(hash_value);
}
```

We hash using a quadratic space $O(N^2)$ solution by generating a random function for all elements and store it for setting & retrieving elements.
We loop on the set of keys and calculate the hash value for it.
After calculating hash value we check for collision if there is an element in this position to insert value to this position.
If there is an element in this position then collision happens and we use recursion to rehash all elements and generate new random functions.
We repeat till no collision happens.

```
protected List<MapEntry> hash_using_quadratic_space_sol(Set<Integer> setOfKeys){
   int table_size = (int) Math.pow(setOfKeys.size(), 2);
   MapEntry[] hash_table = new MapEntry[table_size];
   int b = (int) (Math.log(table_size) / Math.log(2));
   hashFn = util.get_random_hash_fn(b, 32);
   for (int key : setOfKeys) {
       int hash_value = hash(key, hashFn);
       if (hash_table[hash_value] != null && hash_table[hash_value].getKey()!=0)
           return hash_using_quadratic_space_sol(setOfKeys);
       hash_table[hash_value] = new MapEntry(key);
   }
   return Arrays.stream(hash_table).toList();
}
```

We store random hash function used to hash all elements to be able to set and get values by calculating hash value.

```
public int getValue(int key) {
  int index = hash(key, hashFn);
  return quadHashTable.get(index).getValue();
}

public void setValue(int key, int value) {
   int index = hash(key, hashFn);
   quadHashTable.get(index).setValue(value);
}
```

For O(N) Solution:

We store an array of random hash functions as each slot has its own random function in level two hashing beside the hash function of level one hashing at index zero.

For level one hashing we generate a random hash function and store it at the beginning of the hash function array.

We loop on the set of keys and calculate the hash value of each key.

We check if this position is empty then we generate a new set to store all keys at these slots.

Else we add this key to the existing set.

```
private void level1_hashing() {
   int b = (int) (Math.log(setOfKeys.size()) / Math.log(2));
   int[][] level1_hash_fn = util.get_random_hash_fn(b, 32);
   listHashFn[0] = level1_hash_fn;
   for (int key : setOfKeys) {
       int hash_value = hash(key, level1_hash_fn);
       if (level1HashTable.get(hash_value).isEmpty())
           level1HashTable.set(hash_value, new HashSet<>());
       level1HashTable.get(hash_value).add(key);
   }
}
```

After level one hashing we have set at each slot containing keys stored at this position then we have level two hashing.

We loop on a set of keys and get the number of keys in the same slot and update total size by slot size squared.
If there are no elements in the slot we just update the hash function array and skip this loop.
Else we use the hashing function of quadratic hashing in level two hashing and store the hash function used for this slot.

```
private void level2_hashing() {
    int slotSize = 0;
    for (int i = 0; i < setOfKeys.size(); i++) {
        slotSize = level1HashTable.get(i).size();
        totalSpace += Math.pow(slotSize, 2);
        if (slotSize == 0) {
            listHashFn[i + 1] = new int[0][32];
            continue;
        }
        if (linearHashTable.get(i).isEmpty())
            linearHashTable.set(i, new ArrayList<>());
        linearHashTable.set(i,hash_using_quadratic_space_sol(level1HashTable.get(i)));
        listHashFn[i + 1] = hashFn;
    }
}
```

For setters and getters we get the slot index using a random hash function of level one hashing at index zero of the hash function array.
Using slot index we get a hash function for this slot to be able to set/get value in the hash table.

```
public int getValue(int key) {
   int slotIndex = hash(key, listHashFn[0]);
   int level2Index = hash(key, listHashFn[slotIndex + 1]);
   return linearHashTable.get(slotIndex).get(level2Index).getValue();
}
```

```
public void setValue(int key, int value) {
   int slotIndex = hash(key, listHashFn[0]);
   int level2Index = hash(key, listHashFn[slotIndex + 1]);
   LinearHashTable.get(slotIndex).get(level2Index).setValue(value);
}
```

Assumptions:

The default value corresponding to the key in the map entry is zero

Sample Runs:

Sample runs verifying amount of space allocated for hash table:

```
Total no of keys : 10
Quad space : 100
No of times to rebuild Hash Table = 0
-----------------------------------------
Linear space : 16
No of times to rebuild Hash Tables = 0

Total no of keys : 100
Quad space : 10000
No of times to rebuild Hash Table = 1
-----------------------------------------
Linear space : 262
No of times to rebuild Hash Tables = 12

Total no of keys : 1000
Quad space : 1000000
No of times to rebuild Hash Table = 2
-----------------------------------------
Linear space : 2944
No of times to rebuild Hash Tables = 148

Total no of keys : 10000
Quad space : 100000000
No of times to rebuild Hash Table = 1
-----------------------------------------
Linear space : 22272
No of times to rebuild Hash Tables = 1162
```

Sample runs to verify that avg access time for any key is O(1) :

```
Total no of keys : 18
Default value of key 48414445
48414445 -> 0
After setting value of key 48414445
48414445 -> 18484844
Time to look up for key : 6 ms
```

```
Total no of keys : 108
Default value of key 48414445
48414445 -> 0
After setting value of key 48414445
48414445 -> 18484844
Time to look up for key : 7 ms
```

```
Total no of keys : 1008
Default value of key 48414445
48414445 -> 0
After setting value of key 48414445
48414445 -> 18484844
Time to look up for key : 5 ms
```

```
Total no of keys : 10008
Default value of key 48414445
48414445 -> 0
After setting value of key 48414445
48414445 -> 18484844
Time to look up for key : 4 ms
```