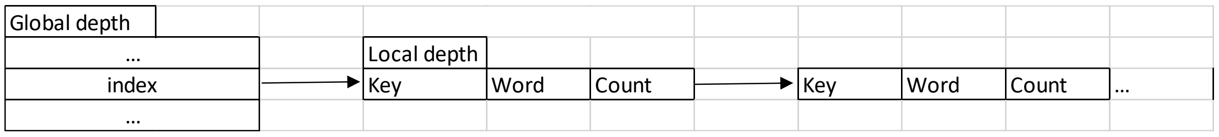
In this assignment, you are expected to implement an **extendible hashing** to index words of a document named as ‘cartoon.txt’ in Java. You must read this document, split it word by word, and index each word to your hash table according to rules given below.

## RULES

- Usage of Abstract Data Types (ADT) is required.
- Object Oriented Principles (OOP) and try-catch exception handling must be used when it is needed.
- Hash function for converting words (value) to a key must be implemented by yourself. The value will be the word and the key will be returned by your hash function. The number of occurrence of each word also must be stored as *count* value.
- The hash code for converting a key to the bit representation must be implemented by yourself.
- The initial table should index a given key according to the last 8 bits (initial global depth and local depths will be 8). Every slot of the table points a bucket. The bucket size is 10.
- When you need to resize the hash table, the new table size must be doubled to index words by using one more bit (global depth+1). The bucket size won’t change. E.g., After the first resize operation, the table should index according to the last 9 bits. After the second resize, it should index according to 10 bits etc.
- When a word is searched in the hash table; key, count, and index of the word should be printed. (Global and local depths should also be printed)

**The Hash table should look like as below:**



The standard output of your program should appear as follows:

```
------ Extendible Hashing ------
Search: Ali
Key: 1243225
Count : 10
Index : 0001011001
Global depth: 10
Local depth: 9

Search: Mehmet
Key: 68294842
Count : 3
Index : 0010111010
Global depth: 10
Local depth: 10
```

**Note**: Results of the words of “Ali” and “Mehmet” were obtained from other documents. You may not obtain the same results by using the ‘cartoon.txt’ file.

**Main Functionalities:**

- **INSERT:** Read the given input file, calculate the number of occurrence of each word, insert this information into the hash table accordingly.
- **SEARCH:** Search a user input (word) in the hash table. If the input is available in the table return an output as shown above, otherwise return a "not found" message to the user.

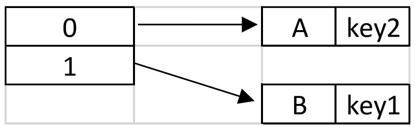
## EXTENDIBLE HASHING

Assume that the hash code function hc(key) returns a string of bits. The last *i* bits of each string will be used as indices to figure out where they will go in the "directory" (hash table). Additionally, *i* is the smallest number such that the index of every item in the table is unique.

Example keys:

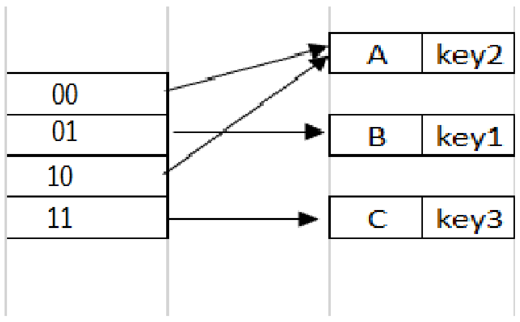
hc(key1) = 001001
hc(key2) = 011010
hc(key3) = 011011

Let's assume that for this example, the bucket size is 1. key1 and key2 can be distinguished by their last bit; they will be inserted into the table as follows:



**Note:** A and B are only names of buckets. For next examples C, D, and E will be also bucket names. You don't have to put a name to a bucket in your codes.

When key3 is to be hashed to the table, it wouldn't be enough to distinguish all three keys by using only their last bit (because both key3 and key1 have 1 as their rightmost bit). Furthermore, due to the bucket size being one, the table would overflow. Comparison of two less significant bits (rightmost bits) would give each key a unique index in the table, and the directory size is doubled as follows:



Key1 and key3 have a unique location, being distinguished by the first two rightmost bits. Key2 is pointed by both 00 and 10, because there is no other key in the table which ends with a 0.

To insert key4 into the table, first compute the hash function as below:

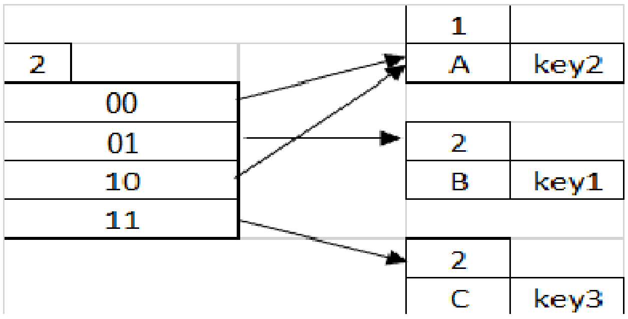
hc(key4) = 011110

Key4 needs to be inserted in the table, hash function returns the last two bits as 10. Using a 2 bits depth in the directory, key4 will map to index 10 and corresponding to the Bucket A. Bucket A is already full (due to max size =1), so it must be splitted. Because there is more than one pointer to the Bucket A, there is no need to increase the directory size. To perform such operations, some rules are defined:

-The hash function returns binary representation of a given key (*bk*). The last N bits of *bk* shows the directory (table) size (i.e., the global depth).

-The previously used the global depth value of a bucket is created shows the local depth of this bucket.

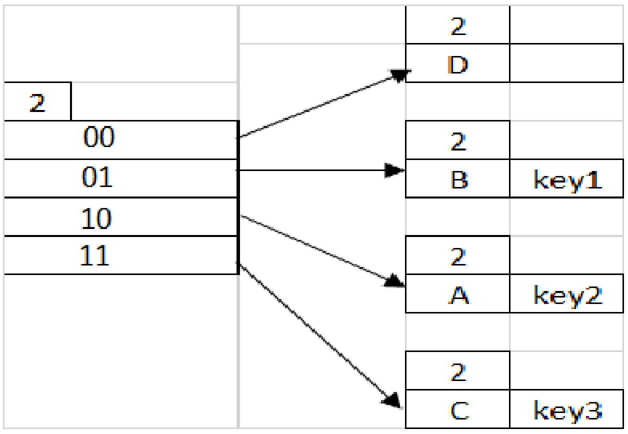
- The number of directory entries is equal to $2^{\text{global depth}}$, and the initial number of buckets is equal to $2^{\text{local depth}}$.



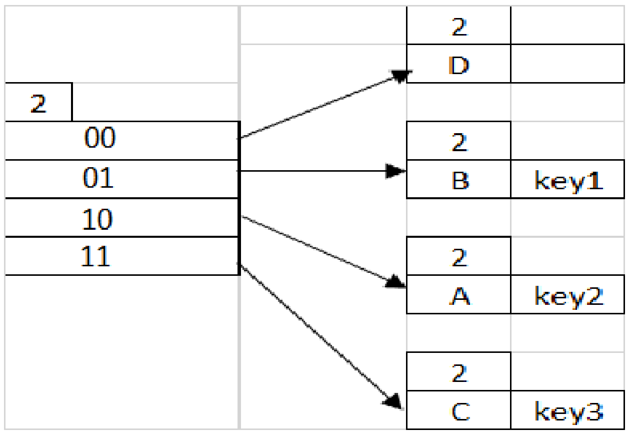
When a bucket becomes full:

a) If (global depth - local depth) = 0, then $2^0 = 1$, one pointer shows to one bucket. The directory size will be doubled.

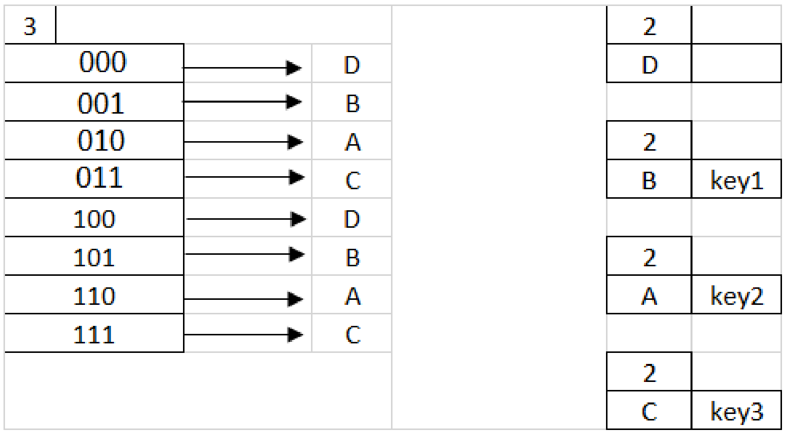
b) If (global depth - local depth) > 0, a new bucket will be created, and the entries in the old bucket will re-distributed between the old and new bucket.



Index 10 points to Bucket A, and Bucket A's local depth of 1, which is less than the directory's global depth of 2, that means keys hashed to Bucket A only use a 1 bit postfix (i.e., 0). Key4 will be indexed to 10 and Bucket A, however Bucket A is full. A new bucket should be created and pointed by index 00, old Bucket A is pointed by 10. Key2 in re-hashed to the table, it will be indexed to 10. Index of Key4 is computed by hash function, it is indexed to 10 as well. The difference of global and local depth is 0 (global depth - local depth=0), and the index 10 shows a full bucket A; so the directory (table) size is doubled.

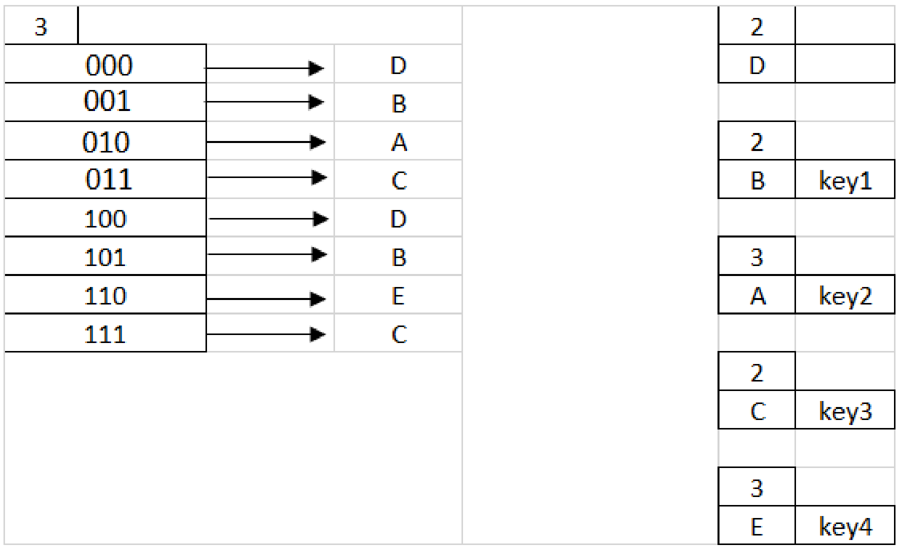


The double directory size will contain 8 slots due to using last 3 bits in the hash function.



After the directory size is doubled, Key4 will be inserted to the directory by using the following steps:

1. The last 3 digits of Key4 are 110, so it will show the Bucket A.
2. Global depth > Local depth(A), so bucket A needs to be splitted due to being full.
3. Due to the split operation of bucket A, which becomes bucket A and bucket E. The local depth of both buckets will be 3.
4. The entries (key2) in the old bucket A has been re-hashed by using the last 3 bits, and they will be saved in the new bucket A.
5. Key4 is re-hashed by using the last 3 bits and it ends up in bucket E which has a spare slot.



Note that, the bucket size was assigned to 1 in this example, BUT bucket size must be 10 in your assignment.