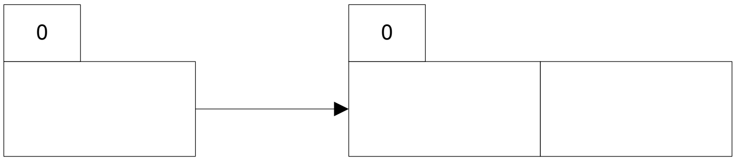
# Extendible Hashing – In-class Example

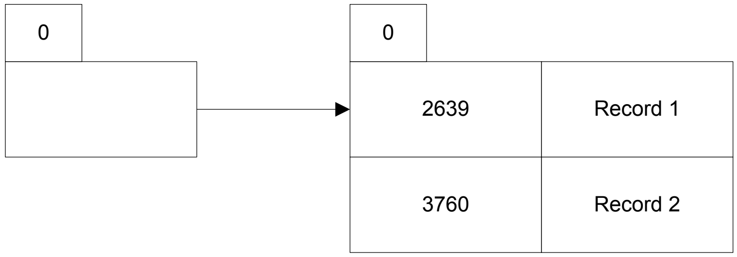
Below is a set of records we are going to insert into a hash table using extendible hashing. The Record column contains a pointer to the data record; K is the search key value. H(K) is the result of running K through our hashing algorithm, shown in decimal and bits.

| Record | K | H(K) | Bits of H(K) |
|---|---|---|---|
| Record 1 | 2639 | 1 | 00001 |
| Record 2 | 3760 | 16 | 10000 |
| Record 3 | 4692 | 20 | 10100 |
| Record 4 | 4871 | 7 | 00111 |
| Record 5 | 5659 | 27 | 11011 |
| Record 6 | 1821 | 29 | 11101 |
| Record 7 | 1074 | 18 | 10010 |
| Record 8 | 2115 | 11 | 01011 |
| Record 9 | 1620 | 20 | 10100 |

We start with the hash table structure initialized to the following:

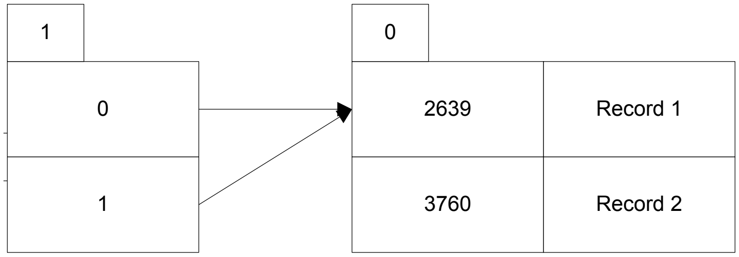


Note the global depth is set to zero to start. For this example, we are assuming that we are using alternative 2 and only 2 k* entries exist per bucket, – in reality the numbers are much larger. The first two entries fit into the bucket and we look like this:

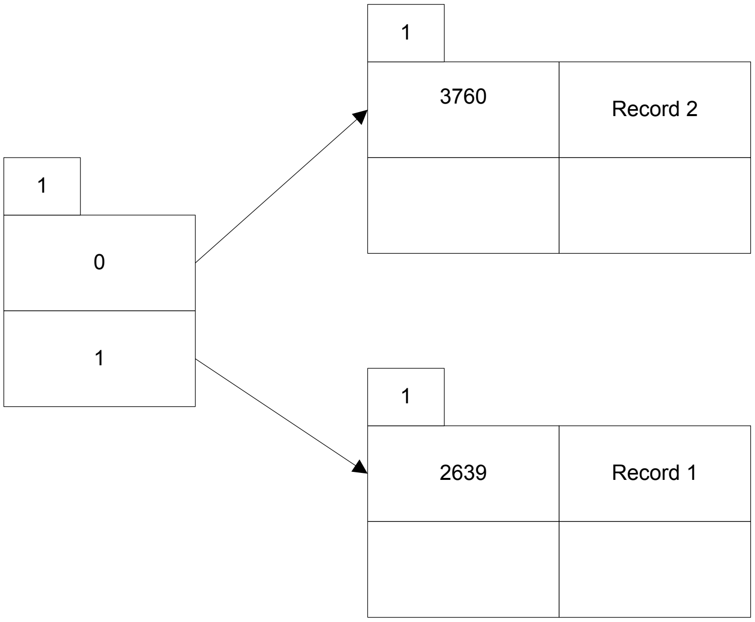


When we attempt to insert the next index, the bucket is full, so we need to increase the local depth. Since before splitting the bucket, the local depth is equal to the global depth, we must also double the directory.

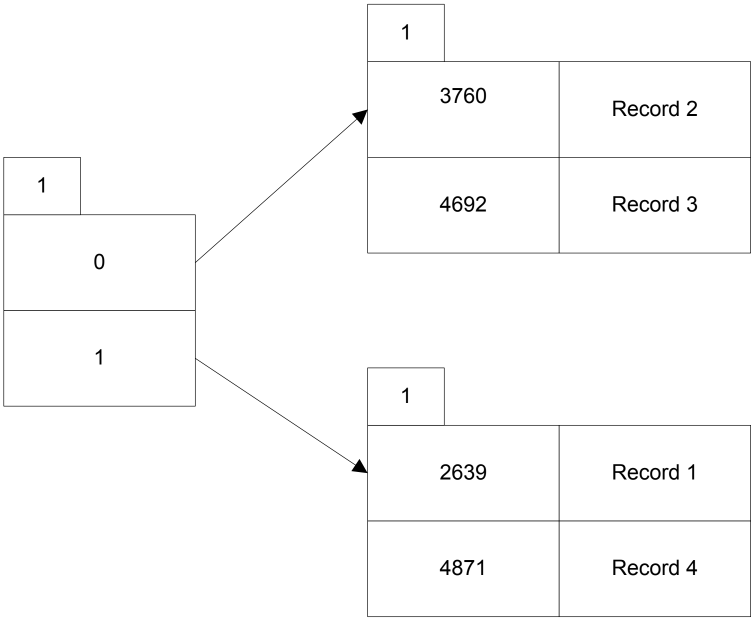
We start by doubling the directory. This is done by doing a memcopy() of the existing directory onto the end of the existing directory.



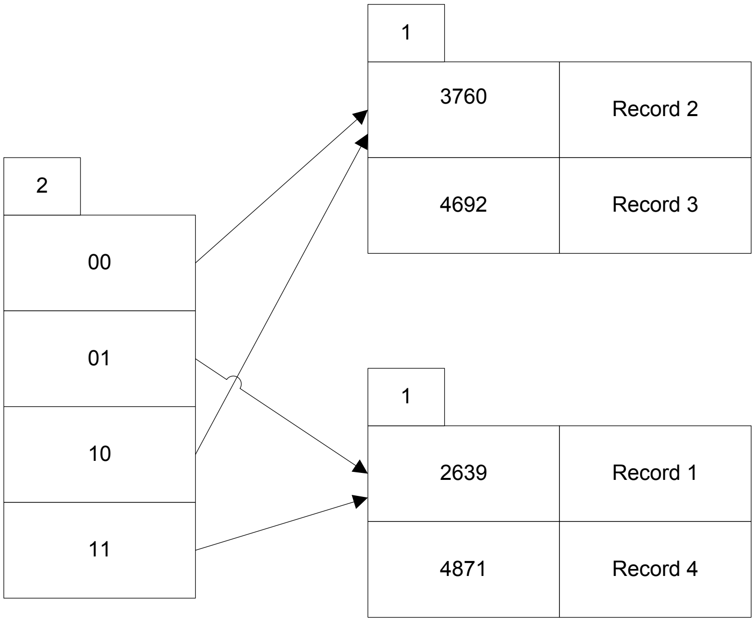
Now that we have doubled the directory, we can split the bucket. We allocate a new bucket, and rehash the existing values. We use the first bit of the hash result of each to place them in the appropriate bucket.



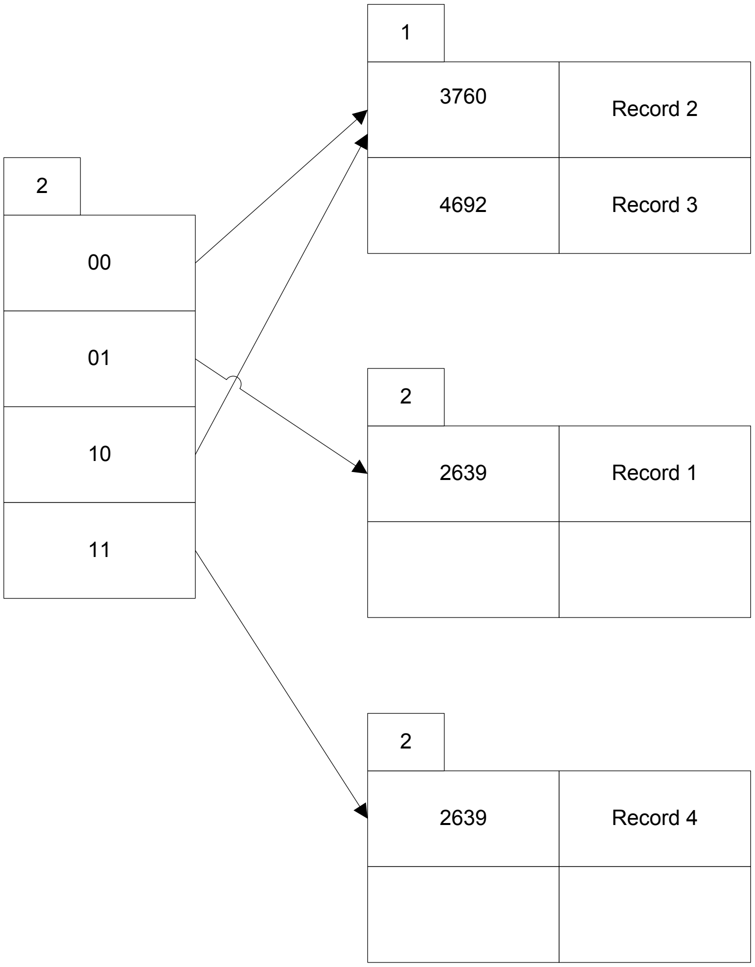
Placing Record 3's data entry is the same process. The least significant bit is a 0, so it is placed in the hash bucket pointed to by the 0th entry. Record 4's least significant bit is a 1 – there is an open slot in that bucket – so it is placed without issue.



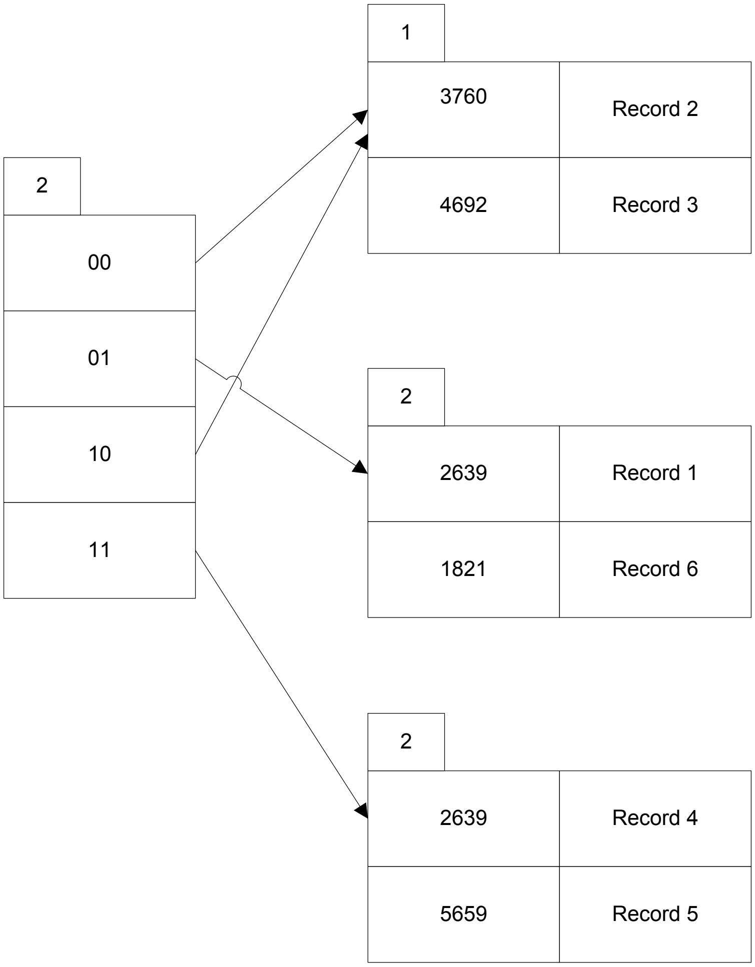
When we attempt to place Record 5, the LSB is a 1, and the bucket pointed to by 1 is full. The local depth is equal to the global depth, so we must also double the size of the directory. After the memcopy and updating the global depth, our structure looks like this:



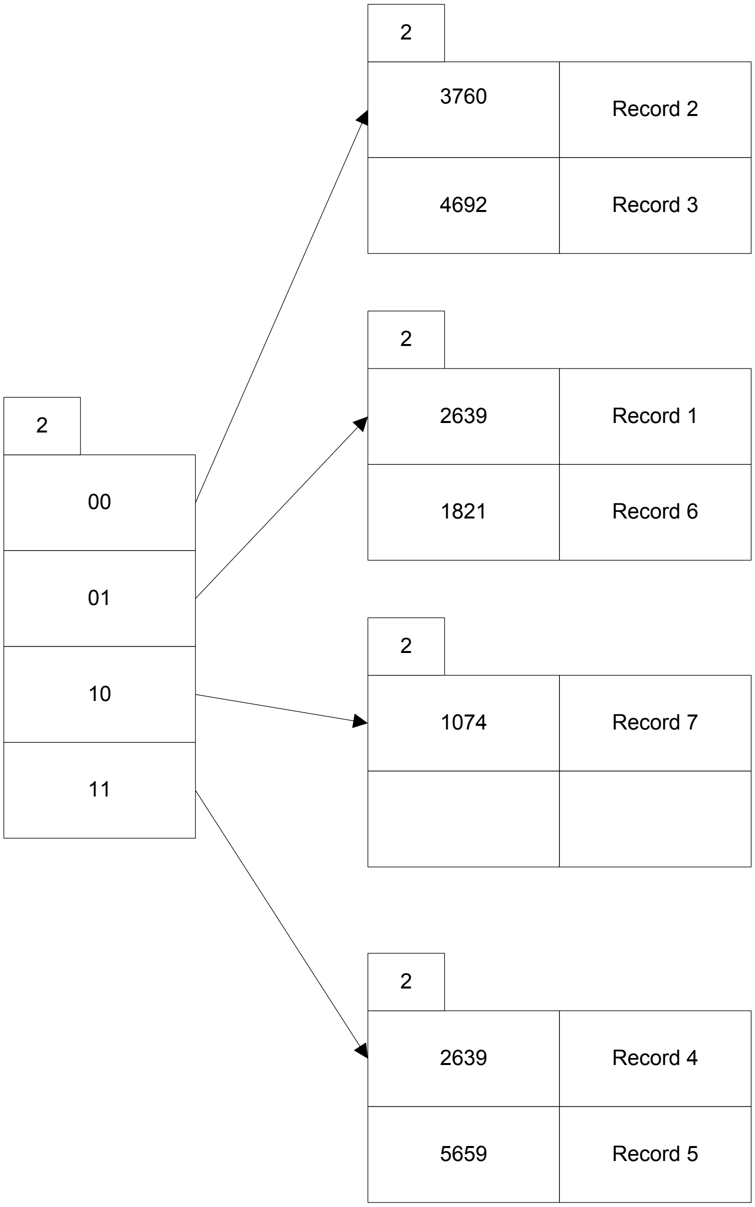
Since our local depth is now less than our global depth, we can split the node and rehash the values. We need to now use the 2 least significant bits (designated $LCB_2$) when we rehash. The result looks like this:



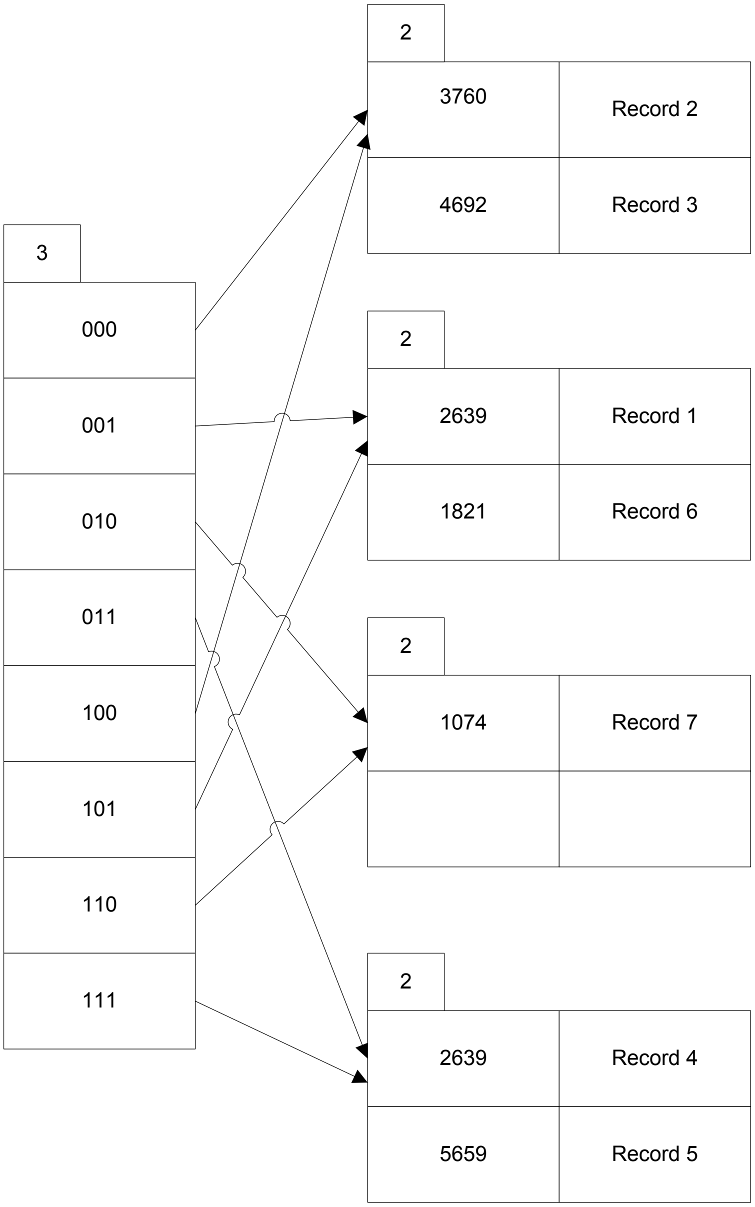
Adding the data entries for Records 5 and 6 give us the following using $LCB_2$ of their hash result:



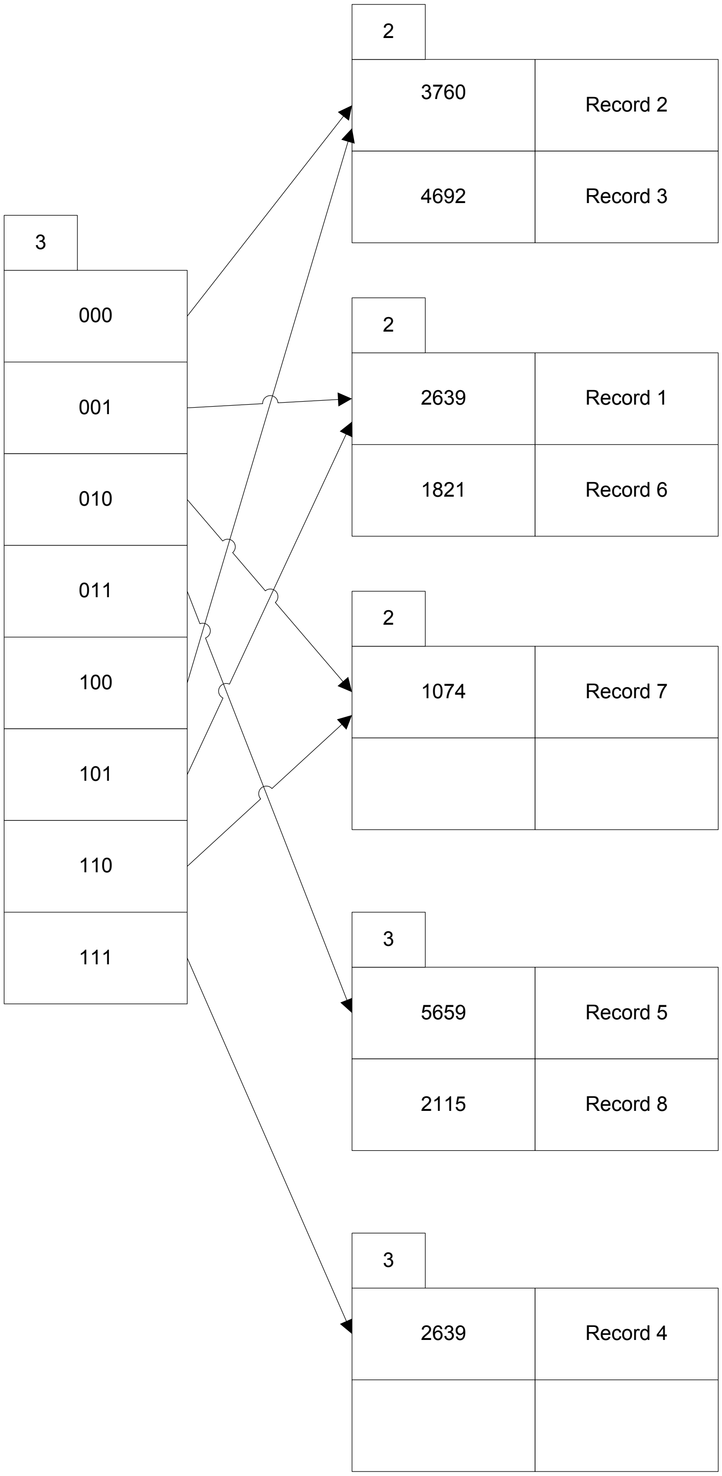
When we add Record 7, the bucket pointed to by 10 is full, but the local depth is less than the global depth.  Because of this, we do not have to double the directory, and we proceed with splitting the bucket and rehashing the entries.  After adding the data entry for record 7, we look like this:



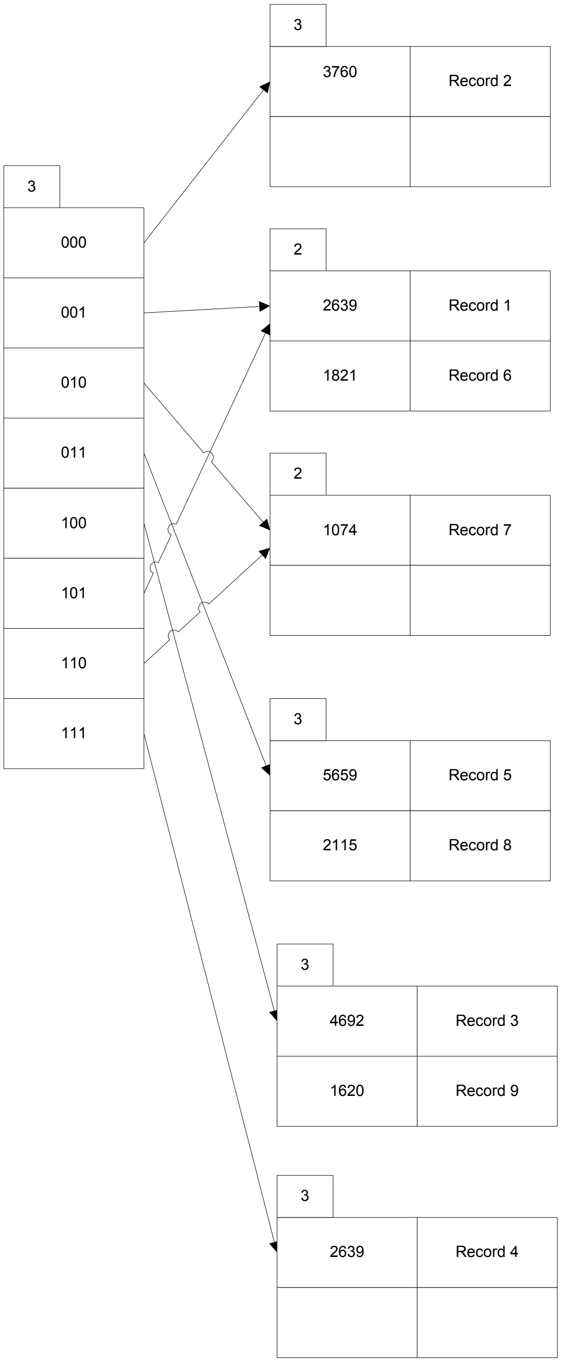
Record 8 hashes to a 11, the bucket pointed to by 11 is full.  The local depth is equal to the global depth – so we must start by doubling the directory.  That results in the following:


3
000
001
010
011
100
101
110
111
2
3760
Record 2
4692
Record 3
2
2639
Record 1
1821
Record 6
2
1074
Record 7
2
2639
Record 4
5659
Record 5

We can split the bucket pointed to by 11, rehash the existing values, and add the data entry for record 8 giving us:



Now that our global depth is 3, we must use the 3 least significant bits (LCB$_3$) of our hash result. Record 9 hashes to 100, whose bucket is full, and the local depth is less than the global depth, we split the bucket, rehash the values, and add the data entry for record 9 giving us:


3
000
001
010
011
100
101
110
111
3
3760
Record 2
2
2639
Record 1
1821
Record 6
2
1074
Record 7
3
5659
Record 5
2115
Record 8
3
4692
Record 3
1620
Record 9
3
2639
Record 4