Extendible hashing

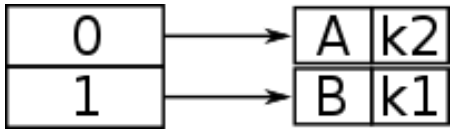
Extendible hashing is a type of hash system which treats a hash as a bit string and uses a trie for bucket lookup. Because of the hierarchical nature of the system, re-hashing is an incremental operation (done one bucket at a time, as needed). This means that time-sensitive applications are less affected by table growth than by standard full-table rehashes.

Assume that the hash function h(k) returns a string of bits. The first 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.

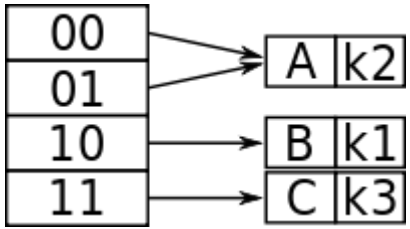
Keys to be used:

$$h(k_1) = 100100$$
$$h(k_2) = 010110$$
$$h(k_3) = 110110$$

Let's assume that for this particular example, the bucket size is 1. The first two keys to be inserted, k_1 and k_2, can be distinguished by the most significant bit, and would be inserted into the table as follows:



Now, if k_3 were to be hashed to the table, it wouldn't be enough to distinguish all three keys by one bit (because both k_3 and k_1 have 1 as their leftmost bit). Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:



And so now k_1 and k_3 have a unique location, being distinguished by the first two leftmost bits. Because k_2 is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.

$$h(k_4) = 011110$$

Now, k_4 needs to be inserted, and it has the first two bits as 01..(1110), and using a 2 bit depth in the directory, this maps from 01 to Bucket A. Bucket A is full (max size 1), so it must be split; because there is more than one pointer to Bucket A, there is no need to increase the directory size.

What is needed is information about:

1. The key size that maps the directory (the global depth), and
2. The key size that has previously mapped the bucket (the local depth)

In order to distinguish the two action cases:

1. Doubling the directory when a bucket becomes full
2. Creating a new bucket, and re-distributing the entries between the old and the new bucket

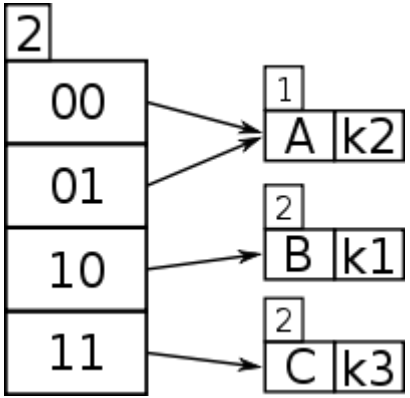
Examining the initial case of an extendible hash structure, if each directory entry points to one bucket, then the local depth should be equal to the global depth.

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

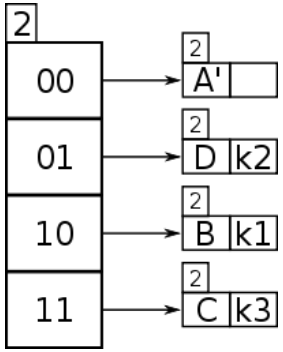
Thus if global depth = local depth = 0, then $2^0 = 1$, so an initial directory of one pointer to one bucket.

Back to the two action cases; if the bucket is full:

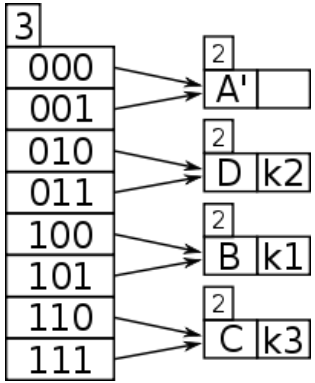
1. If the local depth is equal to the global depth, then there is only one pointer to the bucket, and there is no other directory pointers that can map to the bucket, so the directory must be doubled.
2. If the local depth is less than the global depth, then there exists more than one pointer from the directory to the bucket, and the bucket can be split.



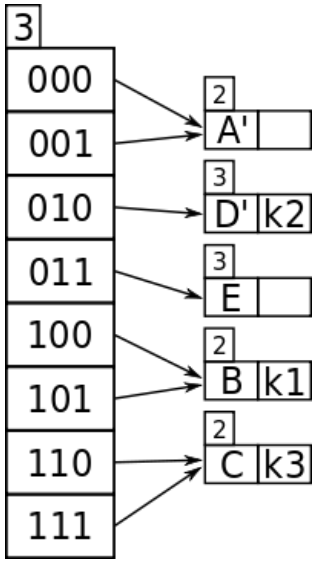
Key 01 points to Bucket A, and Bucket A's local depth of 1 is less than the directory's global depth of 2, which means keys hashed to Bucket A have only used a 1 bit prefix (i.e. 0), and the bucket needs to have its contents split using keys 1 + 1 = 2 bits in length; in general, for any local depth d where d is less than D, the global depth, then d must be incremented after a bucket split, and the new d used as the number of bits of each entry's key to redistribute the entries of the former bucket into the new buckets.



Now, $h(k_4) = 011110$ is tried again, with 2 bits 01.., and now key 01 points to a new bucket but there is still k2 in it. If k2 had been 000110, with key 00, there would have been no problem, because k2 would have remained in the new bucket A' and bucket D would have been empty. So Bucket D needs to be split, but a check of its local depth, which is 2, is the same as the global depth, which is 2, so the directory must be split again, in order to hold keys of sufficient detail, e.g. 3 bits.



1. Bucket D needs to split due to being full.
2. As D's local depth = the global depth, the directory must double to increase bit detail of keys.
3. Global depth has incremented after directory split to 3.
4. The new entry k4 is rekeyed with global depth 3 bits and ends up in D which has local depth 2, which can now be incremented to 3 and D can be split to D' and E.
5. The contents of the split bucket D, k2, has been re-keyed with 3 bits, and it ends up in D.
6. K4 is retried and it ends up in E which has a spare slot.



Now, $h(k_2) = 010110$ is in D and $h(k_4) = 011110$ is tried again, with 3 bits 011.., and it points to bucket D which already contains k2 so is full; D's local depth is 2 but now the global depth is 3 after the directory doubling, so now D can be split into bucket's D' and E, the contents of D, k2 has its h(k2) retried with a new global depth bitmask of 3 and k2 ends up in D', then the new entry k4 is retried with h(k4) bitmasked using the new global depth bit count of 3 and this gives 011 which now points to a new bucket E which is empty. So k4 goes in Bucket E.

Significance of Hashing

- Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure.
- Hashing gives a more secure and adjustable method of retrieving data compared to any other data structure. It is quicker than searching for lists and arrays. In the very range, Hashing can recover data in 1.5 probes, anything that is saved in a tree. Hashing, unlike other data structures, doesn't define the speed. A balance between time and space has to be maintained while hashing. There are two ways of maintaining this balance.
  1. Controlling speed by selecting the space to be allocated for the hash table
  2. Controlling space by choosing a speed of recovery
- Hashed passwords cannot be modified, stolen, or jeopardized. No well-recognized and efficient key or encryption scheme exists that can be misused. Also, there is no need to worry if a hash code is stolen since it cannot be applied anywhere else.
- Two files can be compared for equality easily through hashing. There is no need to open the two documents individually. Hashing compares them word-by-word and the computed hash value instantly tells if they are distinct. This advantage can be used for verification of a file after it has been shifted to a new place. It is an example of SyncBack which is a file backup program.