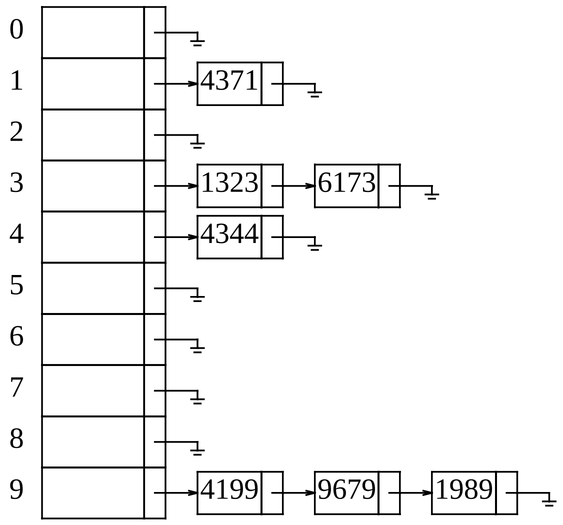
Chapter 5: Hashing

5.1 (a) On the assumption that we add collisions to the end of the list (which is the easier way if a hash table is being built by hand), the separate chaining hash table that results is shown here.



(b)

0	9679
1	4371
2	1989
3	1323
4	6173
5	4344
6	
7	
8	
9	4199

(c)

0	9679
1	4371
2	
3	1323
4	6173
5	4344
6	
7	
8	1989
9	4199

(d) 1989 cannot be inserted into the table because $hash_2(1989) = 6$, and the alternative locations 5, 1, 7, and 3 are already taken. The table at this point is as follows:

0	
1	4371
2	
3	1323
4	6173
5	9679
6	
7	4344
8	
9	4199

5.2 When rehashing, we choose a table size that is roughly twice as large and prime. In our case, the appropriate new table size is 19, with hash function $h(x) = x \, (mod\ 19)$.

(a) Scanning down the separate chaining hash table, the new locations are 4371 in list 1, 1323 in list 12, 6173 in list 17, 4344 in list 12, 4199 in list 0, 9679 in list 8, and 1989 in list 13.

(b) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 14 because both 12 and 13 are already occupied, and 4199 in bucket 0.

(c) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 16 because both 12 and 13 are already occupied, and 4199 in bucket 0.

(d) The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 15 because 12 is already occupied, and 4199 in bucket 0.

5.4 We must be careful not to rehash too often. Let p be the threshold (fraction of table size) at which we rehash to a smaller table. Then if the new table has size N, it contains $2pN$ elements. This table will require rehashing after either $2N - 2pN$ insertions or pN deletions. Balancing these costs suggests that a good choice is $p = 2/3$. For instance, suppose we have a table of size 300. If we rehash at 200 elements, then the new table size is $N = 150$, and we can do either 100 insertions or 100 deletions until a new rehash is required.

If we know that insertions are more frequent than deletions, then we might choose p to be somewhat larger. If p is too close to 1.0, however, then a sequence of a small number of deletions followed by insertions can cause frequent rehashing. In the worst case, if $p = 1.0$, then alternating deletions and insertions both require rehashing.

5.5 (a) Since each table slot is eventually probed, if the table is not empty, the collision can be resolved.

(b) This seems to eliminate primary clustering but not secondary clustering because all elements that hash to some location will try the same collision resolution sequence.

(c, d) The running time is probably similar to quadratic probing. The advantage here is that the insertion can't fail unless the table is full.

(e) A method of generating numbers that are not random (or even pseudorandom) is given in the references. An alternative is to use the method in Exercise 2.7.

5.6 Separate chaining hashing requires the use of pointers, which costs some memory, and the standard method of implementing calls on memory allocation routines, which typically are expensive. Linear probing is easily implemented, but performance degrades severely as the load factor increases because of primary clustering. Quadratic probing is only slightly more difficult to implement and gives good performance in practice. An insertion can fail if the table is half empty, but this is not likely. Even if it were, such an insertion would be so expensive that it wouldn't matter and would almost certainly point up a weakness in the hash function. Double hashing eliminates primary and secondary clustering, but the computation of a second hash function can be costly. Gonnet and Baeza-Yates [8] compare several hashing strategies; their results suggest that quadratic probing is the fastest method.

5.7 Sorting the MN records and eliminating duplicates would require $O(MN \log MN)$ time using a standard sorting algorithm. If terms are merged by using a hash function, then the merging time is constant per term for a total of $O(MN)$. If the output polynomial is small and has only $O(M + N)$ terms, then it is easy to sort it in $O((M + N)\log (M + N))$ time, which is less than $O(MN)$. Thus the total is $O(MN)$. This bound is better because the model is less restrictive: Hashing is performing operations on the keys rather than just comparison between the keys. A similar bound can be obtained by using bucket sort instead of a standard sorting algorithm. Operations such as hashing are much more expensive than comparisons in practice, so this bound might not be an improvement. On the other hand, if the output polynomial is expected to have only $O(M + N)$ terms, then using a hash table saves a huge amount of space, since under these conditions, the hash table needs only

$O(M + N)$ space.

Another method of implementing these operations is to use a search tree instead of a hash table; a balanced tree is required because elements are inserted in the tree with too much order. A splay tree might be particularly well suited for this type of a problem because it does well with sequential accesses. Comparing the different ways of solving the problem is a good programming assignment.

5.8 The table size would be roughly 60,000 entries. Each entry holds 8 bytes, for a total of 480,000 bytes.

5.9 (a) This statement is true.

(b) If a word hashes to a location with value 1, there is no guarantee that the word is in the dictionary. It is possible that it just hashes to the same value as some other word in the dictionary. In our case, the table is approximately 10% full (30,000 words in a table of 300,007), so there is a 10% chance that a word that is not in the dictionary happens to hash out to a location with value 1.

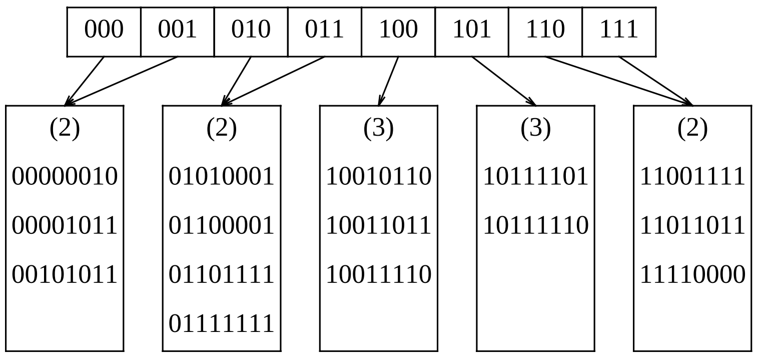
(c) 300,007 bits is 37,501 bytes on most machines.

(d) As discussed in part (b), the algorithm will fail to detect one in ten misspellings on average.

(e) A 20-page document would have about 60 misspellings. This algorithm would be expected to detect 54. A table three times as large would still fit in about 100K bytes and reduce the expected number of errors to two. This is good enough for many applications, especially since spelling detection is a very inexact science. Many misspelled words (especially short ones) are still words. For instance, typing *them* instead of *then* is a misspelling that won't be detected by any algorithm.

5.10 To each hash table slot, we can add an extra field that we'll call *WhereOnStack,* and we can keep an extra stack. When an insertion is first performed into a slot, we push the address (or number) of the slot onto the stack and set the *WhereOnStack* field to point to the top of the stack. When we access a hash table slot, we check that *WhereOnStack* points to a valid part of the stack and that the entry in the (middle of the) stack that is pointed to by the *WhereOnStack* field has that hash table slot as an address.

5.14


000
001
010
011
100
101
110
111
(2)
00000010
00001011
00101011
(2)
01010001
01100001
01101111
01111111
(3)
10010110
10011011
10011110
(3)
10111101
10111110
(2)
11001111
11011011
11110000