DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 10: Hashing

Introduction (continued)

- If the hash function *h* is able to transform different key values into different hash values, it is called a ***perfect hash function***
- For the hash function to be perfect, the table must have as many positions as there are items to be hashed
- However, it is not always possible to know how many elements will be hashed in advance, so some estimating is needed
- Consider a symbol table for a compiler, to store all the variable names
- Given the nature of the variable names typically used, a table with 1000 positions may be more than adequate
- However, even if we wanted to handle all possible variable names, we still need to design an appropriate *h*

Hash Functions

- The total possible number of hash functions for *n* items assigned to *m* positions in a table ($n \leq m$) is m^n
- The number of perfect hash functions is equal to the number of different placements of these items, and is $\frac{m!}{(m-n)!}$
- With 50 elements and a 100-position array, we would have a total of 100^{50} hash functions and about 10^{94} perfect hash functions (about 1 in a million)
- Most of the perfect hashes are impractical and cannot be expressed in a simple formula

Hash Functions (continued)

- Division
  - Hash functions must guarantee that the value they produce is a valid index to the table
  - A fairly easy way to ensure this is to use modular division, and divide the keys by the size of the table, so $h(K) = K \bmod TSize$ where $TSize = sizeof(table)$
  - This works best if the table size is a prime number, but if not, we can use $h(K) = (K \bmod p) \bmod TSize$ for a prime $p \geq TSize$
  - However, nonprimes work well for the divisor provided they do not have any prime factors less than 20
  - The division method is frequently used when little is known about the keys

Hash Functions (continued)

- Folding
  - In folding, the keys are divided into parts which are then combined (or “folded”) together and often transformed into the address
  - Two types of folding are used, ***shift folding*** and ***boundary folding***
  - In shift folding, the parts are placed underneath each other and then processed (for example, by adding)
  - Using a Social Security number, say 123-45-6789, we can divide it into three parts - 123, 456, and 789 – and add them to get 1368
  - This can then be divided modulo *TSize* to get the address
  - With boundary folding, the key is visualized as being written on a piece of paper and folded on the boundaries between the parts

Hash Functions (continued)

- Folding (continued)
  - The result is that alternating parts of the key are reversed, so the Social Security number part would be 123, 654, 789, totaling 1566
  - As can be seen, in both versions, the key is divided into even length parts of some fixed size, plus any leftover digits
  - Then these are added together and the result is divided modulo the table size
  - Consequently this is very fast and efficient, especially if bit strings are used instead of numbers
  - With character strings, one approach is to exclusively-or the individual character together and use the result
  - In this way, *h*("abcd") = "a" ⋁ "b" ⋁ "c" ⋁ "d"

Hash Functions (continued)

- Folding (continued)
  - However, this is limited, because it will only generate values between 0 and 127
  - A better approach is to use chunks of characters, where each chunk has as many characters as bytes in an integer
  - On the IBM PC, integers are often 2 bytes long, so *h*("abcd") = "ab" $\vee$ "cd", which would then be divided modulo *TSize*

Hash Functions (continued)

- Mid-Square Function
  - In the mid-square approach, the numeric value of the key is squared and the middle part is extracted to serve as the address
  - If the key is non-numeric, some type of preprocessing needs to be done to create a numeric value, such as folding
  - Since the entire key participates in generating the address, there is a better chance of generating different addresses for different keys
  - So if the key is 3121, 3121^2 = 9,740,641, and if the table has 1000 locations, $h(3121) = 406$, which is the middle part of 3121^2
  - In application, powers of two are more efficient for the table size and the middle of the bit string of the square of the key is used
  - Assuming a table size of 1024, 3121^2 is represented by the bit string 1001010 *0101000010* 1100001, and the key, 322, is in italics

Hash Functions (continued)

- Extraction
  - In the extraction approach, the address is derived by using a portion of the key
  - Using the SSN 123-45-6789, we could use the first four digits, 1234, the last four 6789, or the first two combined with the last two 1289
  - Other combinations are also possible, but each time only a portion of the key is used
  - With careful choice of digits, this may be sufficient for address generation
  - For example, some universities give international students ID numbers beginning with 999; ISBNs start with digits representing the publisher
  - So these could be excluded from the address generation if the nature of the data is appropriately limited

Hash Functions (continued)

- Radix Transformation
  - With radix transformation, the key is transformed into a different base
  - For instance, if *K* is 345 in decimal, its value in base 9 is 423
  - This result is then divided modulo *TSize*, and the resulting value becomes the address top which *K* is hashed
  - The drawback to this approach is collisions cannot be avoided
  - For example, if *TSize* is 100, then although 345 and 245 in decimal will not collide, 345 and 264 will because 264 is 323 in base nine
  - Since 345 is 423, these two values will collide when divided modulo 100

Hash Functions (continued)

- Universal Hash Functions
  - When little is known about the keys, a ***universal class of hash functions*** can be used
  - Functions are universal when a randomly chosen member of the class will be expected to distribute a sample evenly, guaranteeing low collisions
  - This idea was first considered by Larry Carter and Mark Wegman in 1979
  - Instead of using a defined hash function, for which a bad set of keys may exist with many collisions, we select a hash function randomly from a family of hash functions! This is a real-time decision
  - *H* is called universal if no distinct pair of keys are mapped to the same position in the table by a function chosen at random from *h* with a probability of 1 / *TSize*
  - This basically means there is one chance in *TSize* that two randomly chosen keys collide when hashed with a randomly chosen function

Hash Functions (continued)

- Universal Hash Functions (continued)
  - One example of such a class of functions is defined for a prime number $p \geq |keys|$ and random numbers a and b

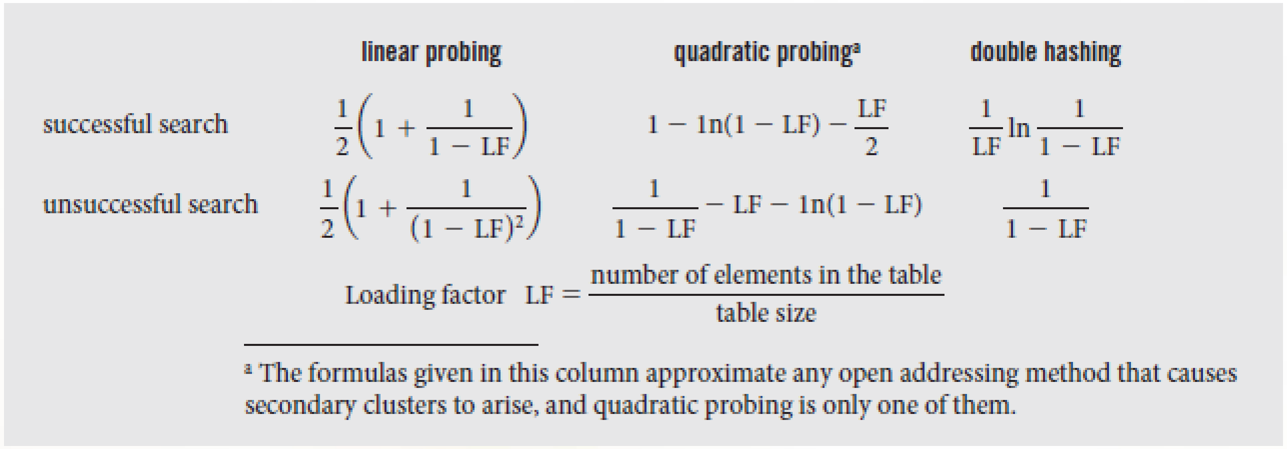
    $H = \{h_{a,b}(K): h_{a,b}(K) = ((aK+b) \bmod p) \bmod TSize \text{ and } 0 \leq a, b < p\}$

  - Another class of functions is for keys considered as sequences of bytes, $K = K_0K_1 \ldots K_{r-1}$
  - For a prime $p \geq 2^8 = 256$ and a sequence $a = a_0, a_1, \ldots, a_{r-1}$,

$$H = \left\{ h_a(K) : h_a(K) = \left(\left(\sum_{i=0}^{r-1} a_i K_i \right) \bmod p \right) \bmod TSize \text{ and } 0 \leq a_0, a_1, \ldots, a_{r-1} < p \right\}$$

Collision Resolution

- Open Addressing
  - The efficiency of different open addressing techniques depends on the size of the table and number of elements in the table
  - There are formulas, developed by Donald Knuth, that approximate the number of times for successful and unsuccessful searches
  - These are shown in Figure 10.3



	linear probing	quadratic probing[a]	double hashing
successful search	$\frac{1}{2}\left(1 + \frac{1}{1 - LF}\right)$	$1 - \ln(1 - LF) - \frac{LF}{2}$	$\frac{1}{LF}\ln\frac{1}{1 - LF}$
unsuccessful search	$\frac{1}{2}\left(1 + \frac{1}{(1 - LF)^2}\right)$	$\frac{1}{1 - LF} - LF - \ln(1 - LF)$	$\frac{1}{1 - LF}$

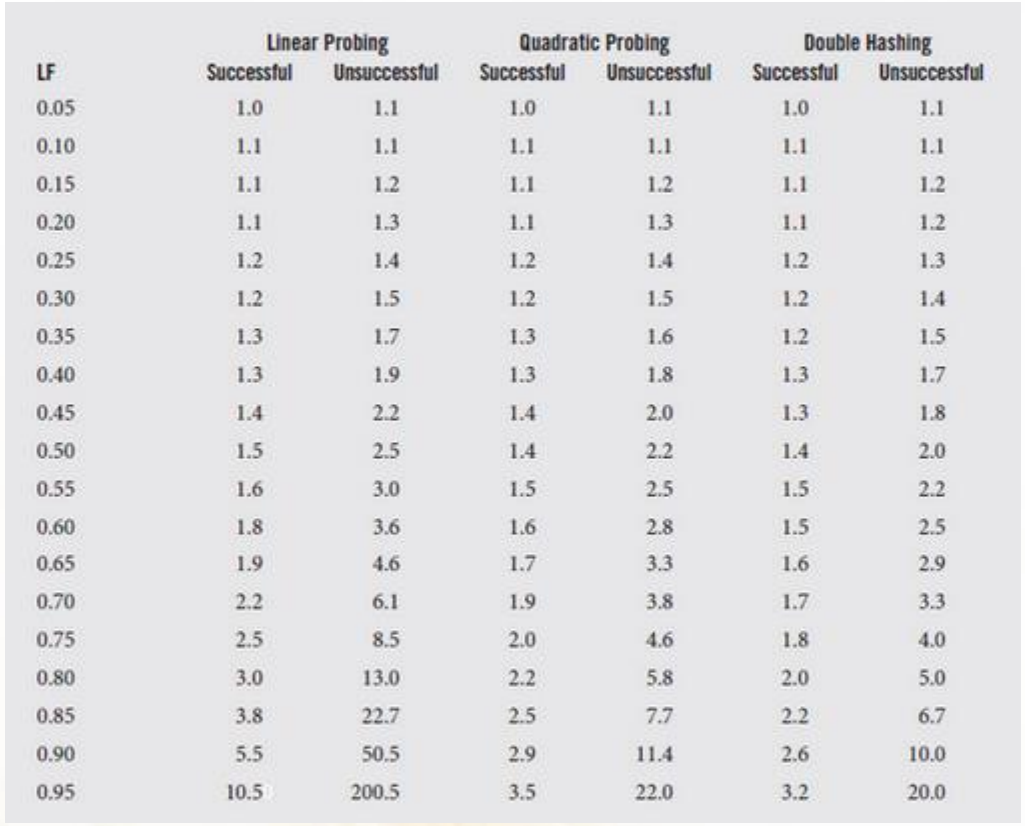
Loading factor $LF = \frac{\text{number of elements in the table}}{\text{table size}}$

[a] The formulas given in this column approximate any open addressing method that causes secondary clusters to arise, and quadratic probing is only one of them.

Fig. 10.3 Formulas approximating, for different hashing methods, the average numbers of trials for successful and unsuccessful searches (Knuth 1998)

Collision Resolution (continued)

- Open Addressing (continued)
  - Figure 10.4 shows the number of searches for different percentages of occupied cells



	Linear Probing		Quadratic Probing		Double Hashing	
LF	Successful	Unsuccessful	Successful	Unsuccessful	Successful	Unsuccessful
0.05	1.0	1.1	1.0	1.1	1.0	1.1
0.10	1.1	1.1	1.1	1.1	1.1	1.1
0.15	1.1	1.2	1.1	1.2	1.1	1.2
0.20	1.1	1.3	1.1	1.3	1.1	1.2
0.25	1.2	1.4	1.2	1.4	1.2	1.3
0.30	1.2	1.5	1.2	1.5	1.2	1.4
0.35	1.3	1.7	1.3	1.6	1.2	1.5
0.40	1.3	1.9	1.3	1.8	1.3	1.7
0.45	1.4	2.2	1.4	2.0	1.3	1.8
0.50	1.5	2.5	1.4	2.2	1.4	2.0
0.55	1.6	3.0	1.5	2.5	1.5	2.2
0.60	1.8	3.6	1.6	2.8	1.5	2.5
0.65	1.9	4.6	1.7	3.3	1.6	2.9
0.70	2.2	6.1	1.9	3.8	1.7	3.3
0.75	2.5	8.5	2.0	4.6	1.8	4.0
0.80	3.0	13.0	2.2	5.8	2.0	5.0
0.85	3.8	22.7	2.5	7.7	2.2	6.7
0.90	5.5	50.5	2.9	11.4	2.6	10.0
0.95	10.5	200.5	3.5	22.0	3.2	20.0

Fig. 10.4 The average numbers of successful searches and unsuccessful searches for different collision resolution methods

Collision Resolution (continued)

- Chaining
  - In ***chaining***, the keys are not stored in the table, but in the `info` portion of a linked list of nodes associated with each table position
  - This technique is called ***separate chaining***, and the table is called a ***scatter table***
  - This was the table never overflows, as the lists are extended when new keys arrive, as can be seen in Figure 10.5
  - This is very fast for short lists, but as they increase in size, performance can degrade sharply
  - Gains in performance can be made if the lists are ordered so unsuccessful searches don't traverse the entire list, or by using self-organizing linked lists
  - This approach requires additional space for the pointers, so if there are a large number of keys involved, space requirements can be high

Collision Resolution (continued)

- Bucket Addressing
  - Yet another approach to collision handling is to store all the colliding elements in the same position in the table
  - This is done by allocating a block of space, called a ***bucket***, with each address
  - Each bucket is capable of storing multiple items
  - However, even with buckets, we cannot avoid collisions, because buckets can fill up, requiring items to be stored elsewhere
  - If open addressing is incorporated into the design, the item can be stored in the next bucket if space is available, using linear probing
  - This is shown in Figure 10.8

Collision Resolution (continued)

- Bucket Addressing (continued)

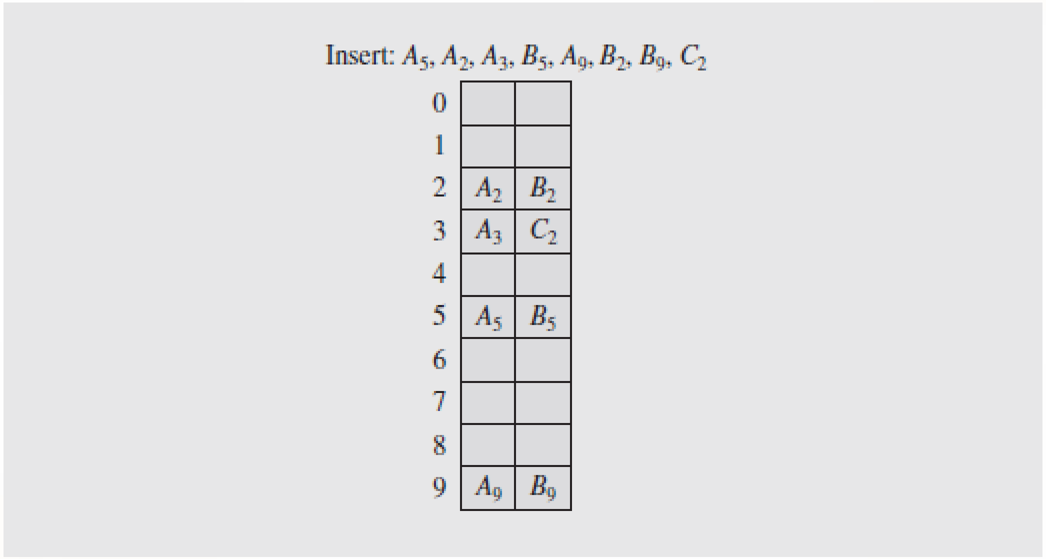


Fig. 10.8 Collision resolution with buckets and linear probing method

  - Collisions can be stored in an overflow area, in which case the bucket includes a field to indicate if a search should consider that area or not
  - If used with chaining, the field could indicate the beginning of the list in the overflow area associated with the bucket shown in Figure 10.9

Deletion

- How can data be removed from a hash table?
- If chaining is used, the deletion of an element entails deleting the node from the linked list holding the element
- For the other techniques we’ve considered, deletion usually involves more careful handling of collision issues, unless a perfect hash function is used
- This is illustrated in Figure 10.10a, which stores keys using linear probing
- In Figure 10.10b, when A_4 is deleted, attempts to find B_4 check location 4, which is empty, indicating B_4 is not in the table
- A similar situation occurs in Figure 10.10c, when A_2 is deleted, causing searches for B_1 to stop at position 2

Deletion (continued)

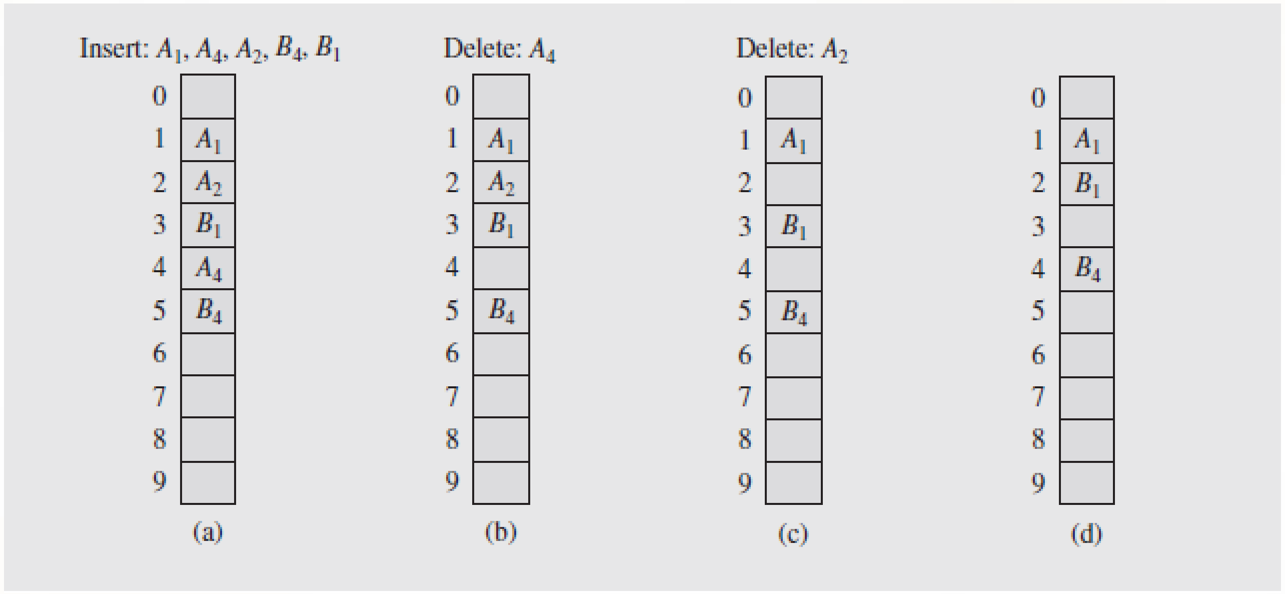


Fig. 10.10 Linear search in the situation where both insertion and deletion of keys are permitted

- A solution to this is to leave the deleted keys in the table with some type of indicator that the keys are not valid
- This way, searches for elements won't terminate prematurely
- When new keys are inserted, they can overwrite the marked keys

Deletion (continued)

- A drawback is if the table becomes overloaded with deleted records, slowing down search times, because open addressing requires testing each element
- So the table needs to be purged periodically by moving undeleted elements to cells occupied by deleted elements
- Those cells containing deleted elements not overwritten can then be marked as available

Perfect Hash Functions

- All the examples we've considered to this point have assumed the data being hashed is not completely known
- Consequently, the hashing that took place only coincidentally turned out to be ideal in that collisions were avoided
- The majority of the time collisions had to be resolved because of conflicting keys
- In addition, the number of keys is usually not known in advance, so the table size had to be large enough
- Table size also played a role in the number of collisions; larger tables had fewer collisions if the hash took this into account
- All these factors were the result of not knowing ahead of time about the body of data to be hashed

Perfect Hash Functions (continued)

- Therefore the hash function was developed first and then the data was processed into the table
- In a number of cases, though, the data is known in advance, and the hash function can be derived after the fact
- This function may turn out to be a perfect hash if items hash on the first try
- Additionally, if the function uses only as many cells as are available in the table with no empty cells left after the hash, it is called a ***minimal perfect hash function***
- Minimal perfect hash functions avoid the need for collision resolution and also avoid wasting table space

Perfect Hash Functions (continued)

- Processing fixed bodies of data occurs frequently in a variety of applications
- Dictionaries and tables of reserved words in compilers are among several examples
- Creating perfect hash functions requires a great deal of work
- Such functions are rare; as we saw earlier, for 50 elements in a 100-position array, only 1 function in a million is perfect
- All the other functions will lead to collisions

- Book has examples on Cichelli's Method and FHCD for minimal perfect hash functions for small number of strings

Rehashing

- When hash tables become full, no more items can be added
- As they fill up and reach certain levels of occupancy (saturation), their efficiency falls due to increased searching needed to place items
- A solution to these problems is rehashing, allocating a new, larger table, possibly modifying the hash function (and at least *TSize*), and hashing all the items from the old table to the new
- The old table is then discarded and all further hashes are done to the new table with the new function
- The size of the new table can be determined in a number of ways: doubled, a prime closest to doubled, etc.

Rehashing (continued)

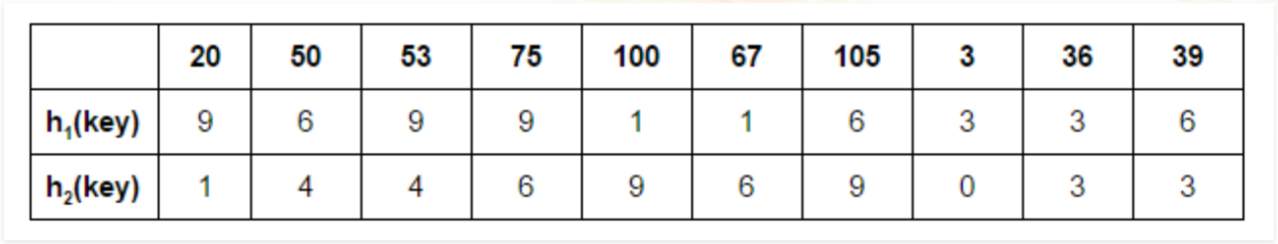
- All the methods we’ve looked at can use rehashing, as they then continue using the processes of hashing and collision resolution that were in place before rehashing
- One method in particular for which rehashing is important is ***cuckoo hashing***, first described by Rasmus Pagh and Flemming Rodler in 2001

Rehashing (continued)

- The cuckoo hashing
  - Two tables, T_1 and T_2, and two hash tables, h_1 and h_2, are used in the cuckoo hash
  - Inserting a key K_1 into table T_1 uses hash function h_1, and if $T_1[h_1(K_1)]$ is open, the key is inserted there
  - If the location is occupied by a key, say K_2, this key is removed to allow K_1 to be placed, and K_2 is placed in the second table at $T_2[h_2(K_2)]$
  - If this location is occupied by key K_3, it is moved to make room for K_2, and an attempt is made to place K_3 at $T_1[h_1(K_3)]$
  - So the key that is being placed in or moved to the table has priority over a key that is already there
  - There is a possibility that a sequence like this could lead to an infinite loop if it ends up back at the first position that was tried

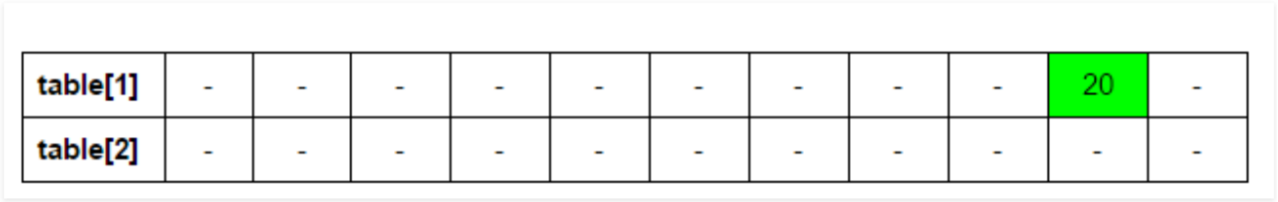
Rehashing (continued)

- The cuckoo hashing (continued)
  - It is also possible the search will fail because both tables are full
  - To circumvent this, a limit is set on tries which if exceeded causes rehashing to take place with two new, larger tables and new hash functions
  - Then the keys from the old tables are rehashed to the new ones
  - If during this the limit on number of tries is exceeded, rehashing is performed again with yet larger tables and new hash functions



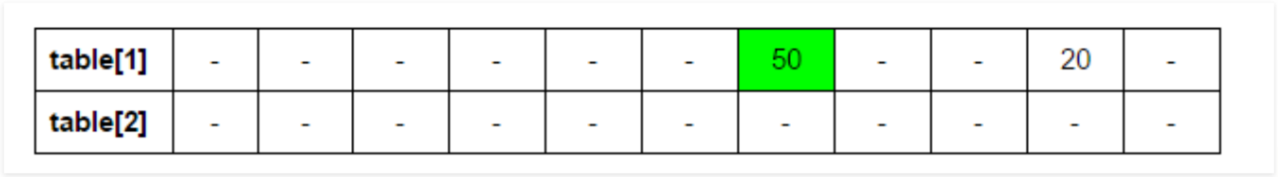
	20	50	53	75	100	67	105	3	36	39
h_1(key)	9	6	9	9	1	1	6	3	3	6
h_2(key)	1	4	4	6	9	6	9	0	3	3

Let’s start with inserting **20** at its possible position in the first table determined by h1(20):



table[1]	-	-	-	-	-	-	-	-	-	20	-
table[2]	-	-	-	-	-	-	-	-	-	-	-

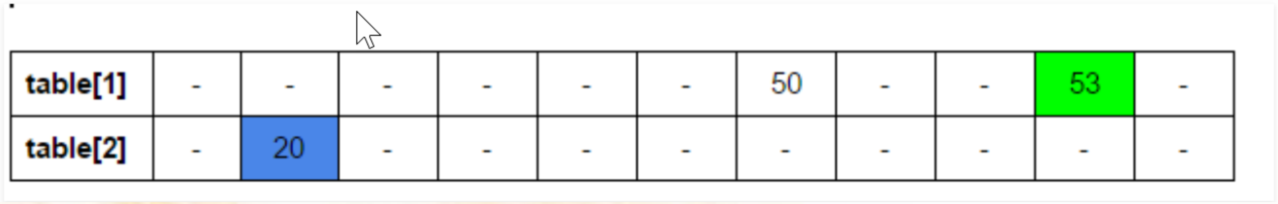
Next: **50**



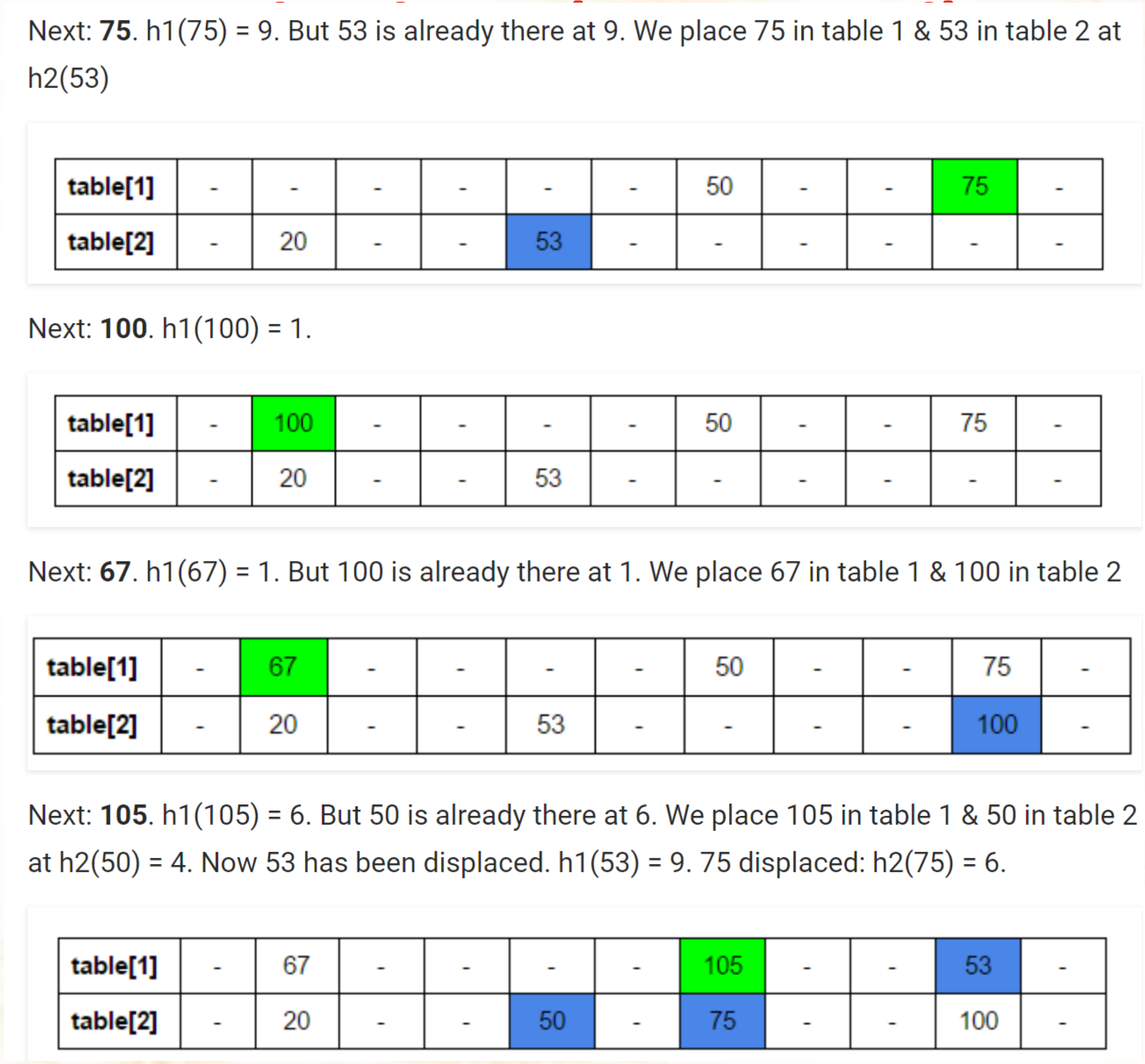
table[1]	-	-	-	-	-	-	50	-	-	20	-
table[2]	-	-	-	-	-	-	-	-	-	-	-

Next: **53**. h1(53) = 9. But 20 is already there at 9. We place 53 in table 1 & 20 in table 2 at h2(20)

.



table[1]	-	-	-	-	-	-	50	-	-	53	-
table[2]	-	20	-	-	-	-	-	-	-	-	-



Next: **75**. h1(75) = 9. But 53 is already there at 9. We place 75 in table 1 & 53 in table 2 at h2(53)

table[1]	-	-	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	-	-

Next: **100**. h1(100) = 1.

table[1]	-	100	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	-	-

Next: **67**. h1(67) = 1. But 100 is already there at 1. We place 67 in table 1 & 100 in table 2

table[1]	-	67	-	-	-	-	50	-	-	75	-
table[2]	-	20	-	-	53	-	-	-	-	100	-

Next: **105**. h1(105) = 6. But 50 is already there at 6. We place 105 in table 1 & 50 in table 2 at h2(50) = 4. Now 53 has been displaced. h1(53) = 9. 75 displaced: h2(75) = 6.

table[1]	-	67	-	-	-	-	105	-	-	53	-
table[2]	-	20	-	-	50	-	75	-	-	100	-

Rehashing (continued)

- The cuckoo hashing (continued)
  - One point to note is that rehashing may be limited, so that instead of creating tables, only new hash functions are created and keys reprocessed
  - However, this would be a global operation requiring both tables be completely processed

Hash Functions for Extendible Files

- While rehashing adds flexibility to hashing by allowing for dynamic expansion of the has table, it has drawbacks
- In particular, the entire process comes to a halt while the new table(s) are created and the values rehashed to the new table
- The time required for this may be unacceptable in many cases
- An alternative approach is to expand the table rather than replace it, and only allow for local rehashing and local changes
- This approach won’t work with arrays, because expanding an array can’t be done by simply adding locations to the end
- However, it can be managed if the data is kept in a file

Hash Functions for Extendible Files (continued)

- There are some hashing techniques that take into account variable sizes of tables or files
- These fall into two groups: directory and directoryless
- In directory schemes, a directory or index of keys controls access to the keys themselves
- There are a number of techniques that fall into this category
  - ***Expandable hashing***, developed by Gary D. Knott in 1971
  - ***Dynamic hashing***, developed by Per-Âke Larson in 1978
  - ***Extendible hashing***, developed by Ronald Fagin and others in 1979
- All of these distribute keys among buckets in similar ways
- The structure of the directory or index is the main difference
- Skipping details, see book for more