UNIT-2

HASHING

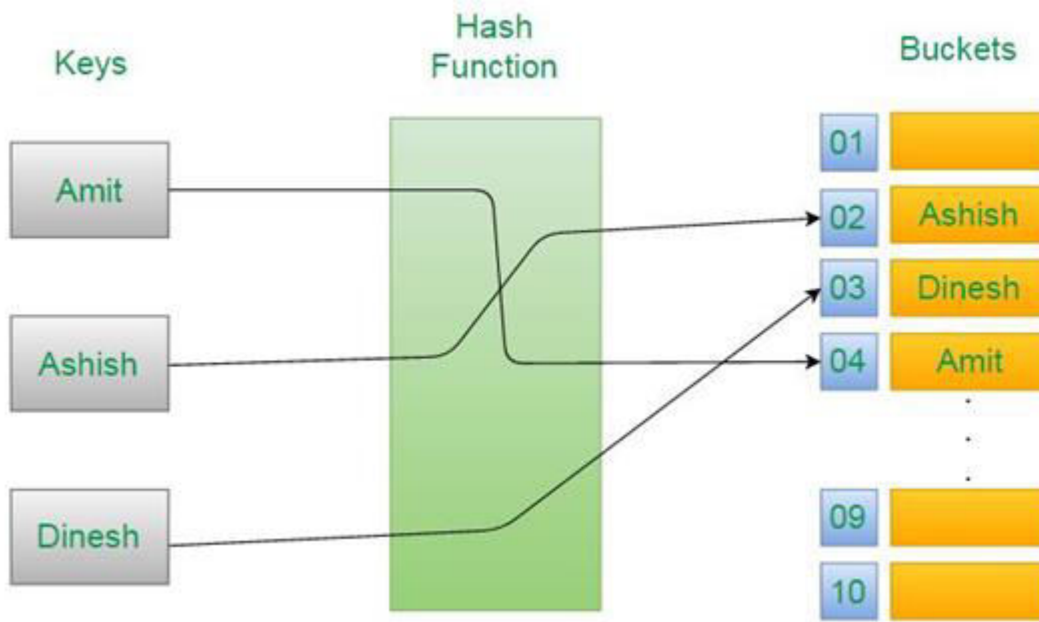
STATIC HASHING

Hash Tables

Hash Table

Hash Table is one of the most important and widely used data structure which uses a hash function to compute an index into an array of buckets/slots where the value can be stored/retrieved.

For example:- Let us see if we have list of names and we want to store these values in hash table -



From the above, it is clear that hash table has two parts: a key used for indexing and data field to store the value associate with that index.

Let us understand the cost of various operations on hash table -

	Average Case	Worst Case
Search	O(1)	O(n)

Insert	O(1)	O(n)
Delete	O(1)	O(n)

Hashing & Hash Function

Hashing is a technique through which given data can be stored/retrieved from hashtable. For this a hash function is required which can generate an integer value for the given input. This integer value also known as hash value can be used to store given value to particular index of hashtable.

For example:- If we have 100 buckets available in hashtable, then the hash value should lie between 0 & 100(0 inclusive). Mathematically,

h(k) => [0,100)

If we are able to write a function which generates an 1-to-1 association between input and generated hash function it is known as perfect hash function but unfortunately in practical scenario it is not possible. In our next chapter we will understand various techiques to write hash function and how to handle situation when more than one input have same hashcode/hash value.

The two principal criteria in selecting a hash function are -

1. It should be easy and quick to compute.
2. It should achieve an even distribution of the keys that actually occur across the range of indices.

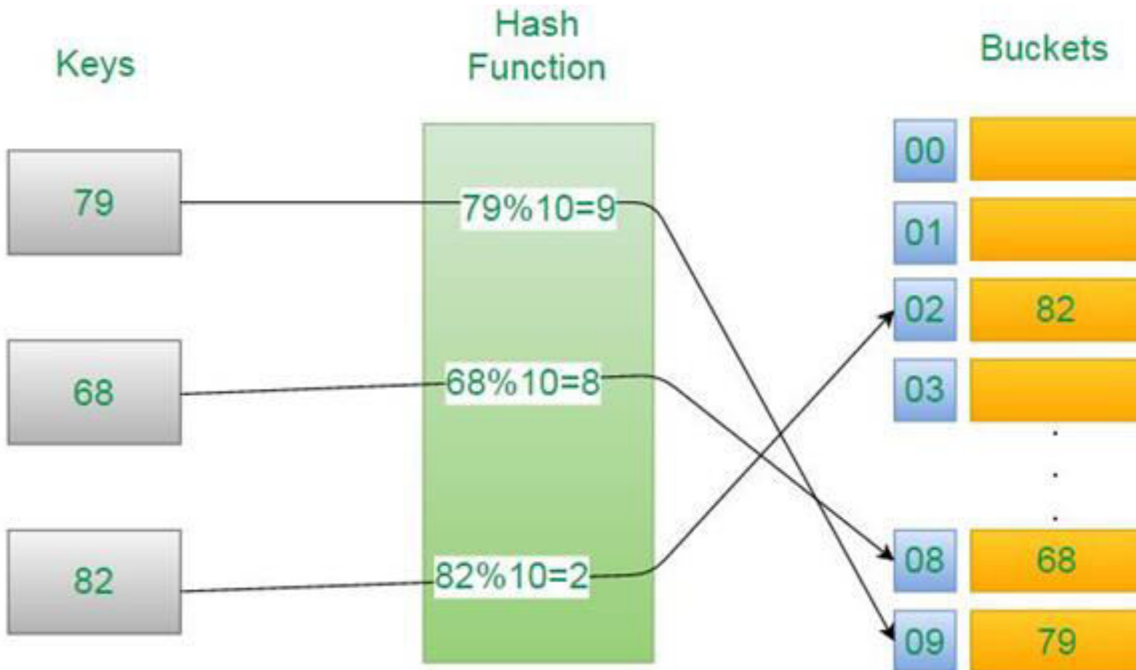
There are many ways to create an hash function. In this chapter, we will understand division/modulo method to create hash function.

Division Method

If there are m slots available, then the key is mapped to given m slots using following formula:-

$h(k) = k \bmod m$

For example:- Let us say apply division approach to find hash value for some values considering number of buckets be 10 as shown below.



The division method is generally a good choice, unless the key happens to have some undesirable properties. For example, if the table size is 10 and all of the keys end in zero. In this case, the choice of hash function and table size needs to be carefully considered. The best table sizes are prime numbers.

What if the given key is not an integer? You have to apply do two operations to get the hash code:

1. function to get an integer for key
2. function to get hashcode for above value.

Folding Method:

In this method, the given key is partitioned into subparts k_1, k_2, k_3, k_4 ...... k_n each of which has the same lenght as the required address. Now add all these parts together and ignore the carry.

For example:- if number of buckets be 100 and last address/index be 99, then the given key for which hashcode is calculated is divided into parts of two digits from begining as shown below

h(95073) = h(95 + 07 + 3)

= h(105) //ignoring the carry

= 5

MidSquare Method:

In this method, hash code for given key is obtained by squaring the key and taking m bits from the middle to obtain the bucket addresss.

The number of bits used to obtain the bucket address depends on the table size. For example:- If the size of table is 100, then we can fetch 2 bits from middle.

Since the middle bits of the square depend upon all the bits of the given key, there is a high probability that different key will generate different hash code.

As we know hash function is used to generate index/address of particular key in hash table. Noe it may be the case that for two keys we get same hashcode. This is known as collision.(We will learn about collision and its resolution in detail in our next chapter).

Hence if a hash function that maps each key to a distinct address/index, it is known as perfect hash function. To design perfect hash function, usually all the keys must be known beforehand.

In this case the worst case search complexity will be O(1).

Overflow Handling:

Open Addressing

There are two ways to handle overflows: open addressing and chaining. In open 30- dressing, we assume the hash table is an array. For simplicity, we assume that s = 1. The class definitions used are in Program.

To detect collisions and overflows, the hash table, h1, is initialized so that each slot contains the null identifier. When a new identifier is hashed into a full bucket, we need to find another bucket for this identifier. The simplest solution is to find the closest unfilled bucket. This is called linear probing or linear open addressing.

Example

Assume we have a 26-bucket table with o~ slot per bucket and the following identifiers: GA, D, A, G, L, A2, A1. A3: A4. z, ZA. E. For simplicity we choose the .hash function h (x) = first character of x. Initially, all entries in the table are null.

When linear open addressing is used to handle overflows, a hash table search for identifiers proceeds as follows:

(1)′ compute h (x) ·

(2) examine identifiers at positions ht [h (x)], ht[h(x) + 1], … , ht [h (x) + j), in this order, until one of the following happens:

(a) ht [h (x) + j] = x; in this case x is found.

(b) ht [h (x) + j] is null; x is not in the table.

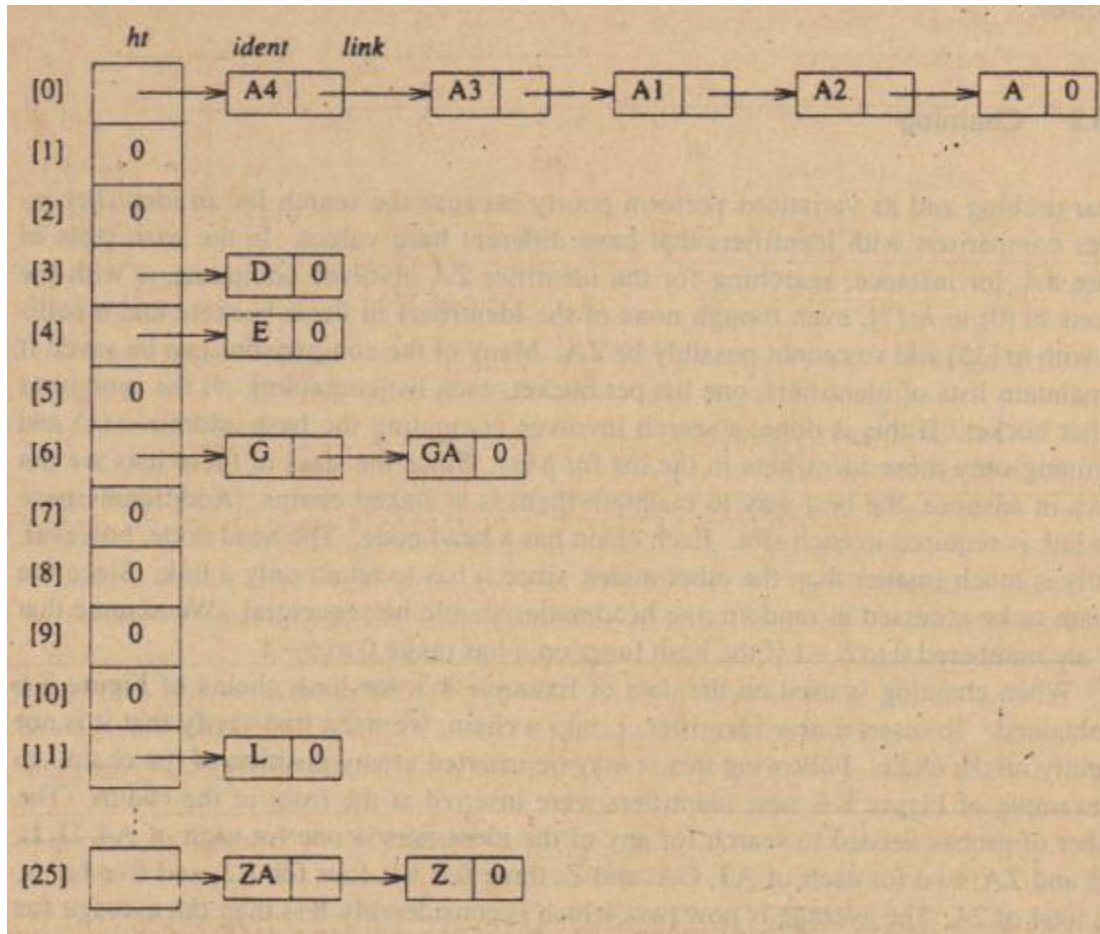
(c) We return to the starting position h(x); the table is full and x is not in the table.

0	A
1	A2
2	A1
3	D
4	A3
5	A4
6	GA
7	G
8	ZA
9	E
10	
11	L
12	
13	
⋮	⋮
24	
25	Z

Linear probing and its variations perform poorly because the search for an identifier involves comparison with identifiers that have different hash values. In the hash table of Figure, for instance. searching for the identifier ZA involves comparisons with the buckets ht [0) to ht [7], even though none of the identifiers in these buckets had a collision with ht [25) and so cannot possibly be ZA.

Chaining:

When chaining is used on the data of Example, the hash chains of Figure are obtained. To insert a new identifier, .r, into a chain. we must first verify that it is not currently on the chain. Following this. x may be inserted at any position of the chain. In the example of figure. new identifiers were inserted at the front of the chains. The number of probes needed to search for any of the identifiers is one for each of A4. D, E. G, L, and ZA; two for each of A3, GA. and Z; three for A I; four for A2; and five for A, for a total of 24. The average is now two. which is considerably less than the. average for linear probing.



The expected number of identifier comparisons can be shown to be 1012. where a is the loading density nib (b = number of head nodes). For a = 0.5 This figure is 1.25, and for a = I it is about .1.5. This scheme has the additional advantage that only the b head nodes must be sequential and reserved at the beginning. Each head node, however, will be at most one-half to one word long. The other nodes will be much bigger and are allocated only as needed. This could represent an overall reduction in space required. for certain loading densities, despite the links. If each record in the table is five words long, n = 100, and a::: 0.5, then the hash table will be of size 200 x 5 = 1000 words. Only 500 of these are used, as a = 0.5. On the other hand, if chaining is used with one full word per link, then 200 words are needed for the head nodes (b = 2(0). Each head node is one word long.

Theoretical Evaluation of Overflow Techniques:

The experimental evaluation of hashing techniques indicates a very good performance over conventional techniques such as balanced trees. The worst-case performance for hashing can. however. be very bad. In the worst case, an insertion or a search in a hash table with II identifiers may take 0(11) time.

Let ht [0 .. b – I] be a hash table with b buckets. each bucket having one slot. Let h be a uniform hash function with range [0. b – 1]. If n identifiers x I. X2 …. Xn are entered into the hash table. then there are b" distinct hash sequences h (x d. h (X2). …. h (xn). Assume that each of these is equally likely to occur. Let Sn denote the expected number of identifier comparisons needed to locate a randomly chosen Xi. 1 S is n. Then. Sn is the average number of comparisons needed to find the jth key Xj. averaged over I Sj:S; n, with each j equally likely, and averaged over all b" hash sequences. assuming each of these also to be equally likely.

Theorem

Let ex = nib be the loading density c; a hash table using a uniform hashing function h. Then

(I) for linear open addressing.

$$U_n \approx \frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$

$$S_n \approx \frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

(2) for rehashing, random probing, and quadratic probing

$$U_n \approx 1/(1-\alpha)$$

$$S_n \approx -\left[\frac{1}{\alpha}\right]\log_e(1-\alpha)$$

(3) for chaining

$$U_n \approx \alpha$$

$$S_n \approx 1 + \alpha/2$$

Proof: Exact derivations of U; .and S; are fairly involved and can be found in Knuth's book The Art of Computer Programming: Sorting and Searching (see the References and Selected Readings section). Here we present a derivation of the approximate formulas for chaining. First, we must make clear our count for Un and Sn. If the identifier x being sought has h (x) = i, and chain i has k nodes on it (not including the head node), then k comparisons are needed if x is not on the chain. If x is j nodes away from the head node, 1 S j S k, then j comparisons are needed.

When the n identifiers are distributed uniformly over the b possible chains, the expected number in each chain is nib = a. Since U; equals the expected number of identifiers on a chain; we get Un = a.

When the ith identifier, Xi' is being entered into the table, the expected number of identifiers on any chain is (i – 1)1b. Hence, the expected number of comparisons needed to search for Xi after .all n identifiers have been entered is 1 + (; – 1)1b (this assumes that new entries will be made at the end of the chain). Thus.

$$S_n = \frac{1}{n}\sum_{i=1}^{n}[1 + (i-1)/b] = 1 + \frac{n-1}{2b} = 1 + \frac{\alpha}{2} \square$$

Dynamic Hashing:

Motivation for Dynamic Hashing

Traditional hashing schemes as described in the previous section are not ideal. This follows from the fact that one must statically allocate a portion of memory to hold the hash table. This hash table is used to point to the pages used to hold identifiers, or it may actually. hold the identifiers themselves. In either case, if the table is allocated to be as large as possible. then space can be wasted. If it is allocated to be too small, then when the data exceed the capacity of the hash table, the entire file must be restructured. a time-consuming process. The purpose of dynamic hashing (also referred to as extendible hashing) is to retain the fast retrieval time of conventional hashing while extending the technique so that it can accommodate dynamically increasing and decreasing tile size without penalty.

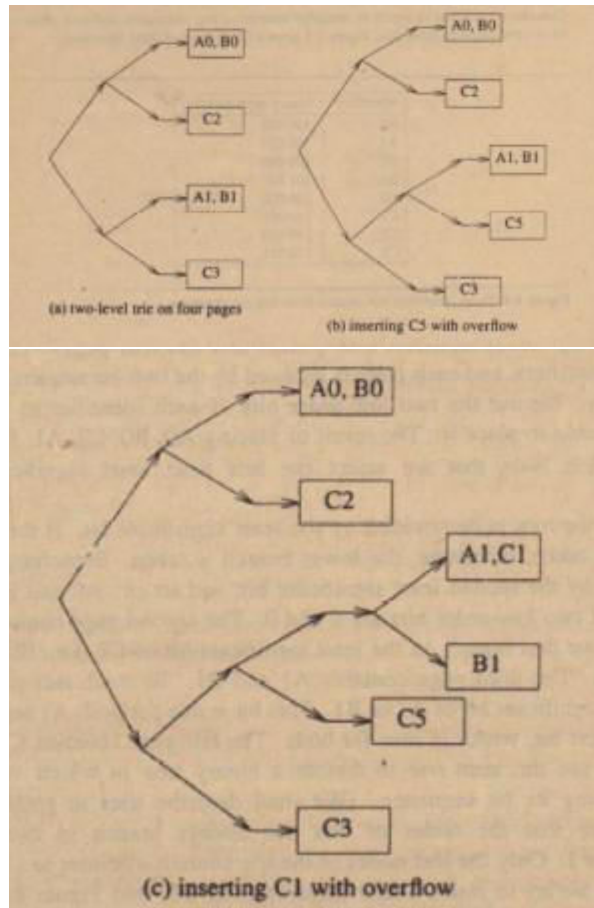
Dynamic Hashing Using Directories

Consider an example in which an identifier consists of two characters. and each character is represented by three bits. Figure gives a list of some of these identifiers.

identifiers	binary represenation
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C0	110 000
C1	110 001
C2	110 010
C3	110 011

Consider placing these identifiers into a table that has four pages. Each page can hold at most two identifiers, and each page is indexed by the two-bit sequence 00. 01. 10. and 11, respectively. We use the two low-order bits of each identifier to determine in which page of the table to place it. The result of placing AO, BO. C2. AI. B I, and C3 is shown. Note that we select the bits from least significant to most significant.

Now suppose we try to insert a new identifier, say C5, into Figure 8.9(a). Since the two low-order bits of C5 are I and O. we must place it in the third page. However, this page is full, and an overflow occurs. A new page is added. and the depth of the trie increases by one level. (b). If we now try to insert the new identifier C 1. an overflow of the page containing A I and B I occurs. A new page is obtained, and the three identifiers are divided between the two pages according to their four low-order bits.

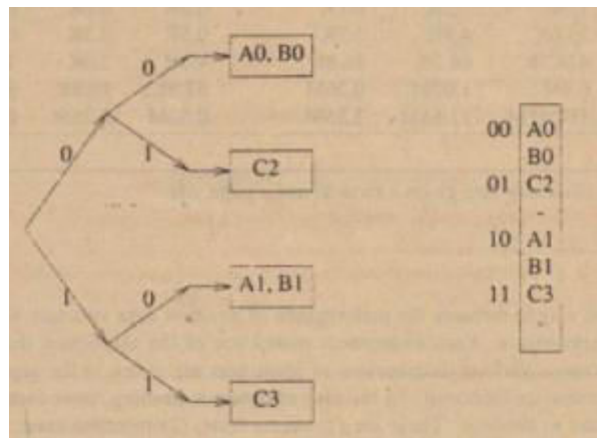


From this example one can see that two major problems exist. First, the access time for a page depends on the number of bits needed to distinguish the identifiers. Second, if the identifiers have a skewed distribution, the tree is also skewed. Both of these factors increase the retrieval time. Fagin etal. (1979) present a method. which they call extendible hashing, for solving these problems. To avoid the skewed distribution of identifiers, a hash function is used. 'This function takes the key and produces a random set of binary digits. To avoid the long search down the trie, the trie is mapped to a directory.

A directory is a table of page pointers. If k bits are needed to distinguish the identifiers, the directory has 2k entries indexed 0, … , 2(-1. To find the page for an identifier, we use the integer with binary representation equal to the last k bits of the identifier. The page pointed at by this directory entry is searched. Figure shows the directories corresponding to the three tries in Figure 8.9. The first directory contains four entries indexed from 0 to 3 (the binary representation of each index is shown in Figure). Each entry contains a pointer to a page. This pointer is shown as .10 arrow in the figure. The letter above each pointer is a page label. The page labels were obtained by labeling the pages of Figure (a) top to bottom, beginning with the label a. The page contents are shown immediately after the page pointer. To see the correspondence between the directory and the trie, notice that if the bits in the directory index are used to follow a path in the trie.

Directory less Dynamic Hashing:

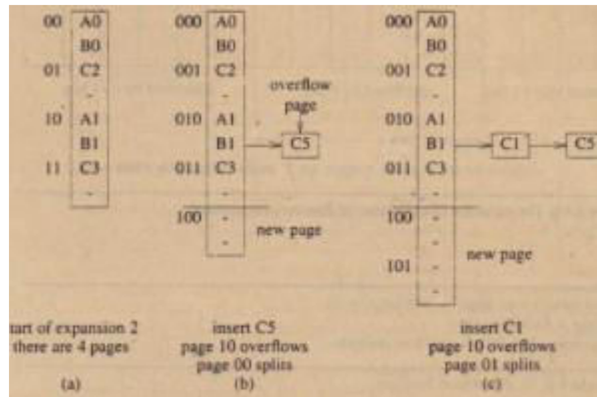
Consider the trie of Figure (a), which has two levels and indexes four pages. In the new method, the two-bit addresses are the actual addresses of these pages (actually they are an offset of some base address). Thus, the hash function delivers the actual address of a page containing the key. Moreover, every value produced by the hash function must point to an actual page. In contrast to the directory scheme, in which a single page might be pointed at by several directory entries, in the directory less scheme there must exist a unique page for every possible address. Figure shows a simple trie and its mapping to contiguous memory without a directory.



What happens when a page overflows? We could double the size of the address space, but this is wasteful. Instead. whenever an overflow occurs, we add a new page to the end of the file and divide the identifiers in one of the pages between its original page and the new page. This complicates the handling of the family of hash functionssomewhat. However, If we had simply added one bit to the result of the hash function, the table size would have to be doubled. By adding only one page, the hash function must distinguish between pages addressed by r bits and those addressed by r + I bits. We will show how this is done in a moment.

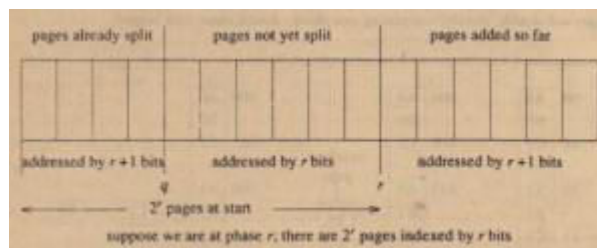
It provides an example of directory less hashing after two insertions. Initially, there are four pages, each addressed by two bits (Figure (a). Two of the pages are full, and two have one identifier each. When C5 is inserted, it hashes to the page whose address is 10 (Figure (b). Since that page is full, an overflow node is allocated to hold C5. At the same time, we add a new page at the end of the storage, rehash the identifiers in the first page, and split them between the first and new page. Unfortunately, none of the new identifiers go into the new page. The first page and the new page are now addressed by three bits, not two as shown in Figure (b). In the next step, we insert the identifier CI. Since it hashes to the same page as C5, we use another overflow node to store it. We add another new page to the end of the file and rehash the identifiers in the second page. Once again none go into the new page. (Note that this is largely a result of not using a uniform hash function.) Now the first two pages and the two new pages are all

addressed using three bits. Eventually the number of pages will double, thereby completing one phase. A new phase then begins.



Consider Figure ,which shows the state of file expansion during the phase at some time q. At the beginning of the rth phase, there are 2′ pages, all addressed by r bits. In the figure, q new pages have been added: To the left of the q line are the pages that have already been split. To the right of the q line are the pages remaining to be split up to the line labeled r. To the right of the r line are the pages that have been added during this phase of the process. Each page in this section of the tile is addressed by r + I bits. Note that the q line is an indicator of which page gets split next. The modified hash function is given in Program 8.6. All pages less than q require r + I bits. The function hash (lcey,r) is in the range (0, 2,-1 I, so if the result is less than q, we rehash using r + I bits. This "gives us either the page to the left of q or the one above 2′ – I. The directory less method always requires overflows.

One sees that for many retrievals the time will be one access, namely, for those identifiers that are in the page directly addressed by the hash function. However, for others, substantially more than two accesses might be required as one moves along the




```
if (hash (key,r) < q)  page = hash (key,r+1);
else page = hash (key,r);
If neccessary, then follow overflow pointers.
```

overflow chain, When a new page is added and the identifiers split across the two pages. all identifiers including the overflows are rehashed.