**UNIT-1**

**Static Hashing:**

**Dictionaries :**

when we want to store some list of items we can use array, linked lists, trees etc. when we want to search an element

* Array performs search operation at the complexity of O(n).
* Linked list performs same operation at the complexity of O(n).
* Binary search tree also performs at the complexity of O(n).
* Balanced binary search tree performs the search operation at the complexity of O(logn).

Is there any way to perform search complexity at O(1)?

The answer is dictionary hash tables gives search complexity O(1).

**Dictionary:**

Dictionary is collection of pairs of the form (k,v) where v is value k is key associated with the value. No two pairs in a dictionary have the same key. In dictionaries each item is associated with key. Basically we have two types of dictionaries.

**Unordered Ordered dictionary:** In unordered dictionaries values are not stored in sorted order. They are stored in the position specified by the key.

**Ordered dictionaries:** In ordered dictionaries the elements are stored in the sorted order. Examples: word dictionary-collection of pairs of the form (word, meaning etc..)

Telephone directory: collection of pairs of the form(subscriber name, telephone number)

Operations on dictionaries:

* Determine whether or not the dictionary is empty
* Determine the dictionary size
* Find the pair with a specified key
* Insert a pair into dictionary
* Delete the pair with a key

**Implementation of dictionaries:**

One of the implementation for unordered dictionary is hashing.

**Hashing**

One of the most efficient way for implementing dictionaries is hashing. hashing is a technique which maps the data items into hash table with the help of hash function. Hashing performs insertion, deletion and search operations are very fast irrespective of the size of the data.

**Hash table:** Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Each location of hash table is referred as bucket. Number of buckets in a table is equal to size of hash table.

**Hash function:**

Hash function maps value into hash table to some index ranges from 0 to size of hash table-1. The value returned by hash function referred as hash value or hash code. We can represent hash function as f(k) or h(k). f(k) is the home bucket for the pair whose key is k. We have several methods for defining hash function the three most common methods are

* Division method.
* Mid Square Metho
* Folding Method
* Universal hashing.

**The division hash function:**

This method of hash function is most common hash function. In hashing by division, the hash function has the form

f(k)=k % N, where N is size of hash table, k is key .

the positions in the hash table are indexed from 0 through N-1. When N=11, the home buckets for the keys 3, 22, 27, 40, 81 and 96 are

f(3)=3%11=3.

f(22)=22%11=0.

f(27)=27%11=5.

f(40)=40%11=7.

f(81)=81%11=4.

f(96)=96%11=9.

0 1 2 3 4 5 6 7 8 9 10

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 22 |  |  | 3 | 81 | 27 |  | 40 |  |  | 96 |

**Search:** to search for the key, first find the location of key, then move to that location, check weather the location is empty or filled. If it is empty then there is no such element in the list. If the location is not empty then compare searching element with the element in that location if they are equal, the element found. Otherwise not found. Suppose we want to search for the element 40. Move to the location f(40)=40%11=7. Compare searching element 40 with the element in 7th location. Since they are equal so element found. So we can perform search operation at the complexity of O(1).

**Deletion:** to perform deletion first we need to search for the element. If the element found delete the element. If the element not found we can’t delete the element. Deletion is also performed at the complexity of O(1).

**Advantage:** Fast, since requires just one division operation.

Disadvantage: Have to avoid certain values of m

**Collisions :**

The above hash table with 11 buckets numbered from 0 through 10, and there is only one key in each bucket. Now suppose we wish to enter key 58 into table. The home bucket is f(58)=58%11=3. This bucket is already occupied by different key. We say that collision has occurred.

A collision is a situation where different keys have same home bucket or same hash value. In these situations we need to define a good hash function which minimises collisions.

**Good hash function**

* Should minimise collisions
* Easy and quick to compute.
* Should distribute the keys evenly in hash table.
* Should use all the information provided in the key

**Making division method a good hash function:**

1. Table size that is power of 2 like 8 , 16 should be avoided to minimise collision.
2. Table size that is power of 10 like10, 20 should be avoided to minimise collision.
3. Better to use the prime number as table size

**Mid Square Method**

Mid square method is a good hash function which works in two steps.

**Step 1:** Square the value of the key. That is, find *k2*

**Step 2:** Extract the middle *r* bits of the result obtained in Step 1.

The algorithm works well because most or all bits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle two digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value. In the mid square method, the same *r* bits must be chosen from all the keys. Therefore, the hash function can be given as, **h (k) = s** where, *s* is obtained by selecting r bits from *k2*

Example: Calculate the hash value for keys 1234 and 5642 using the mid square method. The hash table has 100 memory locations.

Note the hash table has 100 memory locations whose indices vary from 0-99. this means, only two digits are needed to map the key to a location in the hash table, so r = 2.

When k = 1234, k2 = 1522756, h (k) = 27   
When k = 5642, k2 = 31832164, h (k) = 21

Observe that 3rd and 4th digits starting from the right are chosen.

**Folding Method**

The folding method works in two steps.

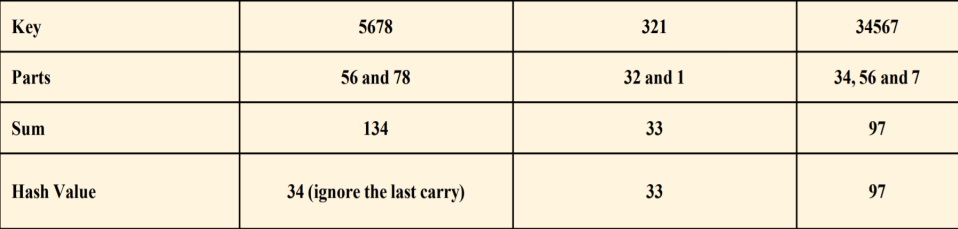
**Step 1:** Divide the key value into a number of parts. That is divide k into parts, k1, k2, …, kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

**Step 2:** Add the individual parts. That is obtain the sum of k1 + k2 + .. + kn. Hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000. Then it means there are 1000 locations in the hash table. To address these 1000 locations, we will need at least three digits, therefore, each part of the key must have three digits except the last part which may have lesser digits.

Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys- 5678, 321 and 34567.

Here, since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. Therefore,



**Universal Hashing:**

Let *U* be the set of universe keys {0,1,2..........m-1} and H be a finite collection of hash functions. Then H is called *universal* if, for *x*, *y* ϵ *U*, (*x* != *y*), jf

| {*h* ϵ H : *h*(*x*) = *h*(*y*)}|≡1/m

In other words, the probability of a collision for two different keys *x* and *y* given a hash function randomly chosen from H is 1/*m*.

**Collision resolution techniques:**

Recall that the main idea of hash table is to take a bucket array A, and a hash function f, and use them to implement a dictionary by storing each item (k,e) in the bucket A[f(k)]. This simple idea is challenged, however, when we have two distinct keys k1, k2 such that f(k1)=f(k2). The collisions prevents us from simply inserting the new item (k,e) in the bucket A[f(k)]. Thus must have consistent strategy for resolving collisions. We have several methods for collision resolution.

* Open hashing
* Separate chaining
* Closed hashing
* Open addressing
* Linear probing
* Quadratic probing
* Double hashing
* Rehashing

**Separate chaining:**

A simple and most efficient collision resolution technique is separate chaining. It stores the keys mapped into the same location A[i] as linear or linked list which is headed by A[i]. that means A[i] is a head pointer to the list. List Si can be implemented using unordered sequence or log file method. This rule is called separate chaining. An insertion will put the new item at the end of the A[f(k)] list and the search will go through this list until it reaches to the end or finds an item with the desired key, and a deletion will remove an item in this list after it is found.

***see the examples in your note book***

**Advantages:**  
1) Simple to implement.  
2) Hash table never fills up, we can always add more elements to chain.  
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

**Disadvantages:**  
2) Wastage of Space (Some Parts of hash table are never used)  
3) If the chain becomes long, then search time can become O(n) in worst case.  
4) Uses extra space for links.

**Open addressing:**

One of the disadvantage of separate chaining is it requires the use of auxiliary data structure a list to hold the colliding keys. Another alternative technique which does not require auxiliary structure is open addressing but it requires bit more complexity to deal with collisions. There are several methods for implementing this approach and those are collectively called as open addressing schemes in which the items are stored only in the bucket array itself.

**Linear probing:**

A simple open addressing strategy is linear probing. In this, if a key k is mapped into the bucket A[i] that is already occupied, where i=f(k), then we try next at A[(i+1) modN]. If A[(i+1) mod N] is also occupied, then we try at A[(i+2) mod N], and so on, until we find the empty bucket in A or we reach the position where we start. Linear probing technique is illustrated in the following figure for the elements 13, 26, 5, 37, 16, 15, 21

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 13 |  | 26 | 5 | 37 | 16 | 15 |  | 21 |

Here f(k)=k mod 11

To perform search, we must examine consecutive blocks, starting from A[f(k)], until we either find the item or we find the empty bucket.

The remove operation is more complicated than this. To implement this function the contents of the bucket array are restored after every removal. To perform such restoration, it requires that, we shift items down in bucket above A[i], while not shifting other in this group. Another way to handle this is place the special marker so that we can detect the previous deletion.

Linear probing saves the space but it complicates the removals. Another disadvantage of linear probing is it forms cluster of items into contiguous runs, which causes searching more complex.

**Quadratic probing:**

Another open addressing strategy is quadratic probing, involves iteratively trying bucket A[(i+f(j)) mod N], for j=1, 2, 3,…….., where f(j)=j2 until empty bucket is found. It avoids kind of clustered patterns formed in linear probing. But it creates its own clusters which are called secondary clusters. Another disadvantage is, even if N is prime it may not find an empty block if the bucket array is at least half full.

*See the examples in the note book*

**Double hashing:**

In this strategy we choose secondary hash function h’. when a key maps to A[i], and it is occupied, then we iteratively try the buckets A[(i+f(j)) mod N] next, for j=1, 2, 3………, where f(j)=j.h’(k). A common choice is h’(k)=q-(k mod q) for some prime number q < N. Moreover we choose secondary hash function to minimise clusters.

*See the examples in the note book*

**Load factor and Rehashing:**

The load factor ƛ can defined as n/N, where n no.of elements in the hash table and N is size of hash table. Some experiments and average case analyses suggests that we should keep ƛ<0.5 for open addressing schemes and ƛ<0.9 for separate chaining.With separate chaining, as ƛ gets close to 1 probability of collisions approaches to 1, which adds overhead to our operations. Same situation occurs in open addressing when ƛ grows beyond 0.5.

**Rehashing into new table:**

Keeping load factor below a certain threshold is vital in open addressing and separate chaining methods. If load factor of hash table goes above a threshold, then it is common to require that the table resized, it is good requirement that the size of the array should be double the previous size. Once we allocated new bucket, we must define new hash function. Given this new hash function we then reinsert every item from old to new table using new hash function. This process is known as rehashing.

*See the examples in the note book*

**Dynamic Hashing:**

**Dynamic Hashing using Directory Structure:**

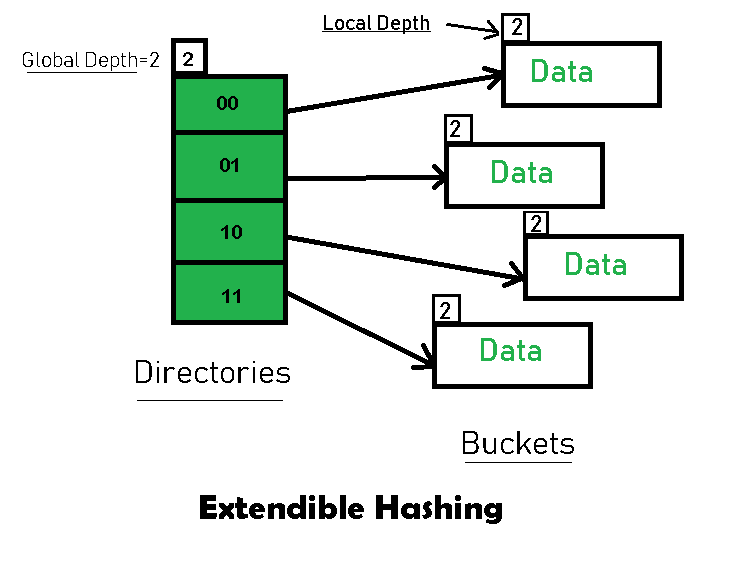
Dynamic hashing is also called extendible hashing. In this method directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

* Motivation
  + Limitations of static hashing
    - When the table is to be full, overflows increase. As overflows increase, the overall performance decreases.
    - We cannot just copy entries from smaller into a corresponding buckets of a bigger table.
    - The use of memory space is not flexible.
* Properties of Dynamic Hashing
* Allow the size of dictionary to grow and shrink.
  + The size of hash table can be changed *dynamically*.
  + The term “dynamically” implies the following two things can be modified:
    - Hash function
    - The size of hash table

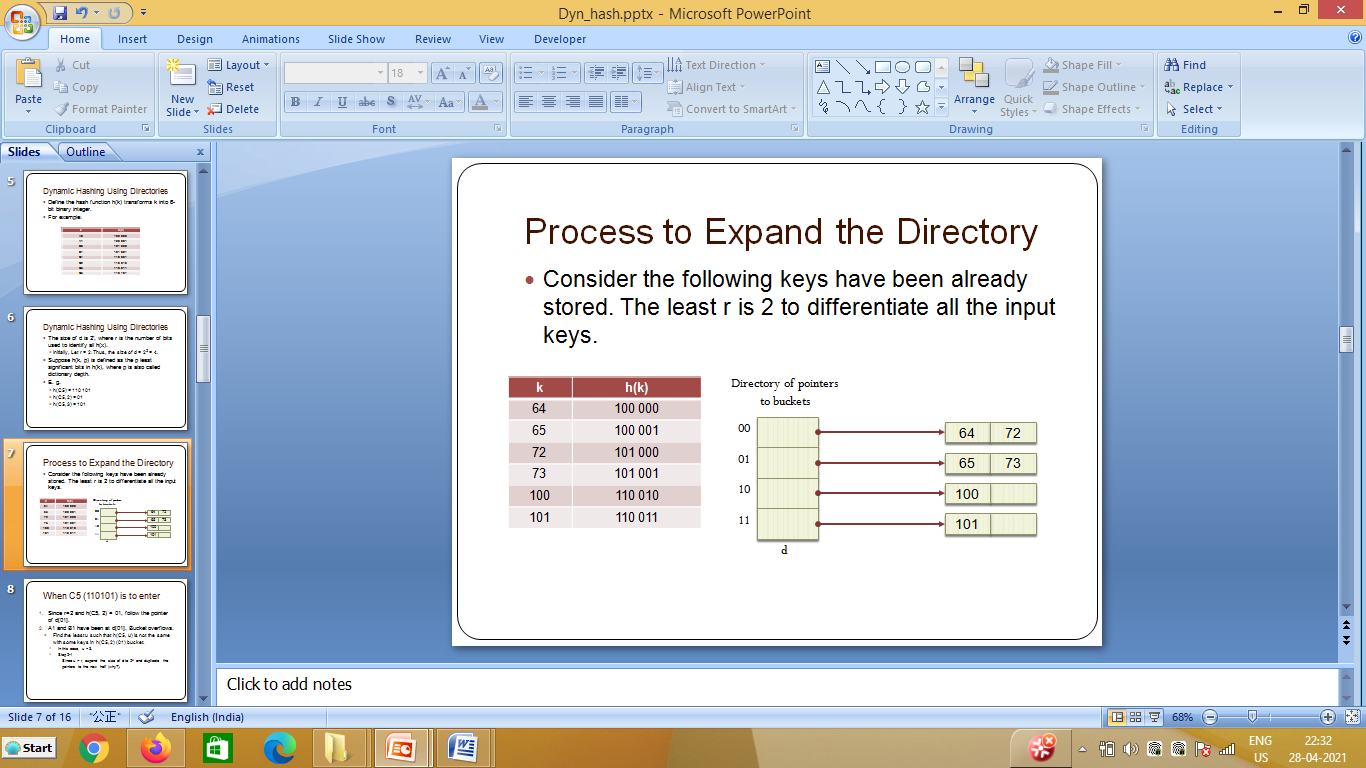
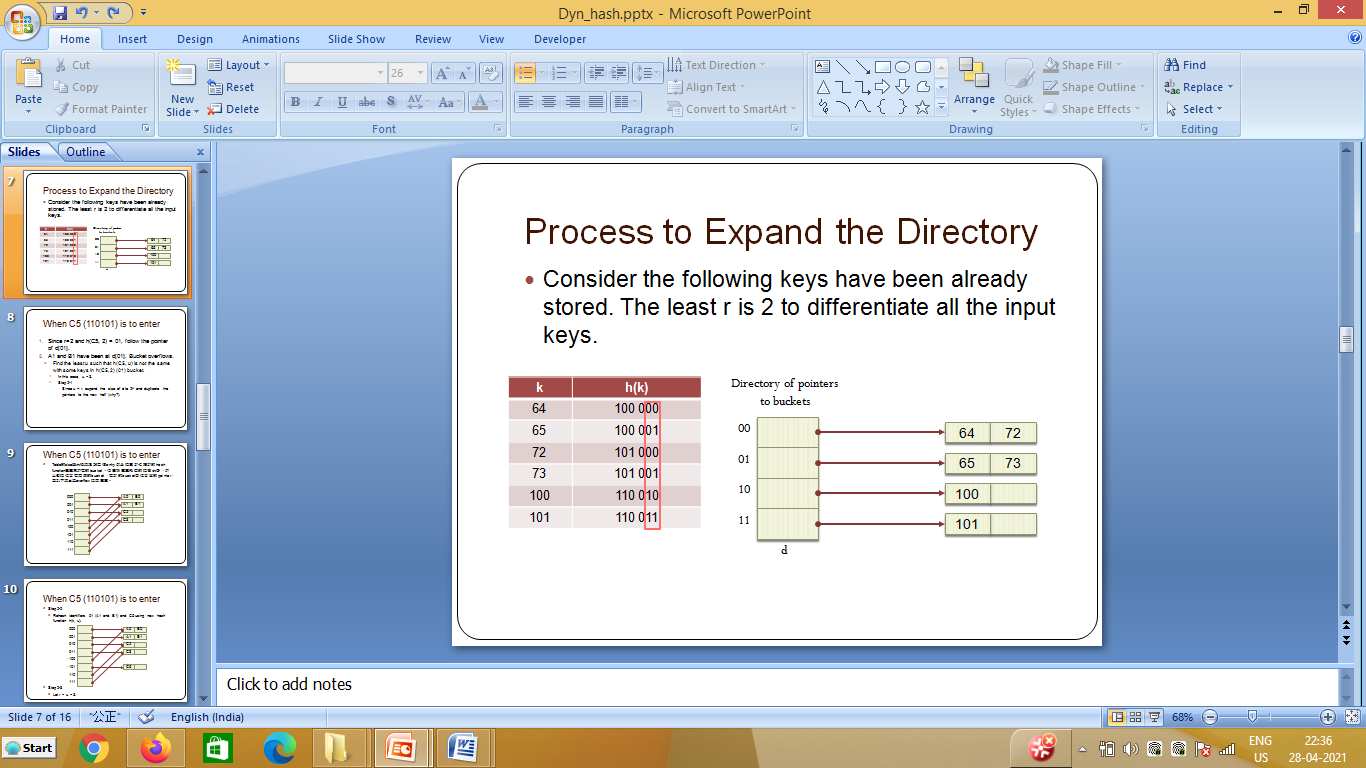
**Main features of Extendible Hashing:** The main features in this hashing technique are:

* **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
* **Buckets:** The buckets are used to hash the actual data.

**Basic Structure of Extendible Hashing:**



**Frequently used terms in Extendible Hashing:**

* **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
* **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.
* **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
* **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.
* **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
* **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.
* **Hash Function in Dynamic Hashing:**
* Define the hash function h(k) transforms k into 6-bit binary integer. For example:
* 
* Suppose h(k, p) is defined as the p least significant bits in h(k), where p is also called dictionary depth.
* E. g. h(110101, 2) = 01
* Consider the following keys have been already stored. The least p is 2 to differentiate all the input keys.

**Basic Working of Extendible Hashing(Insertion):**

* **Step 1 – Analyze Data Elements:** Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
* **Step 2 – Convert into binary format:** Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.
* **Step 3 – Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.
* **Step 4 – Identify the Directory:** Consider the ‘Global-Depth’ number of LSBs in the binary number and match it to the directory id.   
  Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.
* **Step 5 – Navigation:** Now, navigate to the bucket pointed by the directory with directory-id 001.
* **Step 6 – Insertion and Overflow Check:** Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.
* **Step 7 – Tackling Over Flow Condition during Data Insertion:** Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.   
  First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.
  + **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.   
    Directory expansion will double the number of directories present in the hash structure.
  + **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.

*See the examples in the note book*

**Directory less Dynamic Hashing:**

* This method also is known as linear dynamic hashing, in which we do not have the directory, *d, of bucket pointers.*
* Instead, an array, *ht, of buckets is used.*
* We assume that this array is as large as possible and so there is no possibility of increasing its size dynamically.
* To avoid initializing such a large array, we use two variables *q and r, 0 <=q <* 2r*, to keep track of the active buckets.*
* *At any time, only buckets 0 through* 2r *+ q are active.*
* The remaining buckets on a chain are called *overflow buckets.*
* *Informally, r is the number bits of h (k) used to index into the hash table and q is the bucket that will split next.*
* An overflow is handled by activating bucket 2r + *q;*
* Reallocating the entries in the chain *q between q and the newly activated bucket (or chain)* 2r *+ q,* and incrementing *q by 1. In case q now becomes* 2r*, we increment r by 1 and reset q to* O
* The reallocation is done using *h (k, r + 1).*

*See the examples in the note book*