Hash Tables

In an ideal situation, the hash table data structure will be O(1) for search, insert and delete operations

**Concept**

* Problem: how do we design a data structure to store n records of key-value pairs, ex. Name, score for efficient search by key
* Example:
  + Use array, the search time is O(n) in worst case
  + Use AVL tree, the search time is O(log n) in worst case
* What if the keys are of integer values from 0 to n-1? Yes, we can use an array of n elements to store the values, find the value of key I at position I of the array. That has O(1) time for search

**Hash Table example**

Input: given data records:

Myrie, 76.7

Ali, 88.0

Peter, 82.5

Smith, 60

Problem: Design a hast table to store the record data such that search by name can be done in constant time.

1. Define structure data type RECORD

Typedef struct{

Char name[20];

Float score;

}RECORD;

1. Create an array of RECORD type of length bigger than the record number

RECORD ht[7] = {0};

1. Use the name filed as key and define a hash function

Int h(char \*str){

If(str==NULL) return -1;

Else

Return \*str % 7;

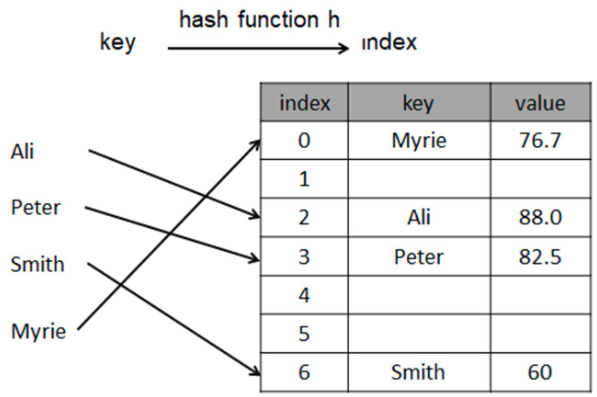
}

1. Store record (name, score) at index position h(name)

For example, record (“Ali”,88.0), h(“Ali”) = 65%7=2 , is stored at ht[2] using statements:

Strcpy(ht[2].name, “Ali”);

Ht[2].score = 88.0;



1. Search by name is to look up record at index position

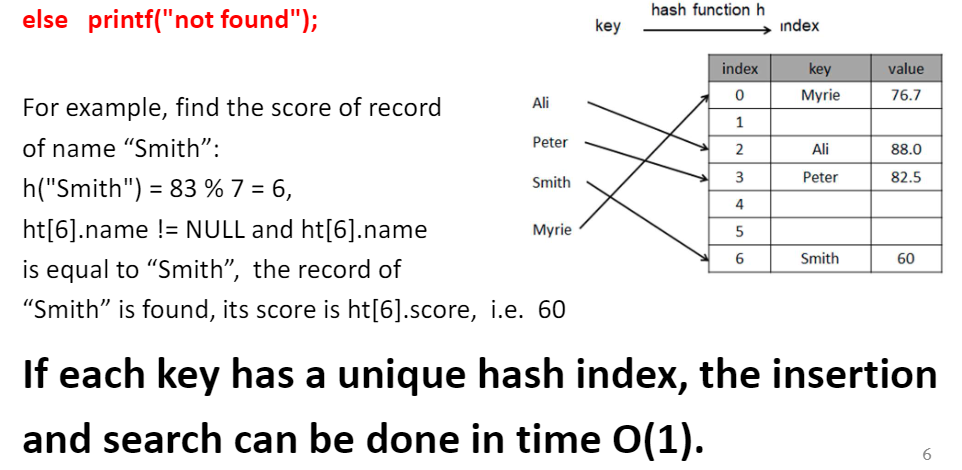
Ht[hash(name)]

Int I = hash(name\_str);

If(ht[h[i].name!=NULL&&strcmp(ht[i].name,name\_str)==0)

Printf(“(found(%s,%f)”, ht[i].name, ht[i].score);

Else printf(“not found”);



**Definition of Hash Tables**

An abstract hash table (or simply hash) is an abstract data structure with the following properties

1. It stores a collection of data values of a certain type, say hash\_data\_type. A part of the data type is used as key, the others (if any) are used as value, so hash\_data\_type can be viewed as a key-value pair (key, value).
2. An array of hash\_data\_type of length m, hash\_data\_type hash\_array[m], is used to store the data records. Where m is called the size of the hash table.
3. A function h, called hash function, is used to map a key k to an integer h(k) between 0 and m-1. H(k) is called hash value of k, or index of k.
4. Insertion puts key-value pair (k,x) at hash\_array[h(k)]. If the array position is taken, a resolved alternate position is used.
5. Search an element by key k (called look-up) is to check data element hash\_array[h(k)]. If the key value of hash\_array[h(k)] matches with k, then return hash\_array[h(k)], otherwise check the data element at resolved alternative positions.
6. Delete an element by key k to remove the data element of key k at hash\_array[h(k)].

* A hash table is a data structure that implements an abstract hash table with concrete hash\_data\_type, hash table size, hash function h, as well as insert, search (look-up), and delete operations.

**Application of hashing tables**

1. Store data records for fast search. Hash tables are used to store massive amounts of information
2. Symbol tables are used in compilers to hold variable names and their values. Ex. In C/C++ compiler, hash tables are used to keep a record of the user-defined symbol in a C++ program. Hashing facilitates the compiler to quickly look up variable names and other attributes associated with symbols.
3. Caching which saves information in memory, the other is hashing which makes looking up the file location in memory much quicker than most other methods.
4. Hash is used for database indexing. DBMS store a separate file known as indexes. When data has to be retrieved from a file, the key information is first found in the appropriate index file which references the exact record location of the data in the database file. This key information in the index file is often stored as a hashed value.

**Hash function and collision**

* Hash function, h, a mathematical function which maps a key k in set U to an integer h(k) of given modular m. h(k) is called the hash value of k by h.



Ex. U is the set of integers. Let m=10, h(k) = k mod m, k=18, h(k)=8

Ex. 2 string hash function

Int hash (char\* word, int m){

Unsigned int hash = 0, i;

For(i=0, word[i] != ‘\0’;i++){

Hash+=word[i];

}

Return hash % m;

}

**Concept of collisions**

* If two keys have the same hash value by the hash function, they are said to be collided, or a collision happens on the keys by the hash function.

Ex. Given hash function m=10, h(k) = k mod . h(18) = h(28) = 8, so 18 and 28 are collided by the hash function.

* Ideal situation, namely no collisions with the given hash function and set of keys. In such a situation, the insert, search and delete operations can be done in O(1) time.

Collision is inevitable. Choose proper size m and hash function

* The size of hash table is usually chosen bigger than the number of records, i.e. 15% more, a trade-off between space and performance
* A good hash function is efficient to computer hash values for given keys. Has less number of collisions with considered key set.

**Simple hash function – Division method**

Division method uses hash function h(x) = x mod m

The division method is efficient to compute

How to choose m?

* Generally, it is best to choose m to be a prime number, m should also not be too close to exact powers of 2

Ex. M=97, calculate hash values of keys 1234 and 5462

H(1234) = 1234%97 = 70

H(5642) = 5642%97 = 16

**Drawback of the division method**

1. Consecutive keys ap to consecutive hash values. This ensure that consecutive keys do not collide
2. Consecutive array locations will be occupied. This may lead to degradation in performance.

**Collision handling**

Hash table collision problem: two keys have the same hash value corresponding to one array location, how to store the values, namely how to resolve the hash table collision?

Solution 1: when a collision happens, find an open position by probing, and store the data at the open position

How to probe?

1. Linear probing
2. Quadratic probing
3. Double hashing

Solution 2: use an alternative data structure to store all data records of the same index value.

**Linear probing (final exam topic)**

When a collision happens on insertion, find the next open position to store new data by linear probing:

If a value is already stored at location generated by h(k), then use the following hash function to find an open position

H(k,i) = [h(k) + i] mod m

Where, m is the size of the hash table, and I is the probe number and varies from 0 to m-1. Find the first i such that position h(k,i) is open.

How do we know array position is open? Values are stored in the hash table. The hash table will contain two types of values – either sentinel value (for example -1) or a data value. The presence of a sentinel value indicates that the location contains no data value at present by can be used to hold a value.

**Example**

Consider a hash table of size =10

hash function h(k) = k mod m

linear probing hash function: h(k,i) = ((k mod m) + i) mod m;

using linear probing to insert the keys 72, 27, 36, 24, 63, 81, and 92 into the table:

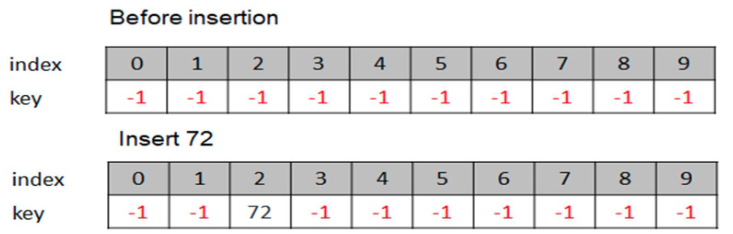
int T[m]; int I; for(i=0;i<m;i++) T[i]=-1;

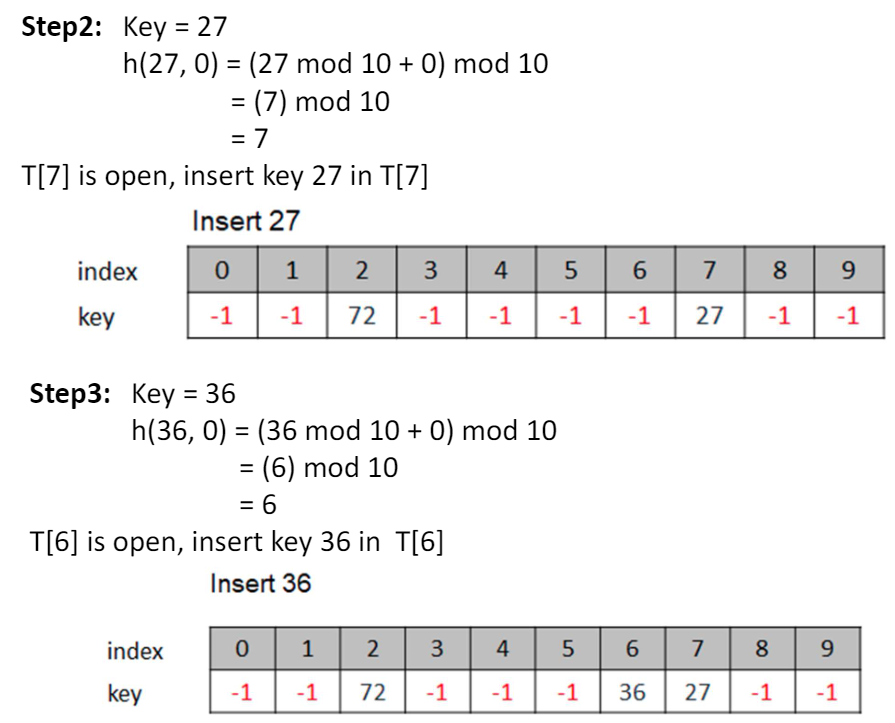
1. Key = 72

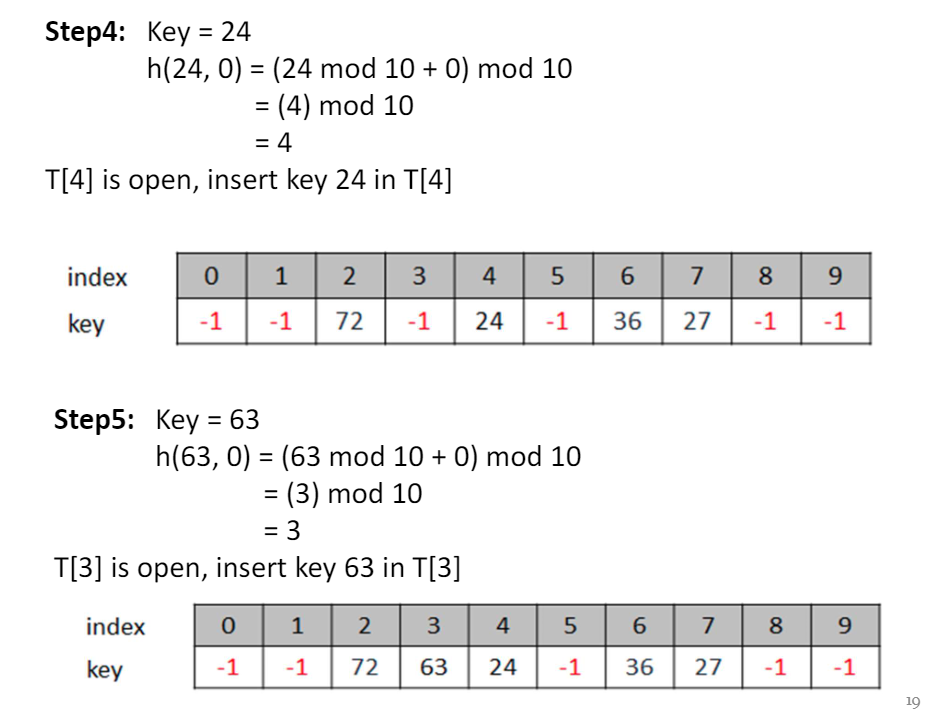
H(72,0) = (72 mode 10 + 0) mod 10

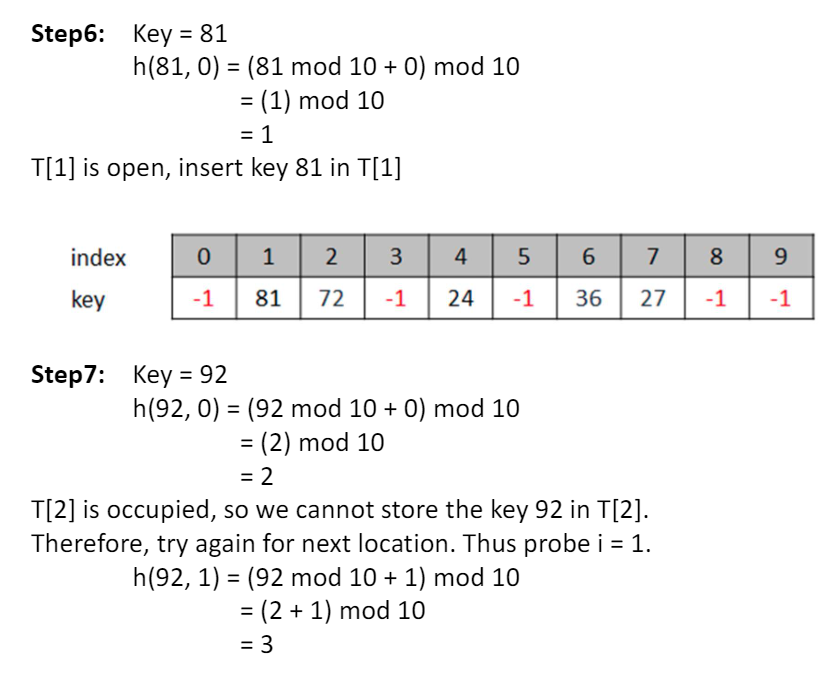
=2 mod 10 = 2

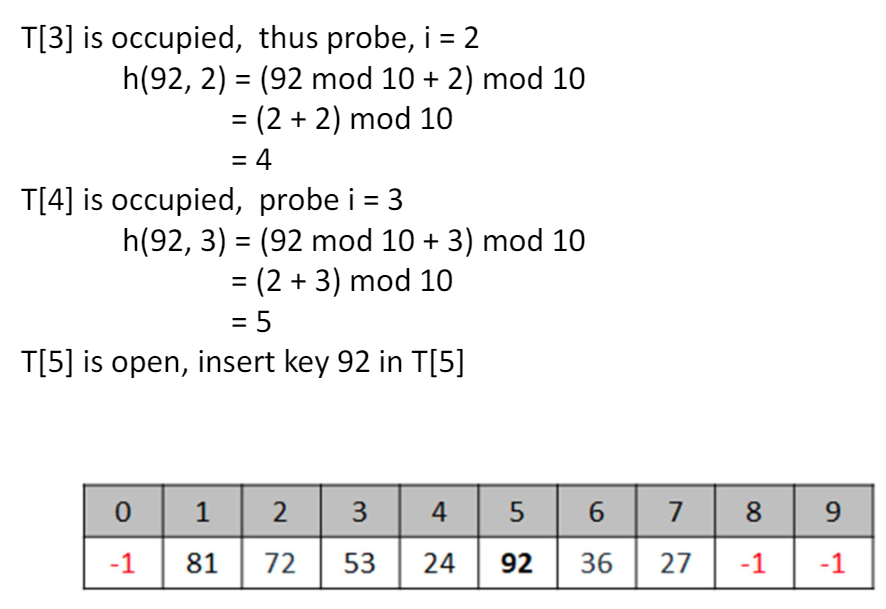
T[2] is open, insert key 72 at T[2]











Algorithm: insert hash table by linear probing

Input: hash table ht[m], key-value pair (k,val)

1. Set i=0 and j=s = h(k,i)
2. If ht[j] has key value -1

Store (k,val) at ht[j]

Goto step 5

1. Set I = i+1, j = h(k,i)

If j==s

Goto step 4

Else

Goto step 2

1. Output -1 (hash table is full, overflow, insertion failed)
2. Output j (return the inserted position)

Time: O(m), space O(1)

**Search by linear probing**

Solution:

When searching a value in the hash table, compute the hash value of the search key, check the data at the hash value position. If a match is found, then the search operation is successful. The search time in this case is given as O(1).

Otherwise, if the key does not match, then check the next index position of linear probing until the value is found or the search function encounters a vacant location in the array, indicating that the value is not present.

Algorithm: search hash table by linear probing

Input: hash table ht[m], and key value k

1. Set i=0, j = s = h(k,i)
2. If k matches the value value of ht[j]

Goto step 4

1. Set I = i+1, j = h(k,i)

If j == s or ht[j] == -1

Goto step 5

Else

Goto step 2

1. Output ht[j];
2. Output NULL (not found)

Time O(m), space O(1)

**Pros and cons of linear probing**

* Pros
  + Linear probing finds an open location by doing a linear search in the array beginning from position h(k)
  + The algorithm provides good memory caching, through good locality of reference
* Cons
  + It results in clustering, and this a higher risk that where there has been one collision there will be more
  + The performance of linear probing is sensitive to the distribution of input values

**What do we do when a hash table is full?**

Solution: rehashing

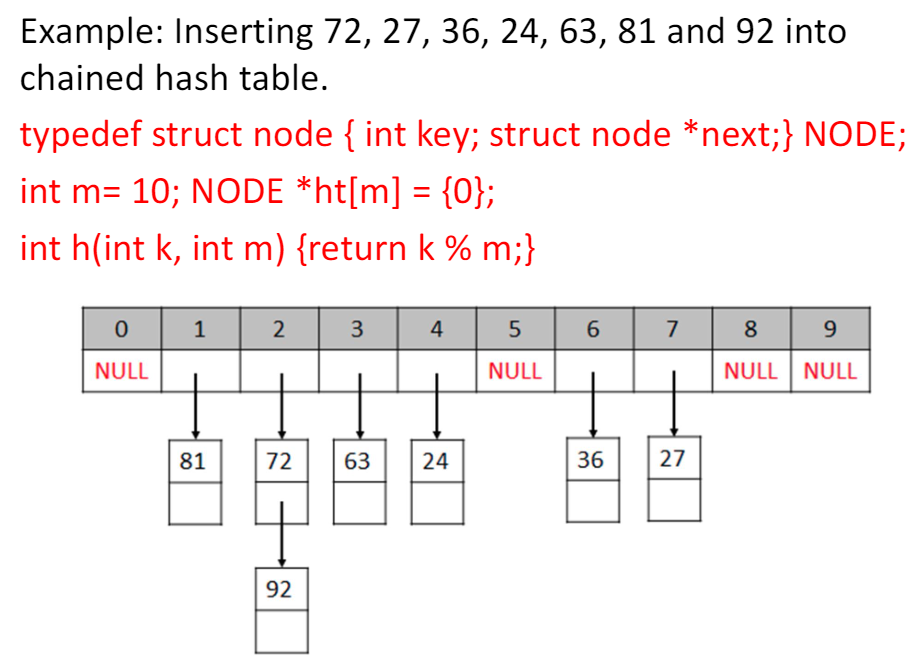
1. Create a new hash table of bigger size ex. 2m
2. For each data record in the current hash table, insert it to the new hash table
3. Clean the old hash table

**Collision resolution by chaining**

* Resolving collision by chaining is to use an alternative data structure (such as linked list, BST< AVL) to store all data records of the same hash value. So insert, search, delete operations are just to do the operation on the corresponding data structure of the hash value.
* Specifically, the hash table array ht[m] stores pointers of linked data structures. Ht[i] holds the pointer to the data structure storing all key-value data records of the same hash index value i.

**Linked hash tables**

* If a linked data structure is used to store all key-value record of the same hash index, we call it a linked hash table. It is a hybrid data structure which contains an array and m linked data structures.
* If a linked list data structure is used to store data records of the same hash index, it is called chained hash table.
* The insert, search, and delete operations on linked hash table mainly have two steps
  + Step 1. Computer the hash index value of the given key
  + Do insert, search, or delete operation on the associated linked data structure at the hash index position.



**Pros and cons of chained hash tables**

* Pros
  + It remains effective even when the number of records to be stored is bigger than hash table
  + The performance does not degrade when the table is more than half full, efficient to handle collisions
* Cons:
  + There is space overhead for using linked data structure
  + Traversing a linked list has poor cache performance, making the processor cache ineffective
  + With the increase in number of records to be stored, the performance of chained hash table does not degrade gracefully (linearly)), O(n/m) on average.

**Java HashMap**

* Java’s hashtable, hashmap, hashset are internally based on linked hash table data structures; Array of references (called buckets) is used, each bucket holds the key-value pairs of the same hash value of keys, and is implemented by linked list.

Size – number of data elements

Capacity – the maximum length of array

Load factor (LF) = size / capacity

When LF >=0.75, i.e. 3 quarters of its capacity then

* Double the array capacity by creating a new array of the new capacity, followed by rehashing. Rehashing is to insert each key-value pair data element of the old hash table to the new hash table. Use insert at the beginning of each linked list of bucket.

**Multiplication Method**

1. Choose a constant A such that 0 < A < 1
2. Multiply the key k by A
3. Extract the fractional part of kA
4. Multiple the result of step 3 by m and take the floor

Hence, the hash function can be given as, h(k) = floor(m(kA mod1))

Where, kA mod 1 gives the fractional part of kA and m is the total number of indices in the hash table

The greatest advantage of the multiplication method is that it works practically with any value of A. Better to choose A = (sqrt(5)-1)/2 = 0.6180339887

Example: Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.

We will use A = 0.618033, m=1000 and k =12345

H(12345) = floor(1000(12345 x 0.618033 mod 1))

= floor(1000 (7629.617 mod 1))

= floor (1000(0.617385))

=617.385

=617

**Mid square Method**

1. Square the value of the key. That is, find k2
2. Extract the middle r bits of the result obtained in step 1.

Example: Calculate the hash value for keys 1234 and 5642 using the mid square method. M=100. Hash value has two digits, so r = 2.

When k=1234, k2 = 152**27**56, h(k) = 27

When k = 5642, k2 = 3183**21**64, h(k) = 21

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

This 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.

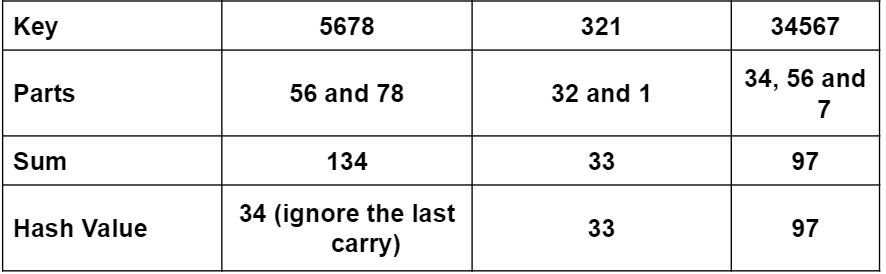
**Folding Method**

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.
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.

The number of digits in each part of the key depends on the size of the hash table. For example, if the hash table has a size of 1000. It needs at least 3 digits.

Example: Given a hash table size m = 100. 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 2 digits.



**Advanced Probing**

**Quadratic Probing**

In this technique, if a value is already stored at location generated by h(k), then the following hash function is used to resolve the collision.

H(k,i) = [h’(k)+c1\*i+c2\*i2] mod m

Where, m is the size of the hash table, h’(k) = k mod m and I is the probe number that varies from 0 to m-1 and c1 and c2 are constants such that c1 and c2 do not equal 0.

Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search. For a given key k, first the location generated by h’(k) mod m is probed. If the location is free, the value is stored in it. Else subsequent locations probed are offset by factors that depend in a quadratic manner on the probe number i. Although quadratic probing performs better than linear probing but to maximize the utilization of the hash table, the values of c1, c1 and m needs to be constrained.

Example: Consider a hash table with size = 10. Using quadratic probing insert the keys 72, 27, 36, 24, 63, 81, 101 into the table. Take c1 = 1 and c2 = 3. H’(k) = k mod m, m=10

Initially the hash table can be given as,



We have, h(k,i) = [h’(k) + c1\*I + c2\*i2] mod m

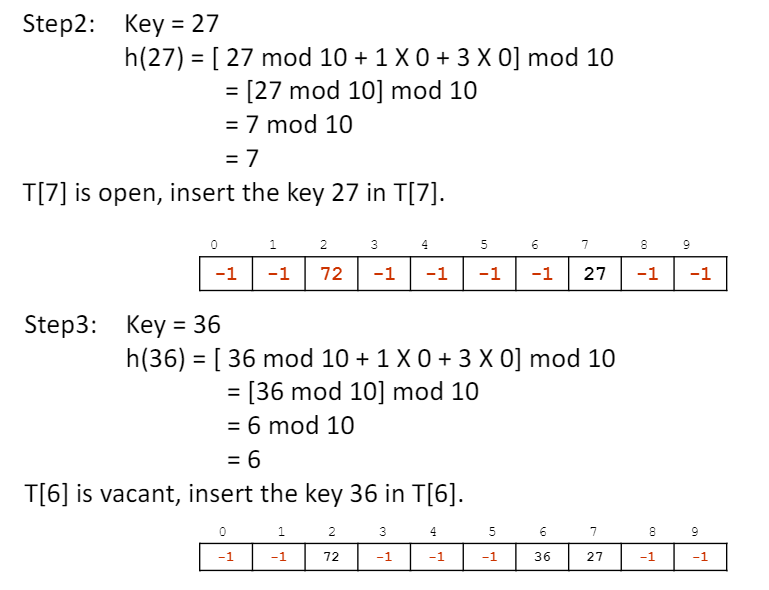
1. Key = 72

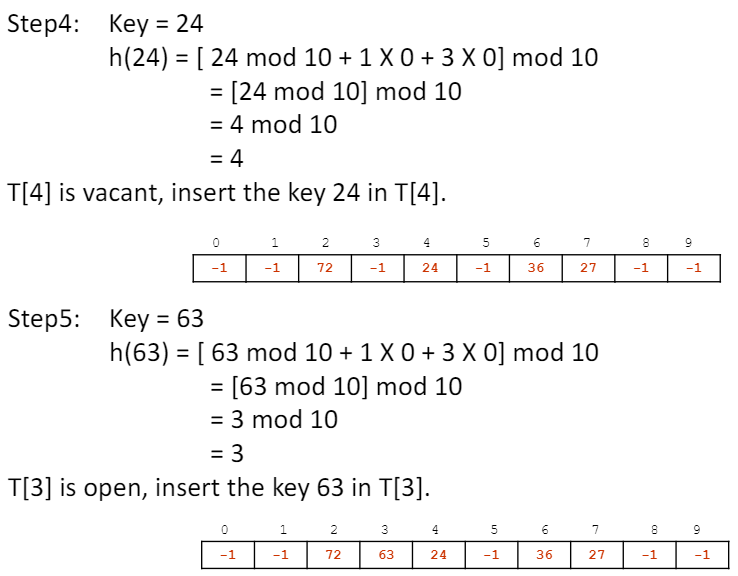
H(72) = [72 mod 10 + 1x0 + 3 x 0] mod 10

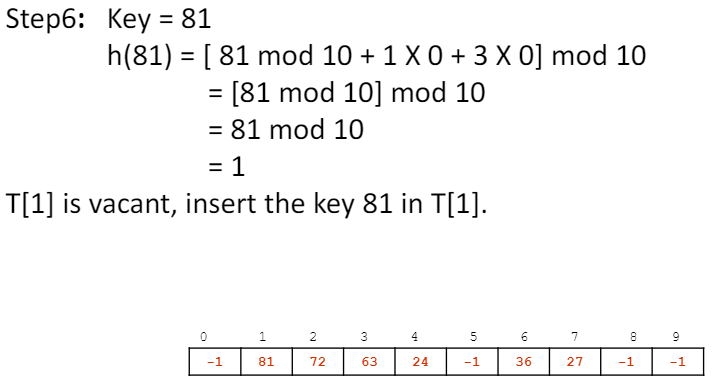
= 2

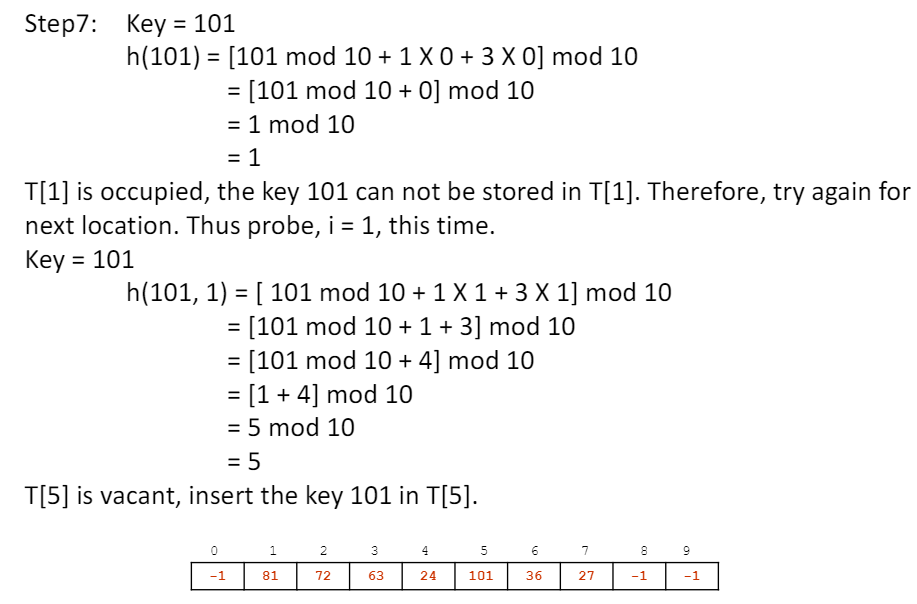
T[2] is open, insert the key 72 in T[2]











**Pros and Cons of Quadratic Probing**

* Quadratic probing caters to the primary clustering problem that exists in linear probing technique. Quadratic probing provides good memory caching because it preserves some locality of reference. But linear probing does this task better and gives better cache performance.
* One of the major drawbacks with quadratic probing is that a sequence of successive probes may only explore a fraction of the table, and this fraction may be quite small. If this happens then we will not be able to find an empty location in the table despite the fact that the table is by no means full.
* Although quadratic probing is free from primary clustering, it is still liable to what is known as secondary clustering. This means that if there is a collision between two keys then the same probe sequence will be followed for both. With quadratic probing, potential for multiple collisions increases as the table becomes full. This situation is usually encountered when the hash table is more than half full.
* Quadratic probing is widely applied in the Berkeley fast file system to allocate free blocks.

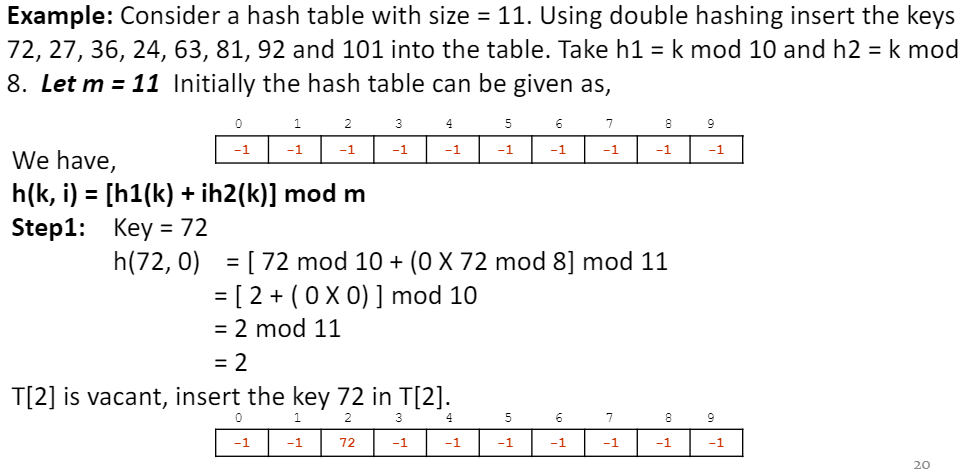
**Searching a value using quadratic probing**

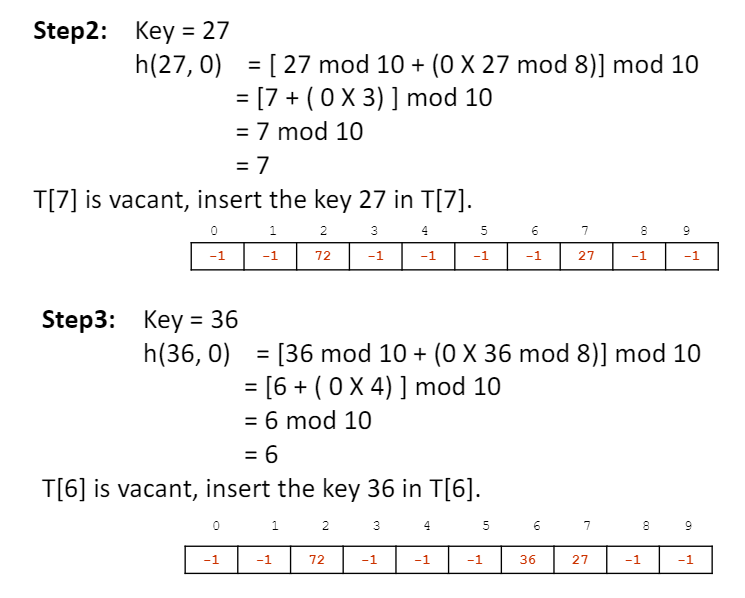
While searching for a value using quadratic probing technique, the array index is re-computed and the key of the element stored at the location is checked with the value that has to be searched. If the desired key value matches the key value at that location, then the element is present in the hash table and the search is said to be successful. In this case the search time is given as O(1). However, if the value does not match then, the search function being a sequential search of the array that continues until:

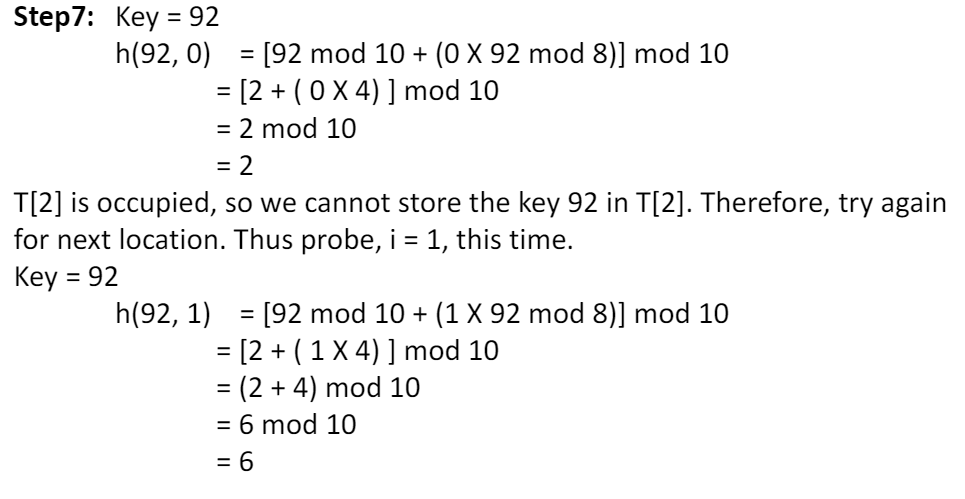
* The value is found
* The search function encounters a vacant location in the array, indicating that the value is not present
* The search function terminates because the table is full and the value is not present

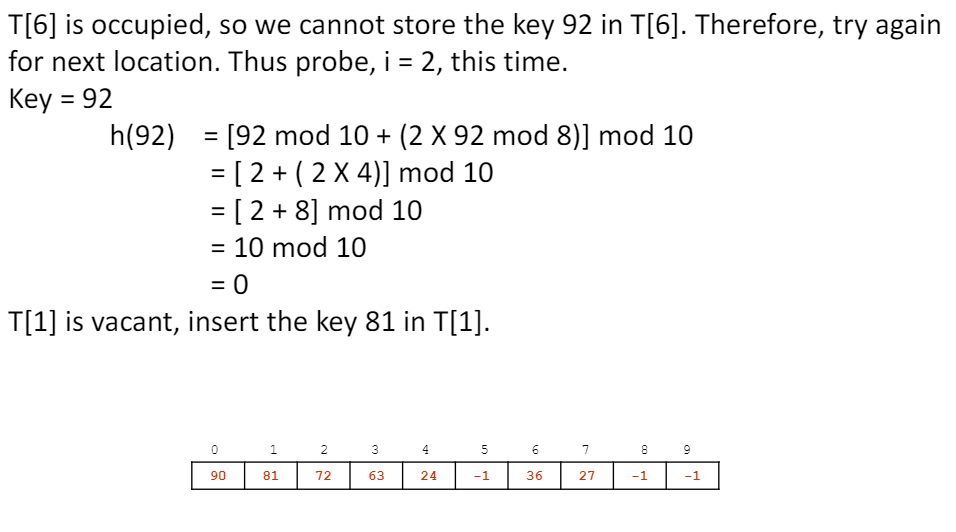
**Double Hashing**

* To start with double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval us decided using a second, independent hash function, hence the name double hashing. Therefore, in double hashing we use two hash function rather than a single function. The hash function in case of double hashing can be given as **h(k,i) = [h1(k) + ih2(k)] mod m**
* Where, m is the size of the hash table, h1(k) and h2(k) are two hash functions given as, h1(k) = k mod m, h2(k) = k mod m’, I is the probe number that varies from 0 to m-1 and m’ is chosen to be less than m. We can choose m’ = m-1 or m-2.
* When we have to insert a key k in the hash table, we first probe the location given by applying h1(k) mod m because during the first probe, i=0. If the location is vacant the key is inserted into it else subsequent probes generate locations that are at an offset of h2(k) mod m from the previous location. Since the offset may vary with every probe depending on the value generated by second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.
* Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well as secondary clustering.









The above is wrong but you get the idea…