Searching

By Asst Prof Christopher Uz

Course: DSAA

Pillai College of Engineering

Introduction to Searching

- Searching means to find whether a particular value is present in an array or not
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array
- However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful
- There are two popular methods for searching the array elements:
  - Linear search
  - Binary search

Linear Search (Sequential Search)

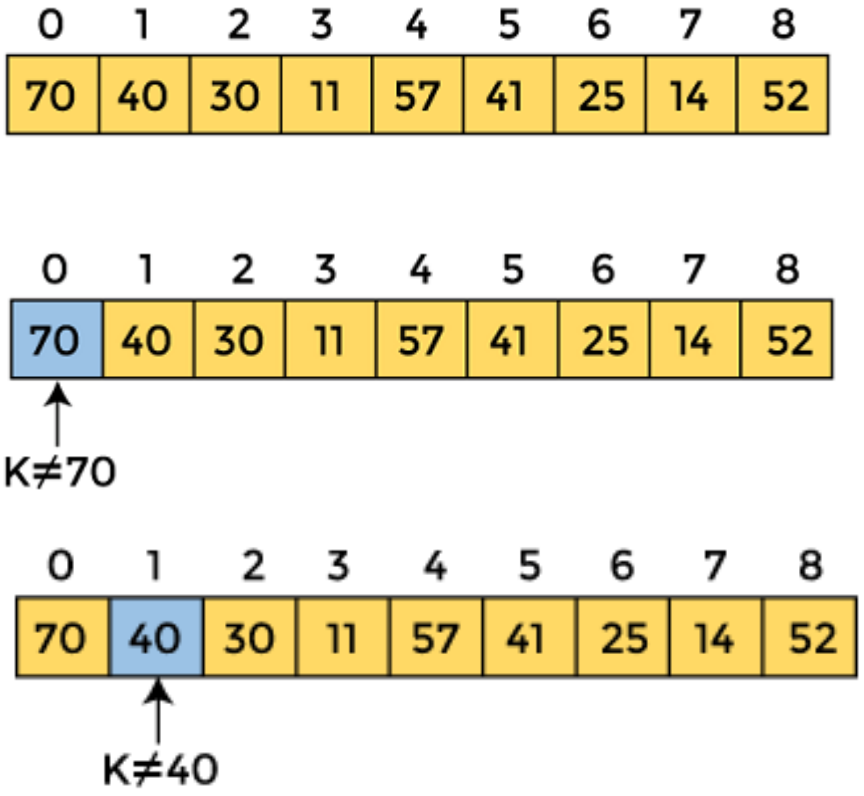
- Linear search works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found
- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted)
- If all the array elements have been compared and no match is found, then it means that the value is not present in the array
- Linear search executes in O(n) time where n is the number of elements in the array

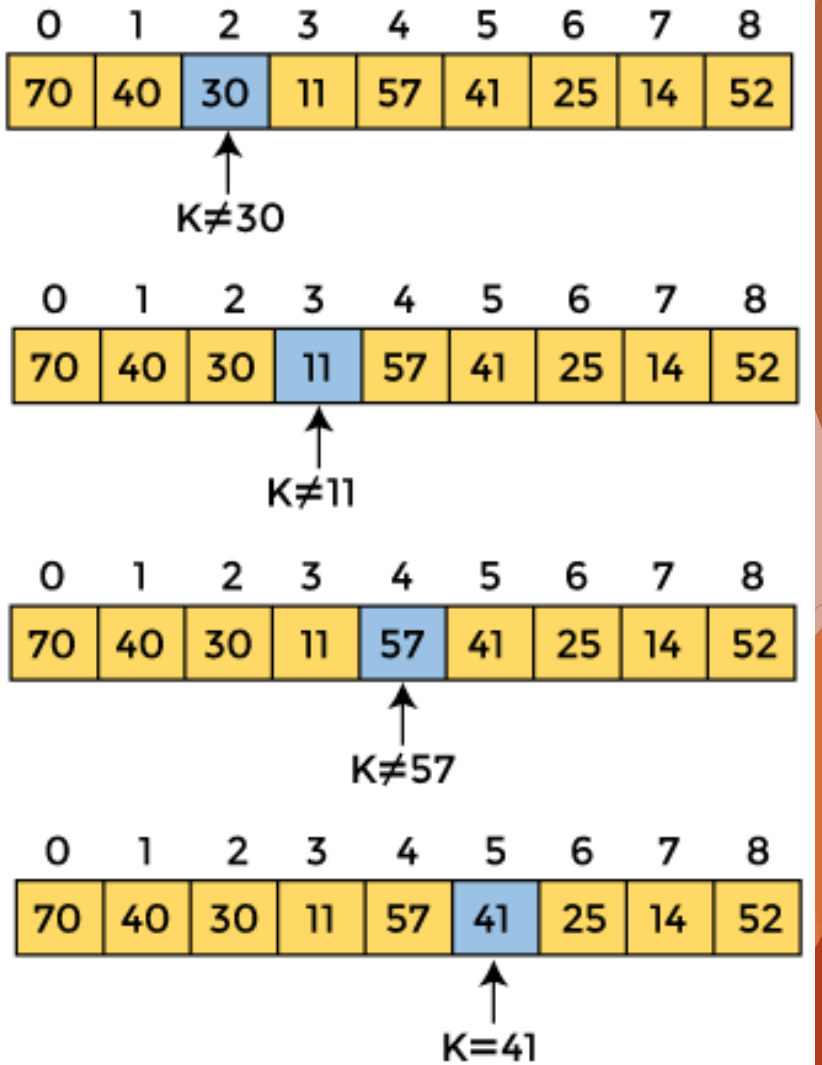
Linear Search Algorithm

- *Step 1* - Read the search element from the user
- *Step 2* - Compare the search element with the first element in the list
- *Step 3* - If both are matched, then display "Given element is found!!!" and terminate the function
- *Step 4* - If both are not matched, then compare search element with the next element in the list
- *Step 5* - Repeat steps 3 and 4 until search element is compared with last element in the list
- *Step 6* - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function

Linear Search Example

- Linear search from array where K = 41





Binary Search

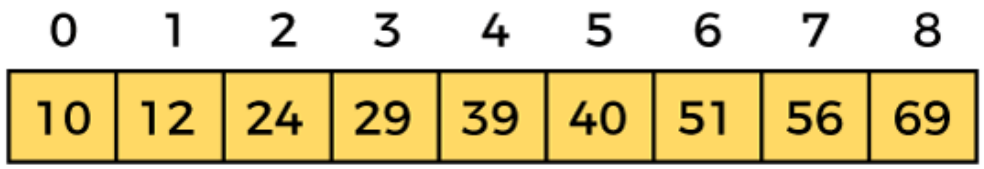
- Binary search is a searching algorithm that works efficiently with a sorted list
- Binary search can be better understood by an analogy of a dictionary
- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match
- Binary search executes in O(log n) time where n is the number of elements in the array
- *NOTE:* Binary search can be implemented on sorted array elements. *If the list elements are not arranged in a sorted manner, we have first to sort them*

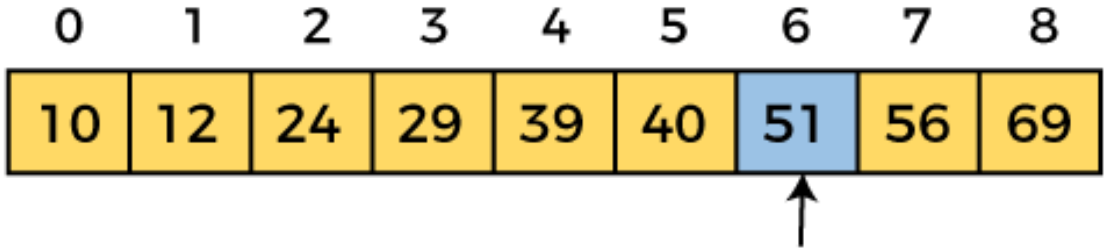
Binary Search Algorithm

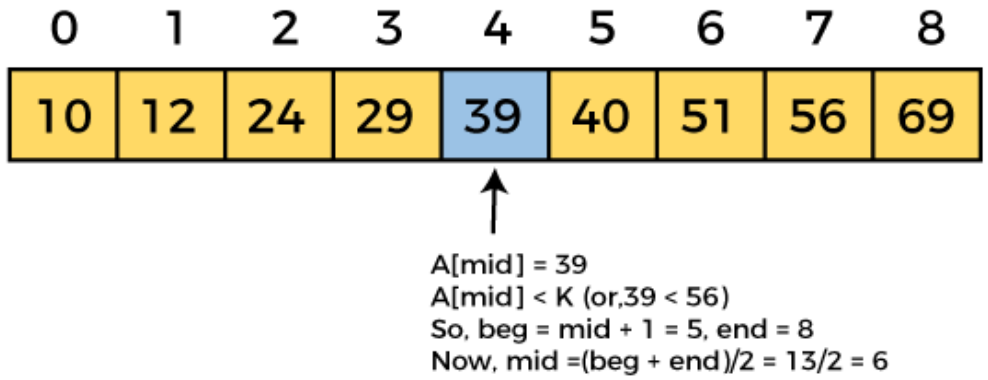
- *Step 1* - Read the search element from the user
- *Step 2* - Find the middle element in the sorted list
- *Step 3* - Compare the search element with the middle element in the sorted list
- *Step 4* - If both are matched, then display "Given element is found!!!" and terminate the function
- *Step 5* - If both are not matched, then check whether the search element is smaller or larger than the middle element
- *Step 6* - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element
- *Step 7* - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element
- *Step 8* - Repeat the same process until we find the search element in the list or until sublist contains only one element
- *Step 9* - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function

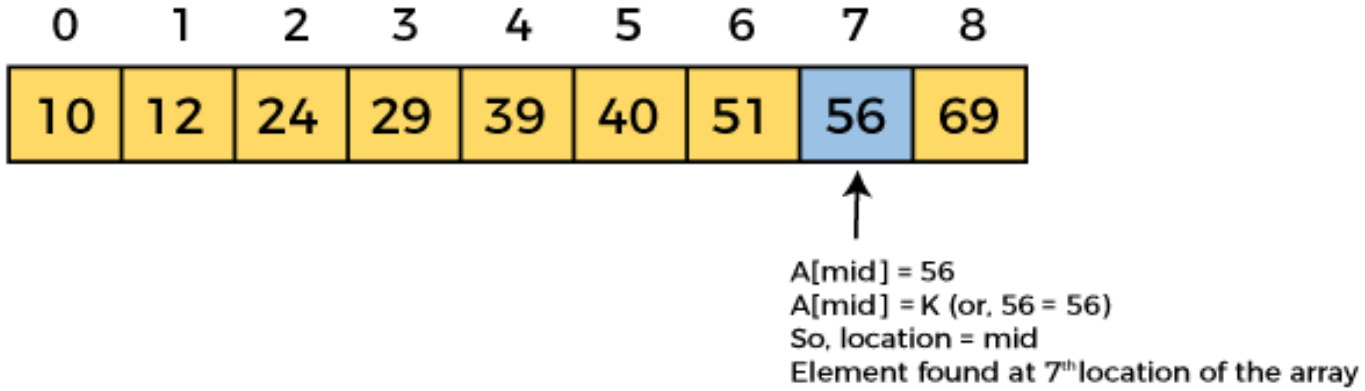
Binary Search Example

- Binary search from array where K = 56









Hashing

Introduction of Hashing

- In all search techniques like linear search, binary search and search trees, the time required to search an element depends on the total number of elements (n) present in that data structure

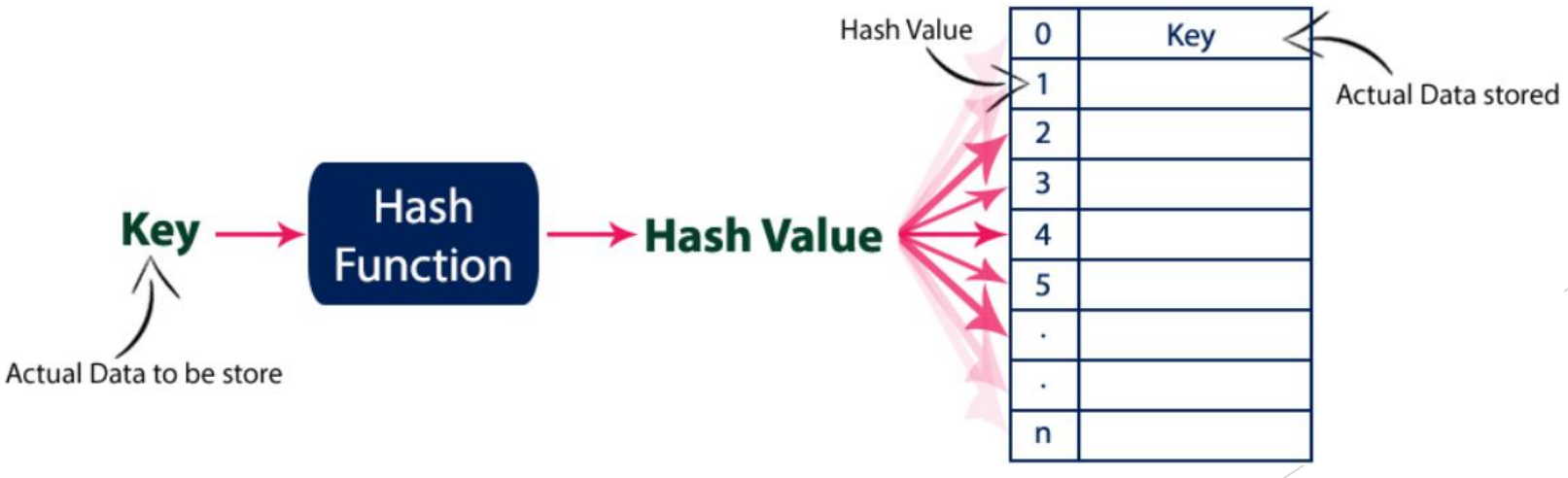
- Hashing is another approach in which time required to search an element doesn't depend on the total number of elements. Using hashing data structure, a given element is searched with constant time complexity

- *Definition:* Hashing is the *process of indexing and retrieving element (data)* in a data structure to provide a faster way of finding the element using a *hash key*

- Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure

Hash tables

- In data structure, we use a concept called Hash table to store data
- All the data values are inserted into the hash table based on the hash key value. The hash key value is used to map the data with an index in the hash table. And the hash key is generated for every data using a hash function
- *Definition:* Hash table is just an array which *maps a key (data) into the data structure with the help of hash function* such that insertion, deletion and search operations are performed with constant time complexity i.e. O(1)



Hash Functions

- Hash function is a function which takes a piece of data *(i.e. key) as input and produces an integer (i.e. hash value) as output* which maps the data to a particular index in the hash table
- The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed
- It produces a unique set of integers within some suitable range in order to reduce the number of collisions
- In practice, there is no hash function that eliminates collisions completely. A good hash function can only *minimize the number of collisions* by spreading the elements uniformly throughout the array

Properties of a Good Hash Function

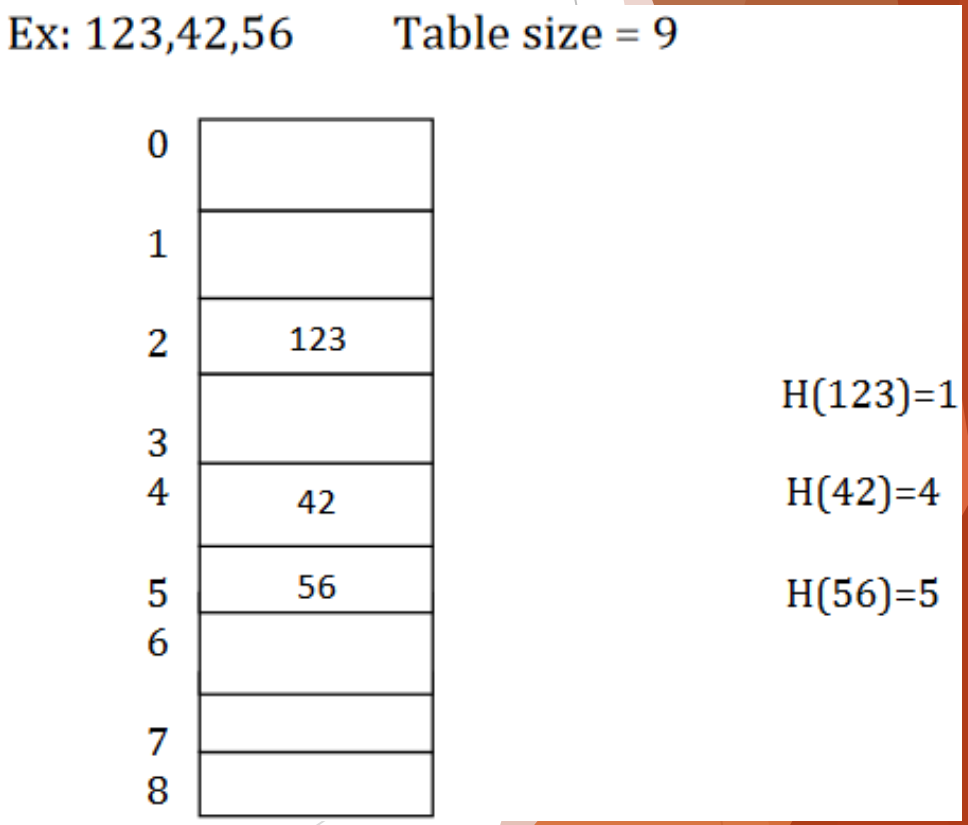
- **Low cost:** The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches
- **Determinism:** A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value
- **Uniformity:** A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same. The property of uniformity also minimizes the number of collisions
- *NOTE:* Collision happens when two keys have same hash value

Types of Hash Functions

- Some of the different types of Hash functions are
  - Truncation Method
  - Mid-Square Method
  - Folding Method
  - Division Method

Truncation Method

- For Truncation Method, ignore part of the key and use the remaining directly as the index
- The Truncation Method truncates a part of the given keys, depending upon the size of the hash table
- First choose the hash table size and then the respective *n right most or left most digits are truncated* and used as hash code/value
- Not a very good method as it does not distribute keys uniformly



Mid-Square Method

- The mid-square method is a good hash function which works in two steps:
- *Step 1:* Square the value of the key. That is, find k^2.
- *Step 2:* Extract the middle r digits of the result obtained in Step 1
- In the mid-square method, the same r digits 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 digits from k^2

Mid-Square Method Sample

- Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations
- When k = 1234, k^2 = 1522756, h (1234) = 27
- When k = 5642, k^2 = 31832164, h (5642) = 21
- Observe that the 3rd and 4th digits starting from the right are chosen
- Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2

Folding Method

- The folding method works in the following two steps:

- *Step 1:* Divide the key value into a number of parts. That is, divide k into parts k_1, k_2, ..., k_n, 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 $k_1 + k_2 + ... + k_n$. The hash value is produced by ignoring the last carry, if any

Folding Method Sample

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

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

Division Method

- For Division method, a standard technique is to use a modulo function on the key, by selecting a divisor M which is a *prime number close to the table size*, so h(K) = K mod M
- The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast
- Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values
- Good values for m are prime numbers that are not very close to powers of 2

Division Method Sample

- Given a hash table of 100 locations, calculate the hash values of keys 1234 and 5462
- Setting M = 97, hash values can be calculated as:
- h(1234) = 1234 % 97 = 70
- h(5642) = 5642 % 97 = 16

Collision Resolution

Collision Resolution

- Collisions occur when the *hash function maps two different keys to the same location*
- Obviously, two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision, also called collision resolution technique, is applied
- The two most popular methods of resolving collisions are:
  - Open Addressing (*Closed hashing*)
  - Separate Chaining (*Open hashing*)

Collision Resolution by Open Addressing

- In an Open Addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. Also, all elements are stored in the hash table itself
- So at any point, the size of the table must be greater than or equal to the total number of keys
- The hash table contains two types of values: sentinel values (e.g., –1) and data values
- The presence of a sentinel value indicates that the location *contains no data value at present but can be used to hold a value*
- The process of examining memory locations in the hash table is called probing
- Open addressing technique can be implemented using linear probing, quadratic probing and double hashing

Linear Probing

- The simplest approach to resolve a collision is linear probing
- In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision
- $h(k, i) = [h'(k) + i] \bmod m$
- Where *m* is the size of the hash table, $h'(k) = (k \bmod m)$, and *i* is the probe number that varies from *0* to *m-1*
- When we have to store a value, we try the slots: $[h'(k)] \bmod m$, $[h'(k) + 1] \bmod m$, $[h'(k) + 2] \bmod m$, $[h'(k) + 3] \bmod m$, $[h'(k) + 4] \bmod m$, $[h'(k) + 5] \bmod m$, and so no, until a vacant location is found
- In linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found
- If the end of the table is reached and no empty cell have been found, then the search is continued from the beginning of the table. It has a tendency to create **cluster** in the table

Linear Probing Sample

- Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table

- Let h'(k) = k mod m, m = 10
- Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

- *Step 1:* Key = 72, h(72, 0) = (72 mod 10 + 0) mod 10 = (2) mod 10 = 2
- Since T[2] *is vacant, insert key* 72 at this location
- *Step 2:* Key = 27, h(27, 0) = (27 mod 10 + 0) mod 10 = (7) mod 10 = 7
- Since T[7] *is vacant, insert key* 27 at this location
- *Step 3:* Key = 36, h(36, 0) = (36 mod 10 + 0) mod 10 = (6) mod 10 = 6
- Since T[6] *is vacant, insert key* 36 at this location

Linear Probing Sample…

- *Step 4:* Key = 24, h(24, 0) = (24 mod 10 + 0) mod 10 = (4) mod 10 = 4
- Since T[4] *is vacant, insert key* 24 at this location
- *Step 5:* Key = 63, h(63, 0) = (63 mod 10 + 0) mod 10 = (3) mod 10 = 3
- Since T[3] *is vacant, insert key* 63 at this location
- *Step 6:* Key = 81, h(81, 0) = (81 mod 10 + 0) mod 10 = (1) mod 10 = 1
- Since T[1] *is vacant, insert key* 81 at this location

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

- *Step 7:* Key = 92, h(92, 0) = (92 mod 10 + 0) mod 10 = (2) mod 10 = 2
- Now T[2] *is occupied, so we cannot store the key* 92 in T[2]
- Therefore, try again for the next location. Thus probe, i = 1, this time
- Key = 92, h(92, 1) = (92 mod 10 + 1) mod 10 = (2 + 1) mod 10 = 3
- Now T[3] *is occupied, so we cannot store the key* 92 in T[3]

Linear Probing Sample…

- Therefore, try again for the next location. Thus probe, i = 2, this time
- Key = 92, h(92, 2) = (92 mod 10 + 2) mod 10 = (2 + 2) mod 10 = 4
- Now T[4] *is occupied, so we cannot store the key* 92 in T[4]
- Therefore, try again for the next location. Thus probe, i = 3, this time.
- Key = 92, h(92, 3) = (92 mod 10 + 3) mod 10 = (2 + 3) mod 10 = 5
- Since T[5] *is vacant, insert key* 92 at this location

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

- *Step 8:* Key = 101, h(101, 0) = (101 mod 10 + 0) mod 10 = (1) mod 10 = 1
- Now T[1] *is occupied, so we cannot store the key* 101 in T[1]
- The procedure will be repeated until the hash function generates the address of location 8 which is vacant and can be used to store the value in it

Challenges in Linear Probing

- One of the problems with linear probing is **Primary clustering**, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster
- More the number of collisions, higher the probes that are required to find a free location and lesser is the performance
- To avoid primary clustering, other techniques such as quadratic probing and double hashing are used

Quadratic Probing

- Suppose a record R with key k has the hash address $H(k) = h$ then instead of searching the location with addresses h, $h+1$, and $h+2$...We linearly search the locations with addresses h, $h+1$, $h+4$, $h+9$...$h+i^2$
- Quadratic Probing uses a hash function of the form
- $h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$ **OR** $h(k, i) = [h'(k) + i^2] \bmod m$
- where m is the size of the hash table, $h'(k) = (k \bmod m)$, i is the probe number that varies from 0 to $m-1$, and c_1 and c_2 are constants such that c_1 and $c_2 \neq 0$
- Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search
- Although quadratic probing performs better than linear probing, in order to maximize the utilization of the hash table, the values of c_1, c_2, and m need to be constrained

Quadratic Probing Sample

- Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81 and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$

- Let h'(k) = k mod m, m = 10
- Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

- *Step 1:* Key = 72, h(72, 0) = (72 mod 10 + 1 x 0 + 3 x 0) mod 10 = (2) mod 10 = 2
- Since T[2] *is vacant, insert key* 72 at this location
- *Step 2:* Key = 27, h(27, 0) = (27 mod 10 + 1 x 0 + 3 x 0) mod 10 = (7) mod 10 = 7
- Since T[7] *is vacant, insert key* 27 at this location
- *Step 3:* Key = 36, h(36, 0) = (36 mod 10 + 1 x 0 + 3 x 0) mod 10 = (6) mod 10 = 6
- Since T[6] *is vacant, insert key* 36 at this location

Quadratic Probing Sample…

- *Step 4:* Key = 24, h(24, 0) = (24 mod 10 + 1 x 0 + 3 x 0) mod 10 = (4) mod 10 = 4
- Since T[4] *is vacant, insert key* 24 at this location
- *Step 5:* Key = 63, h(63, 0) = (63 mod 10 + 1 x 0 + 3 x 0) mod 10 = (3) mod 10 = 3
- Since T[3] *is vacant, insert key* 63 at this location
- *Step 6:* Key = 81, h(81, 0) = (81 mod 10 + 1 x 0 + 3 x 0) mod 10 = (1) mod 10 = 1
- Since T[1] *is vacant, insert key* 81 at this location

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

- *Step 7:* Key = 101, h(101, 0) = (101 mod 10 + 1 x 0 + 3 x 0) mod 10 = (1) mod 10 = 1
- Now T[1] *is occupied, so we cannot store the key* 101 in T[1]

Quadratic Probing Sample…

- Therefore, try again for the next location. Thus probe, i = 1, this time
- Key = 101, h(101, 1) = (101 mod 10 + 1 x 1 + 3 x 1) mod 10
  = (101 mod 10 + 4) mod 10
  = (1 + 4) mod 10
  = (5) mod 10 = 5
- Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	101	36	27	–1	–1

Pros and Cons of Quadratic Probing

- Quadratic probing resolves the primary clustering problem that exists in the linear probing technique
- But linear probing does better memory caching and gives a better cache performance
- One of the major drawbacks of 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
  - It means that if there is a collision between two keys, then the same probe sequence will be followed for both

Double Hashing

- Double hashing is a collision resolving technique in open addressed hash tables
- Double hashing uses the idea of applying a second hash function to key when a collision occurs
- Double hashing uses a hash function of the form
- $h(k, i) = [h_1(k) + ih_2(k)] \bmod m$
- where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as $h_1(k) = k \bmod m$, $h_2(k) = k \bmod m'$, 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 $m-2$ *(preferably prime)*
- Since the offset may vary with every probe depending on the value generated by the second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing

Double Hashing Sample

- Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table. Take h1 = (k mod 10) and h2 = (k mod 8)

- Let m = 10, Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

- We have, $h(k, i) = [h_1(k) + ih_2(k)] \bmod m$
- *Step 1:* Key = 72, h(72, 0) = [72 mod 10 + (0 x 72 mod 8)] mod 10 = [2 + (0 x 0)] mod 10 = 2 mod 10 = 2
- Since T[2] *is vacant, insert key* 72 at this location
- *Step 2:* Key = 27, h(27, 0) = [27 mod 10 + (0 x 27 mod 8)] mod 10 = [7 + (0 x 3)] mod 10 = 7 mod 10 = 7
- Since T[7] *is vacant, insert key* 27 at this location

Double Hashing Sample…

- *Step 3:* Key = 36, h(36, 0) = [36 mod 10 + (0 x 36 mod 8)] mod 10 = [6 + (0 x 4)] mod 10 = 6 mod 10 = 6
- Since T[6] *is vacant, insert key* 36 at this location
- *Step 4:* Key = 24, h(24, 0) = [24 mod 10 + (0 x 24 mod 8)] mod 10 = [4 + (0 x 0)] mod 10 = 4 mod 10 = 4
- Since T[4] *is vacant, insert key* 24 at this location
- *Step 5:* Key = 63, h(63, 0) = [63 mod 10 + (0 x 63 mod 8)] mod 10 = [3 + (0 x 7)] mod 10 = 3 mod 10 = 3
- Since T[3] *is vacant, insert key* 63 at this location
- *Step 6:* Key = 81, h(81, 0) = [81 mod 10 + (0 x 81 mod 8)] mod 10 = [1 + (0 x 1)] mod 10 = 1 mod 10 = 1
- Since T[1] *is vacant, insert key* 81 at this location

Double Hashing Sample…

- The hash table now becomes

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

- *Step 7:* Key = 92, h(92, 0) = [92 mod 10 + (0 x 92 mod 8)] mod 10 = [2 + (0 x 4)] mod 10 = 2 mod 10 = 2
- Now T[2] *is occupied, so we cannot store the key* 92 in T[2]
- Therefore, try again for the next location. Thus probe, i = 1, this time
- Key = 92, h(92, 1) = [92 mod 10 + (1 x 92 mod 8)] mod 10 = [2 + (1 x 4)] mod 10 = 6 mod 10 = 6
- Now T[6] *is occupied, so we cannot store the key* 92 in T[6]
- Therefore, try again for the next location. Thus probe, i = 2, this time
- Key = 92, h(92, 2) = [92 mod 10 + (2 x 92 mod 8)] mod 10 = [2 + (2 x 4)] mod 10 = 10 mod 10 = 0

Double Hashing Sample…

- Since T[0] *is vacant, insert key* 92 in T[0]. The hash table now becomes

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	–1	36	27	–1	–1

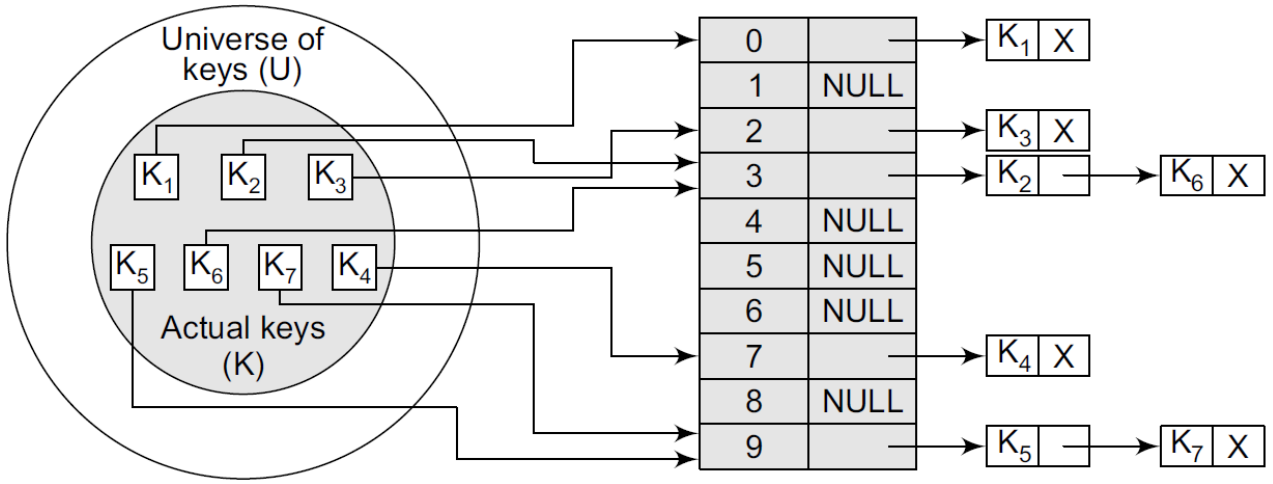
- *Step 8:* Key = 101, h(101, 0) = [101 mod 10 + (0 x 101 mod 8)] mod 10 = [1 + (0 x 5)] mod 10 = 1 mod 10 = 1
- Now T[1] *is occupied, so we cannot store the key* 101 in T[1]
- Therefore, try again for the next location. Thus probe, i = 1, this time
- Key = 101, h(101, 1) = [101 mod 10 + (1 x 101 mod 8)] mod 10 = [1 + (1 x 5)] mod 10 = 6 mod 10 = 6
- Now T[6] *is occupied, so we cannot store the key* 101 in T[6]
- Now repeat the entire process *until a vacant location* is found. In our case m=10, which is *not a prime number*, hence, the degradation in performance. Had m been equal to 11, the algorithm would have worked very efficiently

Advantages of Double Hashing

- Double hashing minimizes repeated collisions and the effects of clustering
- Double hashing is free from problems associated with primary clustering as well as secondary clustering
- Double Hashing technique does not yield any clusters and is one of effective method for resolving collisions

Collision Resolution by (Separate)Chaining

- In chaining, each location in a hash table stores a pointer to a linked list that contains *all the key values that were hashed to that location*
- That is, location l in the hash table points to the head of the linked list of all the key values that hashed to l. However, if no key value hashes to l, then location l in the hash table contains NULL
- It is otherwise called as direct chaining or separate chaining



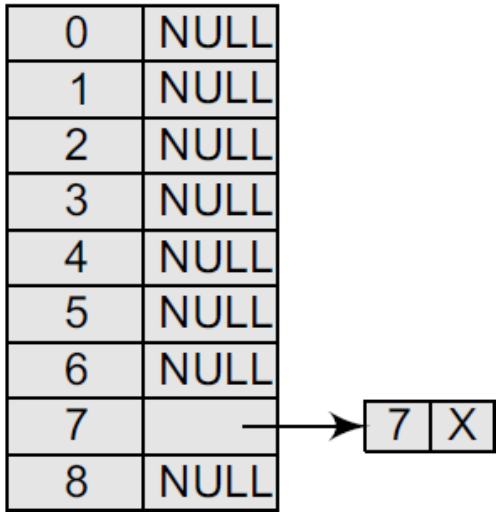
Operations on a Chained Hash Table

- Searching for a value in a chained hash table is as simple as scanning a linked list for an entry with the given key
- Insertion operation appends the key to the end of the linked list pointed by the hashed location
- Deleting a key requires searching the list and removing the element
- Chained hash tables with linked lists are widely used due to the simplicity of the algorithms to insert, delete, and search a key
- Cost of inserting a key in a chained hash table is O(1)
- The cost of deleting and searching a value is given as O(m) where m is the number of elements in the list of that location

Chained Hashing Sample

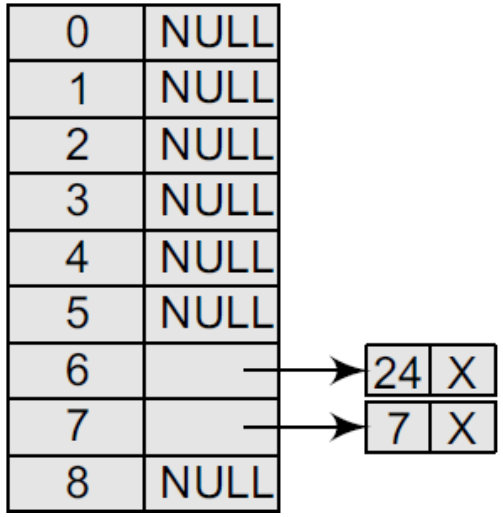
- Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use h(k) = k mod m.
- In this case, m=9. Initially, the hash table can be given as:
- **Step 1** Key = 7
- h(k) = 7 mod 9 = 7

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL



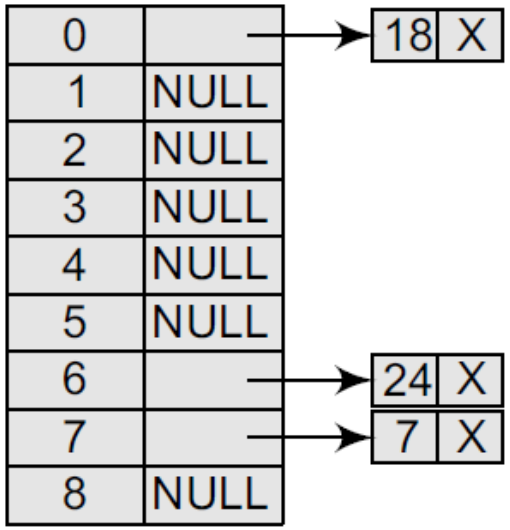
Step 2 Key = 24

h(k) = 24 mod 9 = 6



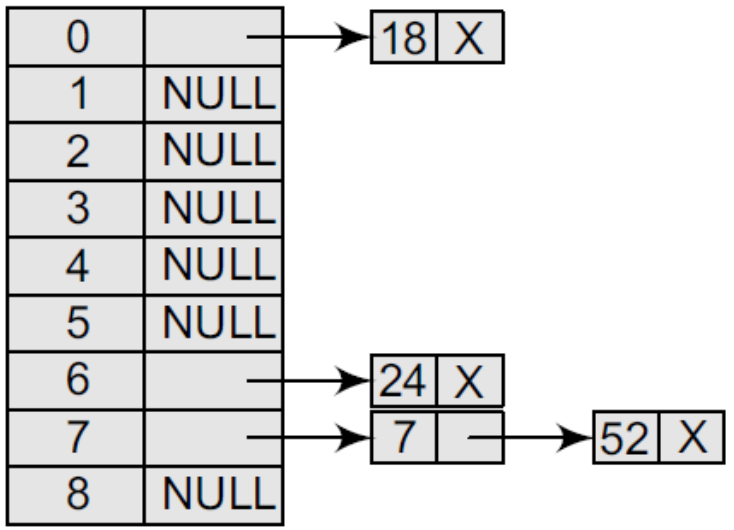
Chained Hashing Sample…

- Step 3 Key = 18
- h(k) = 18 mod 9 = 0



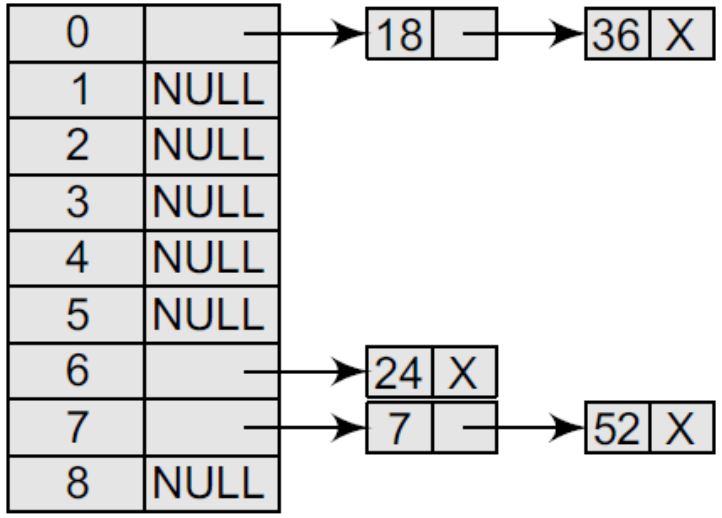
Step 4 Key = 52

h(k) = 52 mod 9 = 7



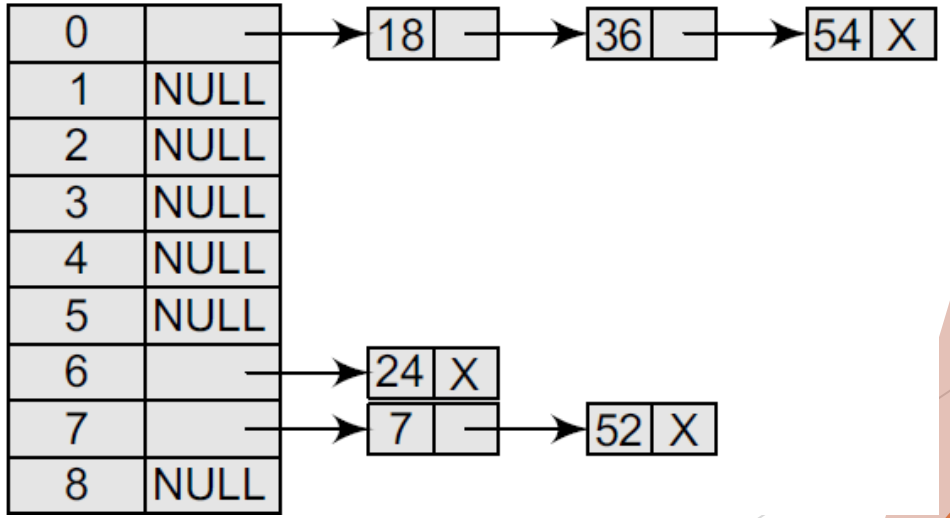
Chained Hashing Sample…

- Step 5: Key = 36
- h(k) = 36 mod 9 = 0



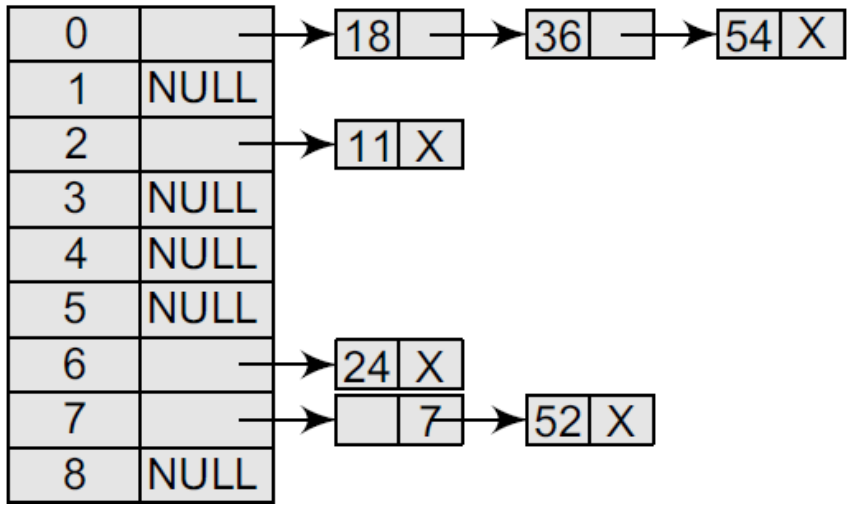
Step 6: Key = 54

h(k) = 54 mod 9 = 0



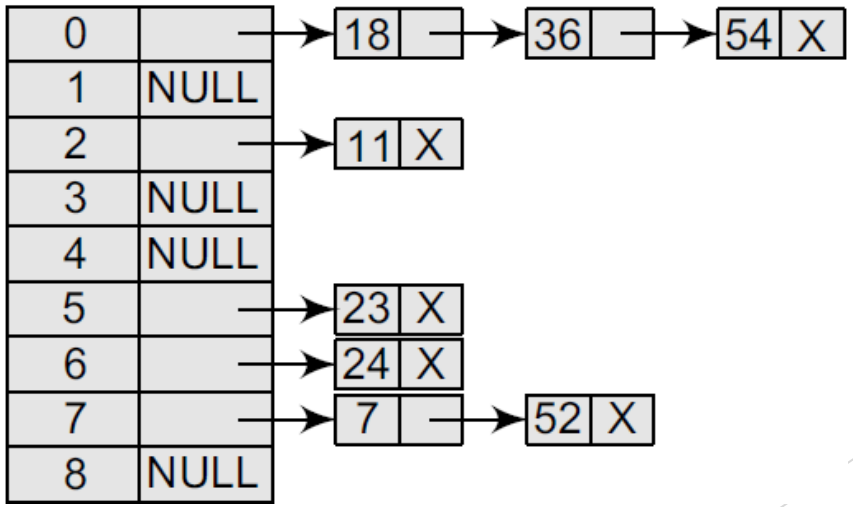
Chained Hashing Sample…

- Step 7: Key = 11
- h(k) = 11 mod 9 = 2



Step 8: Key = 23

h(k) = 23 mod 9 = 5



Pros and Cons of Chaining

- The main advantage of using a chained hash table is that it remains effective even when the number of key values to be stored is much higher than the number of locations in the hash table
- However, with the increase in the number of keys to be stored, the performance of a chained hash table does degrade gradually (linearly)
- The other advantage of using chaining for collision resolution is that its performance, unlike quadratic probing, does not degrade when the table is more than half full
- This technique is absolutely free from clustering problems and thus provides an efficient mechanism to handle collisions
- However, chained hash tables inherit the disadvantages of linked lists. Firstly, the space overhead of the next pointer while storing a key value and secondly traversing a linked list has poor cache performance, making the processor cache ineffective

Analysis of all Searching techniques

- **Linear Search**
- The time complexity is O(n) where n is the number of elements in the array

- **Binary Search**
- The time complexity is O(log n) where n is the number of elements in the array

- **Hashing Techniques**
- The time complexity of hashing is constant i.e., O(1) but will starting getting worse when the collision increases

Analysis of all Searching techniques…

- **Linear Probing**
- The average case and best case time complexity for search, insert and delete is O(1)
- The worst case time complexity for search, insert and delete is O(n)

- **Quadratic Probing**
- The average case and best case time complexity for search, insert and delete is O(1)
- The worst case time complexity for search, insert and delete is O(n)

- **Double Probing**
- The average case and best case time complexity for search, insert and delete is O(1)
- The worst case time complexity for search, insert and delete is O(n)

Analysis of all Searching techniques…

- **Separate Chaining**
- The average case and best case time complexity for search, insert and delete is O(1)
- The worst case time complexity for search, insert and delete is O(n)

Sorting Techniques

- Sorting techniques were covered during practical's (Insertion Sort, Selection Sort, Merge Sort)
- For the exam for sorting techniques (Insertion Sort, Selection Sort, Merge Sort) you need to learn the algorithms and their complexities
- The analysis of sorting techniques focuses on the complexity of the 3 sorting techniques
- *NOTE: Quick Sort has been excluded due to shortage of time and its complexity. But the external might ask about it during viva so learn its theory to be on the safe side.*

Any Questions