Module 6

CHAPTER 6 Searching Techniques

University Prescribed Syllabus

Linear Search, Binary Search, Hashing - Concept, Hash Functions, Collision Resolution Techniques.

6.1 SEARCHING

UQ. 6.1.1 *What are various searching techniques?*

MU - Dec. 17, 3 Marks

Definition : Searching *is an operation which finds the element with its location in given list.*

- In search operation if element is found in given list then search is successful otherwise not.

There are two types of searching

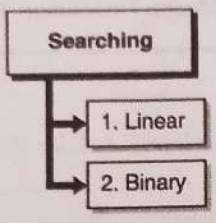


Fig. 6.1.1 : Types of Searching

Syllabus Topic : Linear Search

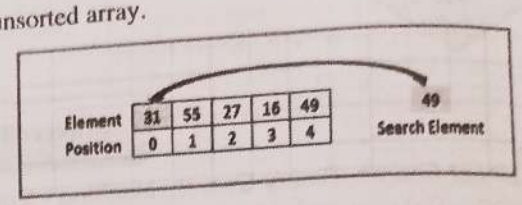
6.1.1 Linear Search

GQ. 6.1.2 *Explain linear search with suitable example.* **(3 Marks)**

- **Concept :** This is simple method of searching.
- In this method, the element is searched in sequential manner; hence this search is called as sequential search.
- This algorithm can be applied on both sorted and unsorted list.
- **Example :** Now consider an array arr[] having 5 elements which are not sorted :

```
int arr[] = {31,55,27,16,49}
```

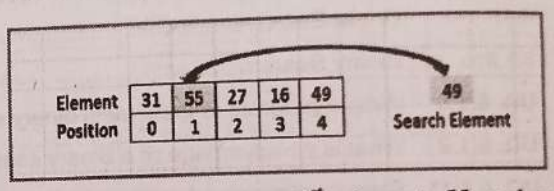
- We have to search 49.
- In unsorted array, the 0^{th} location number is compared with element you want to search if both are equal, then the number is found otherwise element is compared with 1^{st} location element and so on.
- In this method, the element to search is compared with all the array elements from first position. When it is matched with any of element, then search is stopped and prints the element with its position.
- If list is unsorted, continue above step until end of list is reached or number is found.
- If list is sorted, continue above step until number is found or an element is found which is greater than element being searched.
- We will see step by step process of linear search in unsorted array.



Comparison between : 0^{th} element **31** and search element **49.**

Conclusion : 31 != 49

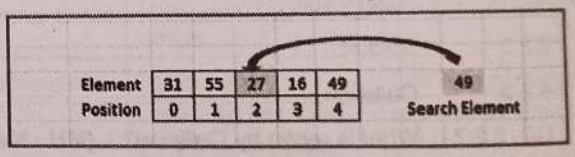
Result : Compare next element



Comparison between : 1^{st} element 55 and search element **49.**

Conclusion : 55 != 49

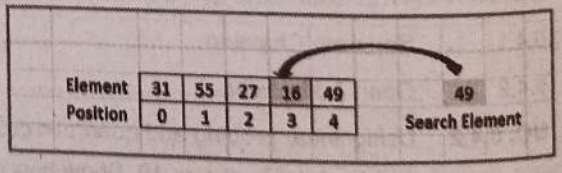
Result : Compare next element



Comparison between : 2^{nd} element 27 and search element **49.**

Conclusion : 27 != 49

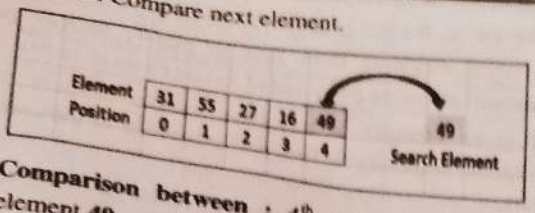
Result : Compare next element.



Comparison between : 3^{rd} element **16** and search element **49.**

Conclusion : 16 != 49

Result : Compare next element.



Comparison between : 4th element 49 and search element **49.**

Conclusion : 49 = 49

Result : Print the element and position nd stop the search.

Algorithm of linear search

GQ. 6.1.3 *Write an algorithm for Linear Search.* **(3 Marks)**

1. Accept element from user which you want to search.
2. Compare the search element with the 0th location element in the list.
3. If both are equal then search is successful. Stop the search
4. If both are not equal then next element in the list is compared with search element.
5. Repeat step 3 and 4 until search element is compared with last element.
6. If none of element matches with element to search, then display message number is not found.

Analysis

- Total numbers of comparisons' are **N.**
- Then the complexity of linear search is O(N).

Program on linear search

GQ. 6.1.4 *WAP to implement linear search.* **(4 Marks)**

```
#include <stdio.h>
int main()
{
int a[100], search, i, n;

printf("\n Enter the size of array : ");
scanf("%d",&n);
printf("\n Enter %d numbers : ", n);
for (i = 0; i < n; i++)
  scanf("%d", &a[i]);

printf("\n Enter the number you want to search : ");
scanf("%d", &search);

for (i = 0; i < n; i++)
{
 if (a[i] == search)
 {
   printf("\n %d is present at location: %d", search, i+1);
   break;
 }
}
if (i == n)
  printf("\n %d is not present in array", search);
return 0;
}
```

Accepts array elements.

Compare given element with array elements.

Up to end if element not found, prints element not found

Output

```
C:\linear_search.exe
Enter the size of array : 5
Enter 5 numbers : 31 55 27 16 49
Enter the number you want to search : 49
49 is present at location: 5.
```

Explanation

- Here program first accepts the size of array and array elements from user.
- Then accept a number which to be searched.
- If given number is matched with array element then print number is found and its location.
- If none of element matches with element to search, then display message number is not found.

Advantages of linear search

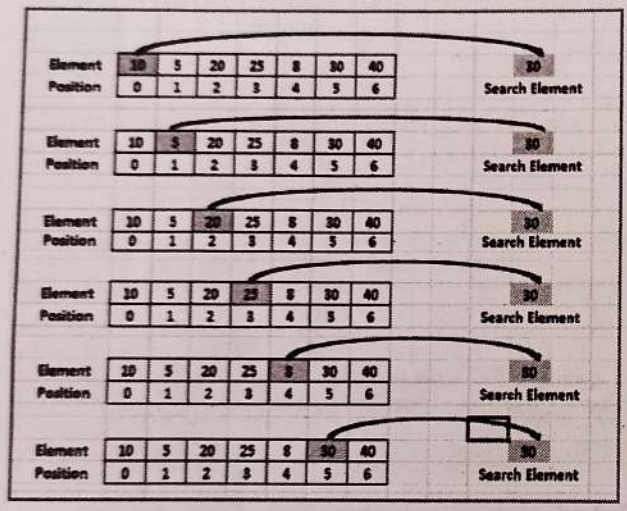
1. The linear search is simple (easy for understand).
2. This algorithm is applied on both sorted and unsorted list.

Disadvantages of linear search

1. This search is not efficient when list is large.
2. Maximum number of comparisons are N. (when list contain n elements).

GQ. 6.1.5 *Find position of element 30 using linear search algorithm in given sequence.*

10, 5, 20, 25, 8, 30, 40 **(2 Marks)**



30 found at location (5+1) = 6.

Syllabus Topic : Binary Search

6.1.2 Binary Search

GQ. 6.1.6 *Explain binary search with suitable example.* **(4 Marks)**

- **Concept :** Binary search is very fast searching technique.
- This algorithm can be applied only on sorted list.
- Mid = $\frac{\text{Lower + Upper}}{2}$ this formula is used to find the mid of array.
- In this algorithm, given element is compared with middle element of the list, if these two are equal then prints element is found (search is successful). Otherwise, the list is divided into two parts. First part contains elements from first to mid-1, another part contains elements from mid+1 to last element in list.
- If given element is less than middle element then continue searching in first part otherwise in the second part.
- Repeat above steps till element is found or the division of part gives only one element.

Example

Now consider an array arr[7] having 5 elements

- int arr[] = {5,17,21,25,47,73,92}
- Search element 17.

	L			M			U
Element	5	17	21	25	47	73	92
Position	0	1	2	3	4	5	6

Here Lower =0 and Upper = 6

$$\text{Mid} = \frac{\text{Lower + Upper}}{2}$$

$$= \frac{0+6}{2} = 3$$

Checks 25 == 17 no, here 17 < 25 hence search is continued in first part of list.

Then

$$\text{Upper} = \text{mid-1} = 3 - 1$$

$$= 2$$

Again calculate mid

$$\text{Mid} = \frac{\text{Lower + Upper}}{2} = \frac{0+2}{2}$$

$$\text{Mid} = 1$$

	L	M	U				
Element	5	17	21	25	47	73	92
Position	0	1	2	3	4	5	6

Here 17==17 the search is successful.

Algorithm of Binary Search

GQ. 6.1.7 *Describe binary search algorithm.* **(3 Marks)**

1. Accept element from user which to be searched.
2. Search element is compared with mid element of the list, if these two are equal then the element is found (search is successful).

Otherwise, the list is divided into two parts. First part contain first element to mid-1 element, another part contain mid+1 to last element in list.

If search element is less than mid then continue searching in first part otherwise in the second part.

Repeat step 2, 3 and 4 until element is found or the division of part gives only one element.

Analysis

If list contain N elements then N/2 comparisons are required.

Then the complexity of Binary search is **O(N/2).**

Program on binary search

UQ. 6.1.8 Write a program to implement binary search. **MU - Dec. 17, 8 Marks**

```
#include <stdio.h>
int main() ,
{
int k, lower, upper, mid, n, search, arr[50];
printf("\n Enter size of array : ");
scanf("%d",&n);
printf("\n Enter %d elements : ", n);
for (k = 0; k < n; k++)
  {
   scanf("%d",&arr[k]);
  }
printf("\n Enter element you want to search : ");
scanf("%d", &search);
lower = 0;
upper = n - 1;
mid = (lower+upper)/2;

while (lower <= upper) {

if (arr[mid] < search)
    lower = mid + 1;
else if (arr[mid] == search)
  {
printf("\n %d found at location : %d", search, mid+1);
   break;
  }
  else
    upper = mid - 1;

   mid = (lower + upper)/2;
}
if (lower > upper)
  printf("\n %d Not found in the list ", search);
return 0;
}
```

Accepts array elements

Setting lower, upper and mid positions

If given element is greater than middle element then continue searching in second part.

If given element is equal to mid element then print number is found.

Output

```
C:\binary_search.exe
Enter size of array : 7
Enter 7 elements : 5 17 21 25 47 73 92
Enter element you want to search : 17
17 found at location : 2
```

Difference between linear and binary search

UQ. 6.1.9 What is an advantage of a Binary search over linear search ? **MU - May 15, 1 Mark**

GQ. 6.1.10 Distinguish between linear search and binary search. **(3 Marks)**

Parameters	Linear Search	Binary Search
Suitable for	Applied on both sorted and unsorted list.	Applied on only sorted list.
Understanding	Very simple to understand.	Difficult to understand.
Use	This search is not useful when list is long.	This search is useful when list is long.
Time complexity	O(n).	O(log n).

UQ. 6.1.11 *State best case and worst case time complexity of linear search.*

MU - May 15, 2 Marks

Algorithm	Time Complexity		
	Best	Average	Worst
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

Syllabus Topic : Hashing

6.2 HASHING

UQ. 6.2.1 *What is hashing?*

MU - May 14, Dec. 16, Dec. 17. Dec. 19, 2 Marks

UQ. 6.2.2 *What is hashing ?* **MU - Dec. 18, 1 Mark**

- In all search techniques like linear search, binary search and search trees, the time required to search an element is depend on the total number of elements in that data structure.
- In all these search techniques, as the number of elements gets increased, the time required to search an element also increased linearly.
- Hashing is another approach in which time required to search an element doesn't depend on the number of elements.
- Using hashing data structure, an element is searched with constant time complexity.
- Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

GQ. 6.2.3 *Define hashing.* **(2 Marks)**

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

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

6.2.1 Hash Table

- In this 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. Hash key value is used to map the data with index in the hash table.
- And the hash key is generated for every data using a hash function.
- That means every entry in the hash table is based on the key value generated using a hash function.

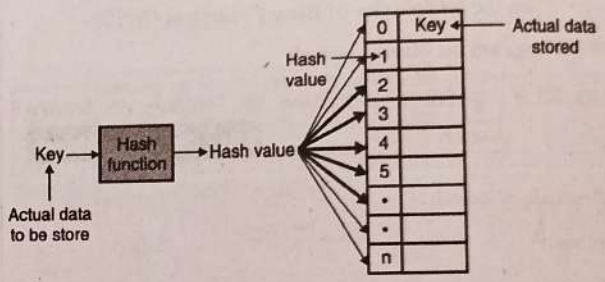


Fig. 6.2.1 : Hash table

GQ. 6.2.4 *Define Hash Table.* **(2 Marks)**

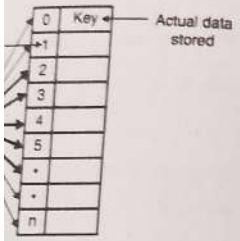
Definition : Hash table is an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

- Hash tables are used to perform the operations like insertion, deletion and search very quickly in a data structure.
- Using hash table concept, insertion, deletion and search operations are accomplished in constant time.
- Generally, every hash table makes use of a function, which we'll call to map the data into the hash table.

e use a concept called **Hash**

erted into the hash table based
ash key value is used to map
ash table.
rated for every data using a

he hash table is based on the
hash function.



table

(2 Marks)

array which maps
structure with the
h that insertion,
erations can be
ne complexity (i.e.

the operations like
y quickly in a data

deletion and search
tant time.

use of a function,
the hash table.

Applications of Hash Table

GQ. 6.2.5 Write Applications of Hash Table. (3 Marks)

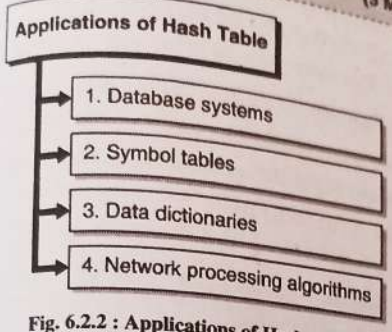


Fig. 6.2.2 : Applications of Hash Table

1. Database systems

- Specifically, those that require efficient random access.
- Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

2. Symbol tables

- The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently.
- Therefore, it is important that symbol tables be implemented very efficiently.

3. Data dictionaries

- Data structures that support adding, deleting, and searching for data.
- Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

4. Network processing algorithms

Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

6.2.2 Bucket

GQ. 6.2.6 What is Bucket? (2 Marks)

- Assume you have a housing colony in which there are lots of buildings or apartments and if asked the address of your house, you can answer that I live in XYZ building.
- So, a building or apartment is a bucket, housing colony is a hash table and your house is an entry.
- Technically, hash table stores entries (key, value) pairs in buckets.
- Hash table uses key's hash-code to figure out the bucket. Ideally, the hash function is written in such a manner that each key is mapped to a unique bucket.

6.2.3 Collision

UQ. 6.2.7 What is mean by Collision?
MU - May 14, Dec. 16, 2 Marks

- A situation when the resultant hashes for two or more data elements in the data set *U*, maps to the same location in the hash table, is called a hash collision.
- In such situation two or more data elements would qualify to be stored / mapped to the same location in the hash table.

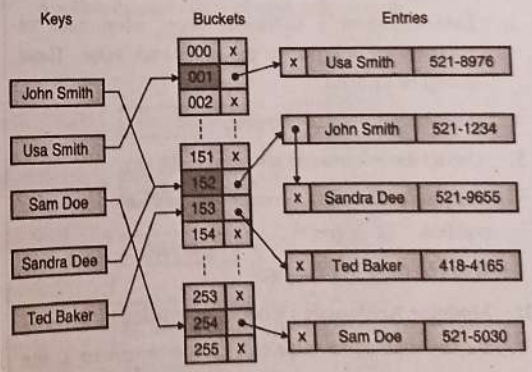


Fig. 6.2.3 : Collision

Module 6

Syllabus Topic : Hash Function

6.3 HASH FUNCTION

GQ. 6.3.1 *Describe hash function.* **(5 Marks)**

- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.
- The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.
- To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:

6.3.1 Characteristics of Good Hash Function

UQ. 6.3.2 *What properties should a good hash function demonstrate ?*

MU - Dec. 18, 4 Marks

1. **Easy to compute :** It should be easy to compute and must not become an algorithm in itself.
2. **Uniform distribution :** It should provide a uniform distribution across the hash table and should not result in clustering.
3. **Less collisions :** Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.
4. Be easy and quick to compute.
5. Use all the information provided in the key.

- The hash function converts the key into the table position.

It can be carried out using :

1. **Modular Arithmetic :** Compute the index by dividing the key with some value and use the remainder as the index.
   - This gives the indexes in the range 0 to $m_7 1$ so the hash table should be of size m
   - This is an example of uniform hash function if value of m will be chosen carefully.
   - Generally a prime number is a best choice which will spread keys evenly.
   - A uniform hash function is designed to distribute the keys roughly evenly into the available positions within the array (or hash table).
   - This forms the basis of the next two techniques.

☞ **For Example**

index := key MOD table_size

Consider key is 102 and table size is 10

index := 102 MOD 10

index := 2

2. **Truncation :** Ignoring part of the key and using the rest as the array index.

   The problem with this approach is that there may not always be an even distribution throughout the table.

☞ **For Example**

If student id's are the key such as 928324312 then select just the last three digits as the index i.e. 312 as the index.

3. **Folding :** Partition the key into several pieces and then combine it in some convenient way.

☞ **For Example**

- The key K is partitioned into number of parts ,each of which has the same length as the required address with the possible exception of the last part .
- The parts are then added together , ignoring the final carry, to form an address.

☞ **Example**

- If key = 356942781 is to be transformed into a three digit address. P1 = 356, P2 = 942, P3 = 781 are added to yield 079.

4. **Mid square method**

GQ. 6.3.3 *Explain mid square method using suitable example.* **(4 Marks)**

- The key value is squared in this method and the middle or mid part of the result is used as the index into the table.

- If this middle part is greater for the size of the table, then it is divided to map the size of the table.
- In this method, the whole key participates in generating the index for the table, thus, there is a greater chance of finding a different address every time per key value.

☞ **Example**

- For Example, if the key value is 2199, the squared result is 4835601.
- The middle part of this result is 356, which will now be used for indexing into the table.
- Say, if the size of the table is 1000, then h(2199) = 356.
- That is, the hash function using mid – square method will produce 356 as the index value for key value 2199.
- The advantage is that the result is not dominated by the distribution of bottom digit or top digit of the original key value.
- The problem with this method is choosing the middle part of the result obtained by squaring the key value.
- To resolve this issue, it is advisable to choose the size of the table to be of a power of 2, such as 2r so that the middle r bits are chosen.
- In such a case, it is easier to choose the middle part of the result using masking and shift operations.

5. Digit analysis

- Suppose the key is a social security number such as 987-65-4321. The hash function could choose the last three digits 321 or use the 3rd, 5th, and 8th digits : 752.
- The digits selected must be 'random' - no patterns that would increase the number of collisions.
- It is often better to analyze the key population in advance. Choose digits yielding the most uniform distribution.
- Realize that correlations among digits are possible. For example, if we look at the first three digits of the social security numbers of people who live in Stanislaus county, we will find many instances of a certain few patterns, and low concentrations of most patterns - thus hashing by the first three digits of the social security number is not a good idea.
- On the other hand the last three or four digits of the social security number tends to be a very good hash function.

Syllabus Topic : Collision Resolution Techniques

6.4 COLLISION RESOLUTION TECHNIQUES

UQ. 6.4.1 *Write short note on : Collision Handling techniques.* **MU - May 19, 5 Marks**

Definition : When two different keys produce the same address, there is a collision. The keys involved are called synonyms.

- Coming up with a hashing function that avoids collision is extremely difficult. It is best to simply find ways to deal with them. The possible solution, can be:
  - Spread out the records
  - Use extra memory
  - Put more than one record at a single address.

☞ **An example of Collision**

- Hash table size : 11 Hash function : key mod hash size So, the new positions in the hash table are :

Key	23	18	29	28	39	13	16	42	17
Position	1	7	7	6	6	2	5	9	6

Fig. 6.4.1 : Collision

☞ **Another example (in a phonebook record)**

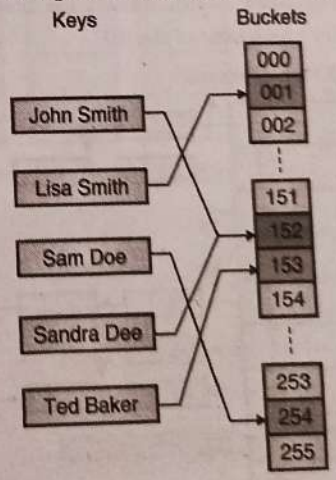


Fig. 6.4.2 : Collision

- Here, the buckets for keys 'John Smith' and 'Sandra Dee' are the same. So, its a collision case.

Module 6

- Collision occurs when h(k1) = h(k2), i.e. the hash function gives the same result for more than one key.

☞ **Strategies used for collision resolution**

1. **Chaining**
   - Store colliding keys in a linked list at the same hash table index
2. **Open Addressing**
   - Store colliding keys elsewhere in the table

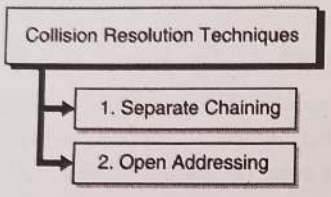


Fig. 6.4.3 : Collision Resolution Techniques

6.4.1 Separate Chaining

- Maintains a linked list at every hash index for collided elements.
- Lets take the example of an insertion sequence: {0 1 4 9 16 25 36 49 64 81}

  Here, h(k) = k mod table_size
  = k mod 10 (tablesize = 10)
- Hash table T is a vector of linked lists. Key k is stored in list at T[h(k)]
- So, the problem is like: "Insert the first 10 perfect squares in a hash table of size 10"
- The hash table looks like:

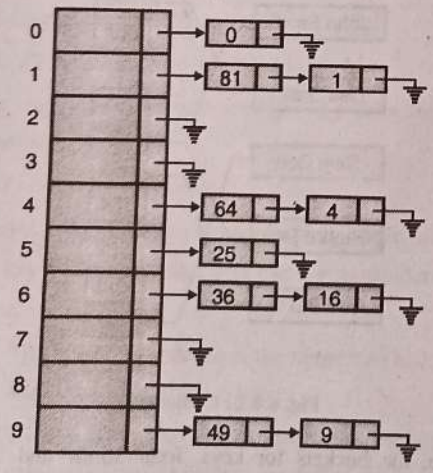


Fig. 6.4.4 : Chaining

☞ **Disadvantages of Chaining**

1. Linked lists could get long. Longer linked lists could negatively impact performance.
2. More memory utilization because of pointers.

6.4.2 Open Addressing

- Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets).
- If the index given by the hash function is occupied, then there is need to find another bucket for the element to be stored. Then increment the table position by some number.

There are three schemes commonly used for probing :

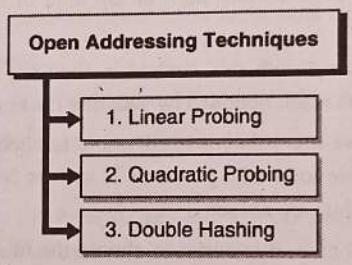


Fig. 6.4.5 : Open Addressing Techniques

1. Linear Probing

- The next slot for the collided key is found in this method by using a technique called "Probing".
- It generates a probe sequence of slots in the hash table and we need to choose the proper slot for the key 'x'.
- Suppose that a key hashes into a position that has been already occupied. The simplest strategy is to look for the next available position to place the item.
- Suppose we have a set of hash codes consisting of {89, 18, 49, 58, 9} and we need to place them into a table of size 10.

- The following table demonstrates this process :

hash (89,10) = 9
hash (18,10) = 8
hash (49,10) = 9
hash (58,10) = 8
hash (9,10) = 9

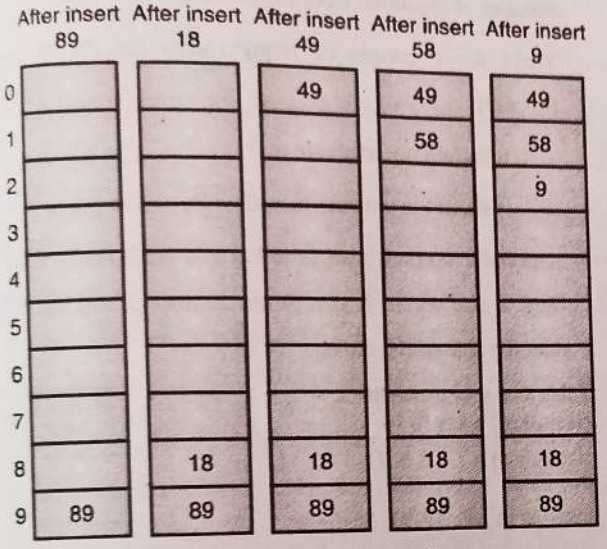


Fig. 6.4.6 : Linear probing

- The first collision occurs when 49 hashes to the same location with index 9.
- Since 89 occupies the A[9], we need to place 49 to the next available position.
- Considering the array as circular, the next available position is 0. That is (9+1) mod 10. So we place 49 in A[0].
- Several more collisions occur in this simple example and in each case we keep looking to find the next available location in the array to place the element.
- Now if we need to find the element, say for example, 49, we first compute the hash code (9), and look in A[9].
- Since we do not find it there, we look in A[(9+1) % 10] = A[0], we find it there and we are done.
- So what if we are looking for 79? First we compute hashcode of 79 = 9.
- We probe in A[9], A[(9+1)]=A[0], A[(9+2)]=A[1], A[(9+3)]=A[2], A[(9+4)]=A[3] etc.
- Since A[3] = null, we do know that 79 could not exists in the set.

2. Quadratic Probing

- Although linear probing is a simple process where it is easy to compute the next available location, linear probing also leads to some clustering when keys are computed to closer values.
- Therefore we define a new process of Quadratic probing that provides a better distribution of keys when collisions occur.
- In quadratic probing, if the hash value is K , then the next location is computed using the sequence K + 1, K + 4, K + 9 etc..
- The following table shows the collision resolution using quadratic probing.

hash (89,10) = 9
hash (18,10) = 8
hash (49,10) = 9
hash (58,10) = 8
hash (9,10) = 9

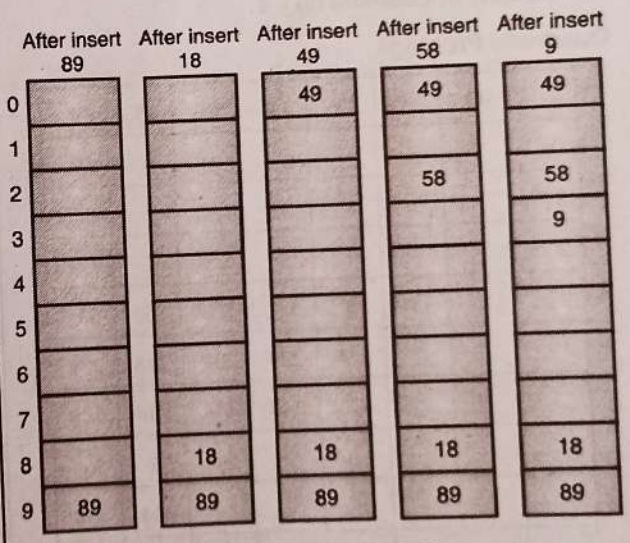


Fig. 6.4.7 : Quadratic probing

UQ. 6.4.2 *Using linear probing and quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each technique :*

99, 33, 23, 44, 56, 43, 19

MU - Dec. 13, May 14, Dec. 16, 10 Marks

Linear Probing

	99	33	23	44	56	43	19
0							19
1							
2							
3		33	33	33	33	33	33
4			23#	23#	23#	23#	23#
5				44#	44#	44#	44#
6					56	56	56
7						43#	43#
8							
9	99	99	99	99	99	99	99

Number of Collisions (#) = 4

Quadratic Probing

	99	33	23	44	56	43	19
0							19
1							
2							
3		33	33	33	33	33	33
4			23#	23#	23#	23#	23#
5				44#	44#	44#	44#
6					56	56	56
7						43#	43#
8							
9	99	99	99	99	99	99	99

Number of Collisions (#) = 4

3. Double Hashing

- Double hashing uses the idea of applying a second hash function to the key when a collision occurs.
- The result of the second hash function will be the number of positions from the point of collision to insert.
- There are a couple of requirements for the second function:
  - It should not evaluate to 0
  - It should make sure that all cells can be probed
- A popular second hash function is : $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.

Table size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 positions from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Fig. 6.4.8 : Double probing

..Chapter Ends

❑❑❑