Unit-7 [Searching and Hashing]

#Searching

Searching is an operation or a technique that helps to find the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:-

@. Linear Search or Sequential Search:

This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list. If the last element also doesn't match, then the result is "Element not found in the list." That means, the search element is compared with element by element in the list.

Algorithm:

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

Pseudo code

```
LinearSearch (A, n, key)
{ flag=0;
  for (i=0; i<n; i++)
  { if (A[i]==key)
        flag=1;
  }
  if (flag==1)
      Print "Search successful"
  else
      Print "Search un-successful"
}
```

Analysis:

Time Complexity = O(n).

(b) Binary Search:

The binary search algorithm can be used only with sorted list of element. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found." Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat same process for left sub list of the middle element. But if the search element is larger, then we repeat the same process for right sub list of middle element. We repeat the process until we find the search element in the list ~~or until we left with a sub list of only and if that element~~ and if the element doesn't match with any of the element in list, then result is "Element not found".

Algorithm:

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

Pseudo Code

```
Binary Search (a, l, r, key)
  {
    int m;
    int flag = 0;
    if (l<=r)
      { m = (l+r)/2;
        if (key== a[m])
        flag = m;
        else if (key< a[m])
          return Binary Search (a, l, m-1, key);
        else
          return Binary Search (a, m+1, r, key);
      }
    else
      return flag;
  }
```

Efficiency:

From the above algorithm we can say that running time of algorithm is:

$$T(n) = T(n/2) + O(1)$$

By solving this we get $O(\log n)$.

In best case output is obtained at one run i.e, $O(1)$, time if key is at middle.

In worst case the output is at the end of the array so running time is $O(\log n)$ time.

In average case also running time is $O(\log n)$.

Hence Time complexity = $O(\log n)$

Hashing

Hashing is an efficient searching technique in which key is placed in direct accessible address for rapid search. Hashing provides the direct access of records from file no matter where the record is in the file. Due to which it reduces the unnecessary comparisions. This technique uses a hashing function say h which maps the key with the corresponding key address or location.

Hashing is a different approach to searching which calculates the position of the key in the table based on value of key. It is a method and useful technique to implement dictonaries. This method is used to perform searching, insertion and deletion at a faster rate. A function called Hash function is used to ~~perform searching~~ compute and return position of the record instead of searching with comparisions. The data is stored in an array called a Hash table. The mapping of keys to indices of a hash table is known as hash function. The major requirement of hash function is to map equal keys to equal indices.

Given a key, the algorithm computes an index that suggests where the entry can be found:

Index = f(key, array_size)

The value of index is determined by 2 steps:

→ hash = hash_func(key)

→ index = hash % array_size.

⊛. Hash Function:

A function that transforms a key into a table index is called a hash function. The values returned by a hash function are called hash values or simply hashes. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1. Assume that we have the set of integer items 54, 26, 93, 17, 77 and 31. Our hash function will simply take an item and divide it by the table size, returning the remainder as its hash value

$h(item) = item \% 11$.

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Table: Simple Hash Function Using Remainders

Once the hash values have been computed, we can insert each item into the hash table at the designated position as below:

0	1	2	3	4	5	6	8	9	10
77	None	None	None	26	93	17	None	31	54

fig. Hash table with six items

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present.

⊛ Hash table:

Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisions and then searching time is dependent upon the size of the hash table.

⊛ Types of hash function:

1). Division → In division method hash function is dependent upon the remainder of a division. For example:- If the record 52, 68, 99, 84 is to be placed in a hash table and let us take the table size is 10. Then;

$h(key) = record \% table\ size.$

So, $2 = 52 \% 10$

$8 = 68 \% 10$

$9 = 99 \% 10$

$4 = 84 \% 10.$

2) Mid Square → In mid-square method firstly key is squared and then mid part of the result is taken as the index. For example:- Consider that if we want to place a record of 3101 and the size of table is 1000. So.

$3101 * 3101 = 9616201$

i.e, $h(3101) = 162$ (middle 3 digits).

3) Digit Folding → In this method the key is divided into seperate parts and by using some simple operations these parts are combined to produce a hash key. For example: consider a record of 124-65512 then it will be divided into parts i.e, 124, 655, 12. After dividing the parts combine these parts by adding it. $H(key) = 124 + 655 + 12 = 791.$

4) Hash collision → In computer science, a collision or clash is a situation that occurs when two distinct pieces of data have the same hash value. Collisions are unavoidable whenever members of a very large set (such as all possible person names, or all possible computer files) are mapped to a relatively short bit string. This is similar to an instance of pigeonhole principle. The impact of collisions depend on the application. When hash functions and fingerprints are used to identify similar data, such as homogenous DNA sequences or similar audio files, the functions are designed so as to maximize the probability of collision between distinct but similar data.

5) Collision Resolution → When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. If the hash function is perfect collisions will never occur. Some popular methods for minimizing collision are:-

i) Open addressing
   - → Linear probing
   - → Quadratic probing
   - → Double hashing

ii) Rehashing

iii) Chaining

iv) Hashing using buckets etc.

i) Open Addressing → In open addressing, when a data item can't be placed at the index calculated by hash function another location in the array is sought. It has following 3 methods:

(a) Linear probing → A hash table in which a collision is resolved by putting the item in the next empty place within the occupied array space is called linear probing. The disadvantage of this process is clustering problem.

Example: Insert keys {89, 49, 18, 58} with the hash function $h(x) = x \bmod 10$ using linear probing.

Solution:

when $x = 89$

$h(89) = 89\%10 = 9$

Insert key 89 in hash-table in location 9.

when $x = 49$

$h(49) = 49\%10 = 9$ (Collision occur)

So insert key 49 in hash-table in next possible vacant location of 9. (i.e 0 position. Since array size is 10, with location 0 to 9. So, after 9, zero comes).

when $x = 18$

$h(18) = 18\%10 = 8$

Insert key 18 in hash-table in location 8.

when $x = 58$

$h(58) = 58\%10 = 8$ (Collision occurs)

Insert key 58 in hash-table in next vacant location of 8 i.e, 1 (since 9, 0 are already containing values).

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

we can write None inside empty buckets or leave empty as it is

(b) Quadratic Probing: It eliminates the primary clustering problem that take place in a linear probing. When collision occur then the quadratic probing works as follows:

$(\text{Hash value} + 1^2)\%$ table size.

If there is again collision occurs then ~~thre~~ there exist rehashing.

$(\text{Hash value} + 2^2)\%$ table size.

In general in ith collision

$$h_i(x) = (\text{hash value} + i^2)\% \text{ table size.}$$

Example:- Insert keys {89, 49, 18, 58} with the hash-table size 10 using quadratic probing.

Solution:

when $x=89$

$h(89) = 89\%10 = 9$

Insert key 89 in hash-table in location 9

when $x=49$

$h(49) = 49\%10 = 9$ (Collision occurs)

So use following hash function,

$h_1(49) = (49+1)\%10 = 0$

Hence insert key 49 in hash-table in location 0.

when $x=18$

$h(18) = 18\%10 = 8$

Insert key 18 in hash-table in location 8.

when $x=58$.

$h(58) = 58\%10 = 8$ (collision occur)

So use following hash function,

$h_1(58) = (58+1)\%10 = 9$

Again collision occurs use again the following hash function

$h_2(58) = (58+2^2)\%10 = 2$

Insert key 58 in hash-table in location 2.

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

© **Double hashing:** It is a next method to quadratic probing that eliminates the primary clustering problem take place in a linear probing. When collision occur double hashing define new function as follows:-

$$h_2(x) = R - (x \bmod R)$$

where, R is a prime number smaller than hash table size.

The new element can be inserted in the position by using function,

$$\text{Position} = (\text{original hash value} + 1 * h_2(x))\% \text{table size}$$

We can insert keys using these formulas in hash-table similarly as before.

⊛. **Rehashing:** As its name suggests Rehashing means hashing again. Load factor is $\lambda = \frac{n}{N}$ where, n is number of entries and N is number of buckets. Load factor must be smaller than 1 (i.e $\lambda < 1$ or $N > n$). When $\lambda > 1$, we increase the number of buckets which is called rehashing. Rehashing includes increment of buckets and modify hash function.

⊛. **Chaining:** Chaining is an alternative method for handling the collision problem by allowing each slot to hold a refrence to collection (or chain) of items. It allows many items to exist at the same location in the hash table. Figure below shows the items as they are added to a hash table that uses chaining to resolve collisions.

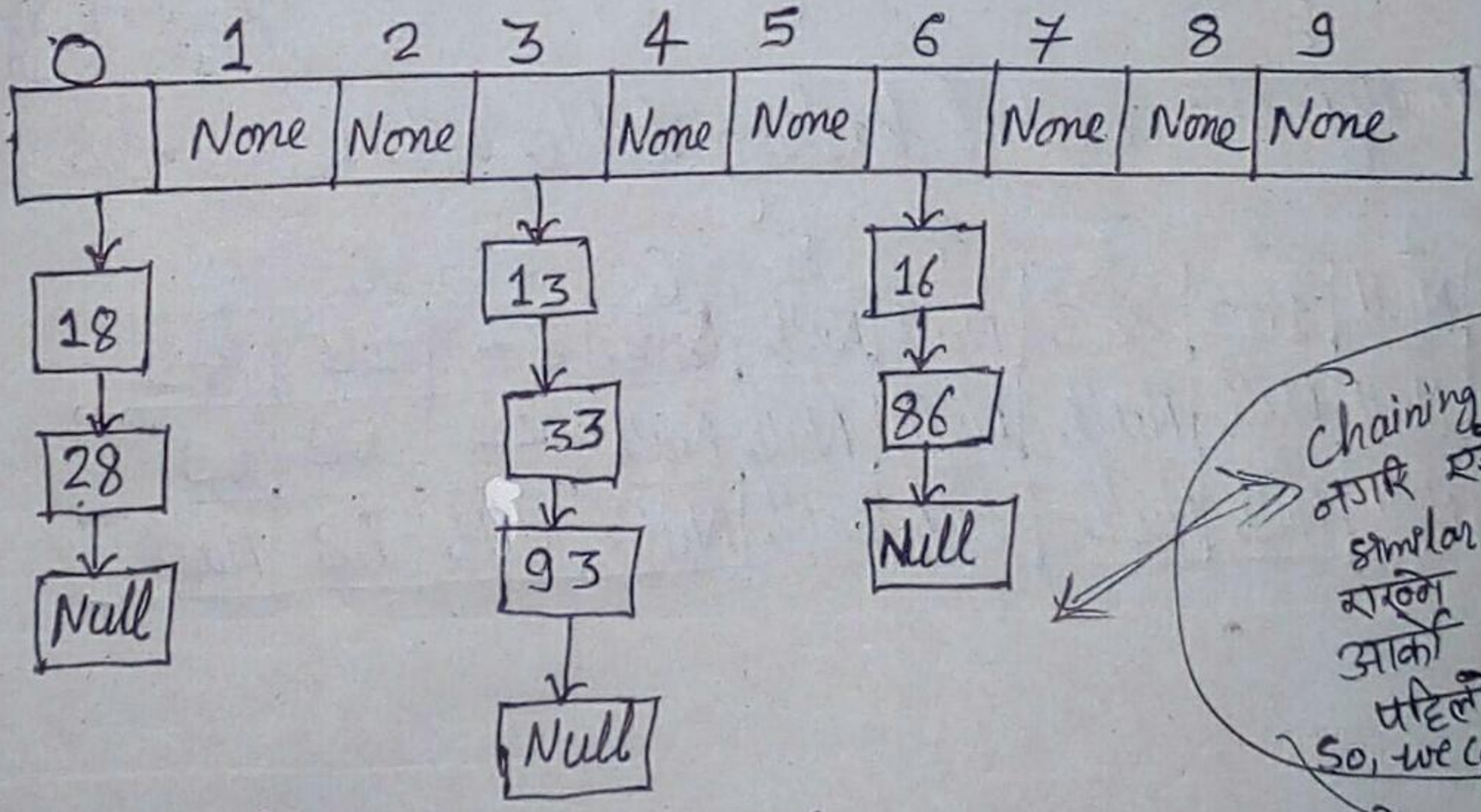

fig. Collision Resolution with chaining

The advantage is that there are likely to be many fewer items on each slot, so the search is perhaps more efficient.

⊛ **Hashing using Buckets/Bucket Addressing:-** It is also an another solution to the collision problem to store colliding elements in the same position in the table. This can be achieved by associating a bucket with each address. A bucket is a block of space large enough to store multiple items.

Example :- Insert keys { 102, 18, 49, 58, 69, 87, 88, 77, 83, 120 } with the hash_table size 10 using bucket hashing.

Solution:

When $x = 102$

$h(102) = 102 \% 10 = 2$

Insert key 102 in hash-table in location 2.

when $x = 18$

$h(18) = 18 \% 10 = 8$

Insert key 18 in hash-table in location 8.

⋮

Similarly we can do for all others

I have left but if asked in eocam we should show all like we did for 102 and 18

⋮

And Finally the bucket hashing table will be as follows:-

0	1	2	3	4	5	6	7	8	9
120	Null	102	83	Null	Null	Null	87	18	49
Null	Null	Null	Null	Null	Null	Null	77	58	69
Null	Null	Null	Null	Null	Null	Null	Null	88	Null