Assignment :- 4

(1) Write an algorithm for selection sort and sort the following elements.

77, 33, 44, 11, 88, 22, 66, 55.

- Algorithm :-

```
Selection-sort(A)
for i ← 1 to n-1 do
        min ← i;
        for j ← i+1 to n do
            if A[j] < A[i] then
                min ← j
        if min != i then
            temp ← A[i]
            A[i] ← A[min]
            A[min] ← temp.
```

[77, 33, 44, 11, 88, 22, 66, 55]

→ Initial array: [77, 33, 44, 11, 88, 22, 66, 55]
→ First pass: min element = 11,
swap it with 77.
⇒ [11, 33, 44, 77, 88, 22, 66, 55]
→ Second pass: min element = 22,
swap it with 33.
⇒ [11, 22, 44, 77, 88, 33, 66, 55]
→ Third pass: min element ~~is~~ 33,
swap it with 44
⇒ [11, 22, 33, 77, 88, 44, 66, 55]

→ fourth pass : Min element is 44, swap it with 77

⇒ [11, 22, 33, 44, 88, 77, 66, 55].

→ fifth Pass : Min element is 55, swap it with 88.

⇒ [11, 22, 33, 44, 55, 77, 66, 88].

→ sixth pass : Min element is 66, swap it with 77.

⇒ [11, 22, 33, 44, 55, 66, 77, 88]

No further swap is needed.

Sorted array : [11, 22, 33, 44, 55, 66, 77, 88].

(2) Write a 'c' program for selection sort.

```
# include <stdio.h>
void main ()
{
    int array [100], n, i, j, min, temp;
    printf (" Enter no. of elements \n");
    scanf (" %d", &n);
    printf (" Enter %d integers \n", n);
    for (i=0 ; i<n ; i++)
    {
     scanf (" %d ", &array [i]);
    }
    for (i=0 ; i<(n-1); i++)
    {
     min = i;
     for (j= i +1 ; j<n ; j++)
     {
       if (array [min] > array [j])
```

```
            min = j;
        }
        if (min != i)
        {
            temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }
    }
    printf("Sorted list in ascending order:\n");
    for(i=0; i<n; i++)
    {
        printf("%d\n", array[i]);
    }
    getch();
}
```

(3) Compare sequential and binary search methods.

Sequential ~~logic~~ Search	Binary search
→ A search algorithm that checks each element in a list sequentially from the beginning until the desired element is found.	→ A search algorithm that divides the sorted list into halves to locates an element.
→ Works on both sorted and unsorted data.	→ Requires the data to be sorted.

→ O(1) (if the target element is the first element).	→ O(1) (if the middle element is the target)
→ O(n) → Time complexity (Average case)	→ O(log n) → Time complexity (Average case).
→ O(n) (if the target is the last element or absent	→ O(log n).
→ Space complexity O(1)	→ O(1) → space complexity
→ Inefficient for large datasets	→ Highly efficient for large, sorted datasets.
→ No preprocessing required	→ Requires the data to be sorted first
→ When the list is small or unsorted it is Best use case.	→ When the list is large and sorted it is Best use case.
→ Compares each element with the target	→ Compares the middle element and halves the list iteratively.
→ Works in any type of list	→ Only applicable to sorted lists.

(4) Write an algorithm for quick sort and apply quick sort for the following data:
9, 7, 5, 11, 12, 2, 14, 3, 10, 6.

- Algorithm for quick sort :-

```
Procedure pivot (T[i...j]; var l)

{ permutes the elements in array T[i....j] and
returns a value l such that, at the end,
i<=l<=j, T[k]<=P for all i<=k<l, T[l]=P,
and T[k]>P for all l<k<=j, where P
is the initial value T[i] }
  P ← T[i]
  k ← i; l ← j+1
  Repeat k = k+1 until T[k] >P
  Repeat l = l-1 until T[l] ≤P
  while k<l do
      Swap T[k] and T[l]
      Repeat k → k+1 until T[k]>P
      Repeat l → l-1 until T[l] ≤P
  swap T[i] and T[l]

Procedure quicksort (T[i....j])
{ sort sub array T[i....j] into non decreasing
  order }
if j-i is sufficiently small then insert(T[i..j])
else
  pivot (T[i...j], l)
  quicksort (T[i,... l-1])
  quicksort (T[l+1, ...j]
```

• Sorting a given array using quick sort :-
[9, 7, 5, 11, 12, 2, 14, 3, 10, 6]

→ Choose pivot 9
→ Partition : left = [7, 5, 2, 3, 6], middle = [9]
right = [11, 12, 14, 10]
→ Recursively sort left : [2, 3, 5, 6, 7]
→ Recursively sort right : [10, 11, 12, 14]
→ combine :- [2, 3, 5, 6, 7, 9, 10, 11, 12, 14].

∴ Result (sorted array) :- [2, 3, 5, 6, 7, 9, 10, 11, 12, 14]

(5) Examine the algorithm for Insertion sort and sort the following array:
77, 33, 44, 11, 88, 22, 66, 55.

• Algorithm for Insertion sort :-

→ Start from the second element of the array (index 1).

→ Compare the current element with its predecessor.

→ If the current element is smaller, shift the predecessor to the right.

→ Continue shifting until a smaller element is found or the beginning of the array is reached.

→ Insert the current element at the correct position.

→ procedure InsertionSort(arr):
for i from 1 to n-1:
key = arr[i]
j = i - 1
while j >= 0 and arr[j] > key;
arr[j+1] = arr[j]
j = j - 1
arr[j+1] = key.

- Sorting the given array using insertion sort.

[77, 33, 44, 11, 88, 22, 66, 55]

→ start with the second element (33).
→ Compare 33 with 77, shift 77 to the right.
→ Insert 33 at the beginning.

[33, 77, 44, 11, 88, 22, 66, 55].
→ Move to the third element 44.
→ Compare 44 with 77, no shift needed.

[33, 44, 77, 11, 88, 22, 66, 55]
→ Move to the fourth element (11).
→ Compare 11 with 44, shift 44 to the right.
→ Compare 11 with 33, shift 33 to the right.
→ Insert 11 at the beginning.

[11, 33, 44, 77, 88, 22, 66, 55]
→ Repeat the process for the remaining elements.

∴ Sorted array :- [11, 22, 33, 44, 55, 66, 77, 88].

(6) Search number 50 from the given data using binary search technique. Illustrate the searching process.
[10, 14, 20, 39, 41, 45, 49, 50, 60]

→ Initialize
Array : [10, 14, 20, 39, 41, 45, 49, 50, 60]
Target : 50
Low : 0 (index of first element)
High : 8 (index of last element)

→ Calculate mid
mid = (low + High) / 2 = (0+8) / 2 = 4.
mid element : 41

→ Compare
50 > 41 move to right half
low = mid + 1 = 5
High remains 8

→ Calculate Mid
Mid = (low + High) / 2 = (5+8) / 2 = 6
Mid element : 49.

→ Compare
50 > 49, move to right half
low = mid + 1 = 7
High remains 8

→ Calculate Mid :

Mid = (low + High) /2 = (7 + 8) /2 = 7

Mid element : 50

→ Found

50 == 50, element found at index 7.

(7) Write algorithm for merge sort method and apply merge sort algorithm to the following elements.

20, 10, 5, 15, 25, 30, 50, 35

Algorithm :-

```
MERGE [A, p, q, r]
n1 = q - p + 1
n2 = r - q
let L [1 ... n1+1] and R [1 ... n2+1] be new
                    arrays
for i = 1 to n1
    L[i] = A [p+i-1]
for j = 1 to n2
    R [j] = A [q+j]
L [n1 + 1] = infinite
R [n2 + 1] = infinite
i = 1
j = 1
for k = p to r
        if L[i] ≤ R [j]
                A[k] = L[i]
                i = i+1
        else A [k] = R[j]
                j = j+1
```

Page No:
Date:

Merge sort (A, p, r)
if p < r
then q = [(p+r)/2]
merge sort (A, p, q)
merge sort (A, q+1, r)
merge (A, p, q, r)

- Sorting the array:-
[20, 10, 5, 15, 25, 30, 50, 35]

→ Divide [20, 10, 5, 15, 25, 30, 50, 35] into [20, 10, 5, 15] and [25, 30, 50, 35]
→ Recursively divide [20, 10, 5, 15] into [20, 10] and [5, 15]
→ Recursively divide [20, 10] into [20] and [10]
→ Merge [20] and [10] = [10, 20].
→ Merge [5, 15] = [5, 15].
→ Merge [10, 20] and [5, 15] = [5, 10, 15, 20].
→ Recursively divide [25, 30, 50, 35] into [25, 30] and [50, 35].
→ Recursively divide [25, 30] into [25] and [30]
→ Merge [25] and [30] = [25, 30].
→ Merge [50, 35] = [35, 50]
→ Merge [10, 20] and [5, 15] = [5, 10, 15, 20].
→ Recursively divide
→ Merge [25, 30] and [35, 50] = [25, 30, 35, 50]
→ Merge [5, 10, 15, 20] and [25, 30, 35, 50]
= [5, 10, 15, 20, 25, 30, 35, 50].

(8). Write 'C' program for Bubble sort.

```
#include <stdio.h>
void main()
{
  int array[100], n, i, j, temp;
  printf(" Enter number of elements\n");
  scanf(" %d ", &n);
  printf(" Enter %d integers \n", n);
  for (i=0; i<n; i++)
  {
    scanf(" %d ", &array[i]);
  }
  for (i=0; i<(n-1); i++)
  {
    for (j=0; j<n-(-1); j++)
    {
      if (array[j] > array[j+1])
      {
        temp= array[j];
        array[j] = array[i+1];
        array[j+1] = temp;
      }
    }
  }
printf(" Sorted list in ascending. order :\n");
for (i=0; i<n; i++)
{
    printf(" %d \n", array[i]);
}
  getch();
}
```

(9) Write algorithm for Bubble sort method and apply bubble sort algorithm on following elements.
[10, 5, 3, 20, 15, 25, 17, 60]

→ • Algorithm :-

```
for i ← 1 to n do
  for j ← 1 to n-i do
      if Array [j] > Array [j+1] then
          temp ← Array [j]
          Array [j] ← A [j+1]
          Array [j+1] ← temp.
```

• Sorting an array
[10, 5, 3, 20, 15, 25, 17, 60]

→ Initial array: [10, 5, 3, 20, 15, 25, 17, 60]
→ first pass : swap 10 and 5 : [5, 10, 3, 20, 15, 25, 17, 60]
→ Swap 10 and 3: [5, 3, 10, 20, 15, 25, 17, 60].
→ Swap 10 and 20: [5, 3, 10, 15, 20, 25, 17, 60]
→ Swap 20 and 15 : [5, 3, 10, 15, 20, 25, 17, 60].
→ Swap 25 and 17 : [5, 3, 10, 15, 20, 17, 25, 60]
→ Swap 60 and 25 : [5, 3, 10, 15, 20, 17, 25, 60].

→ Second Pass: swap 5 and 3 : [3, 5, 10, 15, 20, 17, 25, 60]
→ Swap 10 and 15 : [3, 5, 10, 15, 20, 17, 25, 60]
→ Swap 20 and 17 : [3, 5, 10, 15, 17, 20, 25, 60].

→ Third Pass: Swap 15 and 17 : [3, 5, 10, 15, 17, 20, 25, 60].

result :- [3, 5, 10, 15, 17, 20, 25, 60]

(10) Explain linear search algorithm with an example.

→ In computer Science, linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

→ linear search is the simplest search algorithm.

- Algorithm :-

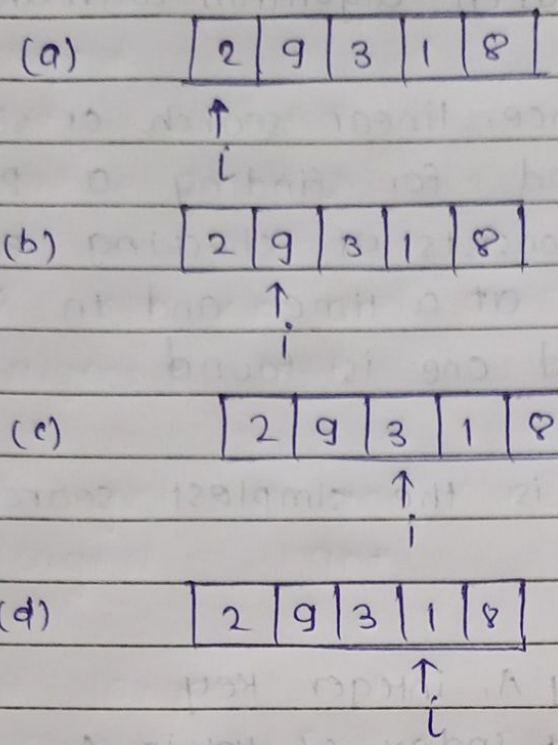
```
# input: Array A, integer key
# output: first index of key in A,
# or -1 if not found:

Linear-search
for i = 0 to last index of A:
    if A[i] equals key:
        return i
return -1.
```

example :-

Search for 1 in the given array :

2	9	3	1	8

comparing value of i^{th} index with element to be search one by one until we get search element or end of the array.



(11) Explain binary search algorithm with an example

→ If we have an array that is sorted, we can use a much more efficient algorithm called a Binary search.

- Algorithm:-

input: Sorted Array A, integer key
output: first index of key in A, or -1 if not found.

```
Binary_search (A, left, right)
while left <= right
    middle = index halfway between left, right
    if D[middle] matches key
        return middle
    else if key less than A[middle]
        right = middle - 1
    else
```

left = middle + 1

return -1.

Example :-

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step-1 ⇒ (middle element is 19 > 6): search in left part

-1 5 6 18 [19] 25 46 78 102 114.

Step-2 ⇒ (middle element is 5 < 6): search in Right part

-1 [5] 6 18

Step-3 ⇒ (middle element is 6 == 6): Element found

[6] 18

(12) Write an algorithm for Heap sort and sort the following data using heap sort.

20, 65, 43, 53, 78, 10, 78, 40, 39, 29.

- Algorithm :-

procedure heapsort (arr):

n = length (arr)

for i from n/2 - 1 to 0:

heapify (arr, n, i)

for i from n-1 to 1:

swap (arr[0], arr[i])

heapify (arr, i, 0).

return arr.

```
procedure heapify (arr, n, i):
  largest = i
  left = 2*i + 1
  right = 2*i + 2

  if left < n and arr[left] > arr[largest]:
    largest = left

  if right < n and arr[right] > arr[largest]:
    largest = right

  if largest != i:
    swap (arr[i], arr[largest])
    heapify (arr, n, largest).
```

- Sorting an Array:-

arr = [20, 65, 43, 53, 78, 10, 78, 40, 39, 29]

Sorted array: [10, 20, 29, 39, 40, 43, 53, 65, 78, 78].

Assignment :- 5.

(1) Explain collision resolution techniques with eg.

→ Collision resolution is the main problem in hashing.

→ If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved.

→ There are several straqtegies for collision resolution. The most commonly used are:

(1) Separate chaining - used with open hashing.

→ In this strategy, a separate list of all elements mapped to the same value is maintained.

→ Separate chaining is based on collision avoidance

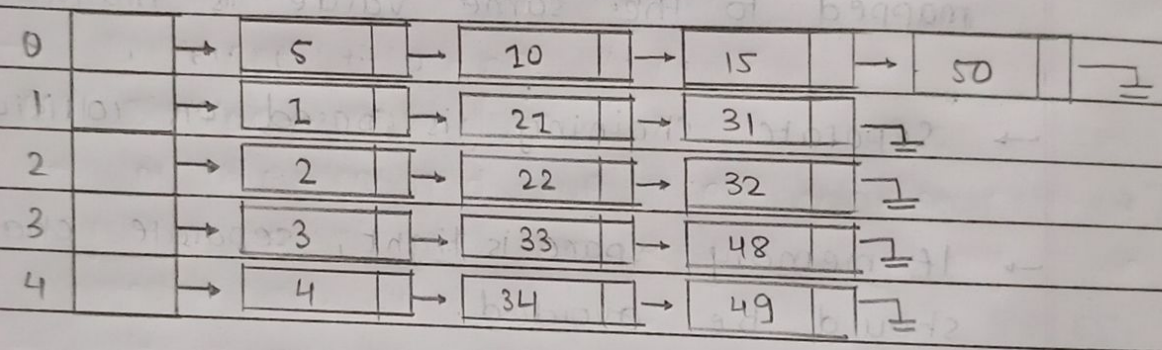
→ If memory space is tight, separate chaining should be avoided.

→ Additional memory space for links is wasted in storing address of linked elements.

→ Hashing function should ensure even distribution of elements among buckets; otherwise the timing behaviour of most operations on hash table will deteriorate.

Example :- The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function. 1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50.

Hash table location	Mapped element
0	5, 10, 15, 50
1	1, 21, 31
2	2, 22, 32
3	3, 33, 48
4	4, 34, 49



(2) Open Addressing :-

→ Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of hashing collision.

→ In open addressing, if a collision occurs, alternate cells are tried until an empty cell is found.

(a) Linear probing :.

→ In linear probing, whenever there is a collision,

cells are searched sequentially (with wraparound) for an empty cell.

→	Empty Table	After 5	After 18	After 55	After 78	After 35	After 15
0							15
1							
2							
3							
4							
5		5	5	5	5	5	5
6				55	55	55	55
7						35	35
8			18	18	18	18	18
9					78	78	78

→ linear probing is easy to implement but it suffers from "primary clustering".

(b) quadratic Probing:-

→ One way of reducing "primary clustering" is to use quadratic probing to resolve collision.

→ Suppose the "key" is mapped to the loca^ j and the cell j is already occupied.

→ In quadratic probing, the location j, (j+1), (j+4), (j+9), ... are examined to find the first empty cell where the key is to be inserted.

→ This table reduces primary clustering.

→ It does not ensure that all cells in the table will be examined to find an empty cell.

→ Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

(c) Double Hashing :

→ This method requires two hashing functions f1(key) and f2(key).

→ Problem of clustering can easily be handled through double hashing.

→ Function f1(key) is known as primary hash function.

(2) List out different hash methods / functions and explain each in detail.

→ Different Hashing functions :-

(1) Division- Method :-

→ In this method we use modular arithmetic system to divide the key value by some integer divisor m

→ It gives us the location value, where the element can be placed.

→ We can write,

$L = (k \bmod m) + 1$

where L => location in table / file.

k => key value

m => table size / number of slots in file.

Suppose, k = 23, m = 10 then

$L = (23 \bmod 10) + 1 = 3 + 1 = 4$; The key whose value is 23 is placed in 4th posn.

(2) Midsquare Methods:-

→ In this case, we square the value of a key and take the number of digits required to form an address, from the middle position of squared value.

→ Suppose a key value is 16, then its square is 256.

→ Now if we want address of two digits, then you select the address as 56.

(3) Folding Method:-

→ Most machines have a small number of primitive data types for which there are arithmetic instructions.

→ Frequently key to be used will not fit easily in to one of these data types.

→ It is not possible to discard the portion of the key that does not fit into such an arithmetic data type.

→ The solution is to combine the various parts of the key in such a way that all parts of the key effect for final result such an operation is termed folding of the key.

→ That is the key is actually partitioned into number of parts, each part having the same length as that of the required address.

→ This is done in two ways:

- Fold-shifting: Here actual values of each parts of key are added.

→ Suppose, the key is 12345678, and the required address is of two digits,

→ Then break the key into: 12, 34, 56, 78.

→ Add these, we get 12+34+56+78 : 180, ignore first 1 we get 80 as location

- Fold Boundary :-

→ Here the reversed values of outer parts of key are added.

→ suppose, the key is : 12345678, and the required address is of two digits,

→ Then break the key into : 21, 34, 56, 87.

→ Add these, we get 21+34+56+87 : 198, ignore first 1 we get 98 as location.

(4) Digit Analysis :-

→ This Hashing function is a distribution-dependent.

→ Here we make a statistical analysis of digits of the key, and select those digits which occur frequently.

→ Then reverse or shifts the digits to get the address.

→ For example, if the key : 9861234.

→ If the statistical analysis has revealed the fact that the third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key.

→ so we get, 62. Reversing it we get 26 as the address.

(5) Length Dependent Method :-

→ In this type of hashing function we use the length of the key along with some portion of the key j to produce the address, directly.

→ In the direct method, the length of the key along with some portion of the key is used to obtain intermediate value.

(6) Algebraic coding :-

→ Here a n bit key value is represented as a polynomial.

→ The divisor polynomial is then constructed based on the address range required.

→ The modular division of key- polynomial by divisor polynomial, to get the address-polynomial.

→ let $f(x)$ = polynomial of n bit key = $a_1 + a_2 x + \dots + a_n x^{n-1}$.

→ $d(x)$ = divisor polynomial = $x^1 + d_1 + d_2 x + \dots d_1 x^{l-1}$

→ Then the required address polynomial will be $f(x) \bmod d(x)$.

(7) Multiplicative Hashing :-

→ This method is based on obtaining an address of a key, based on the multiplicn value.

→ If k is the non-negative key, and a constant c, $(0 < c < 1)$, compute kc and $\bmod 1$, which is a fractional part of kc.

→ Multiply this fractional part by m and take a floor value to get the address.

→ $h, m(kc \bmod 1)$

→ $0 < h(k) < m$

(3) What is hashing ? what are the qualities of a good hash function ? Explain internal & external hashing with eg.

→ Sequential search requires, on the average O(n) comparisons to locate an element.

→ So many comparisons are not desirable for a large database of elements.

→ Binary search requires much fewer comparisons on the average O(logn) but there is an additional requirement that the data should be sorted.

→ There is another widely used technique for storing of data called hashing.

→ It does away with the requirement of keeping data sorted and its best case timing complexity is ★ of constant order O(1).

→ In its worst case, hashing algorithm starts behaving like linear search.

- Characteristics of a good hash function:-

→ A good hash function avoids collisions.

→ A good hash function tends to spreads keys evenly in the array.

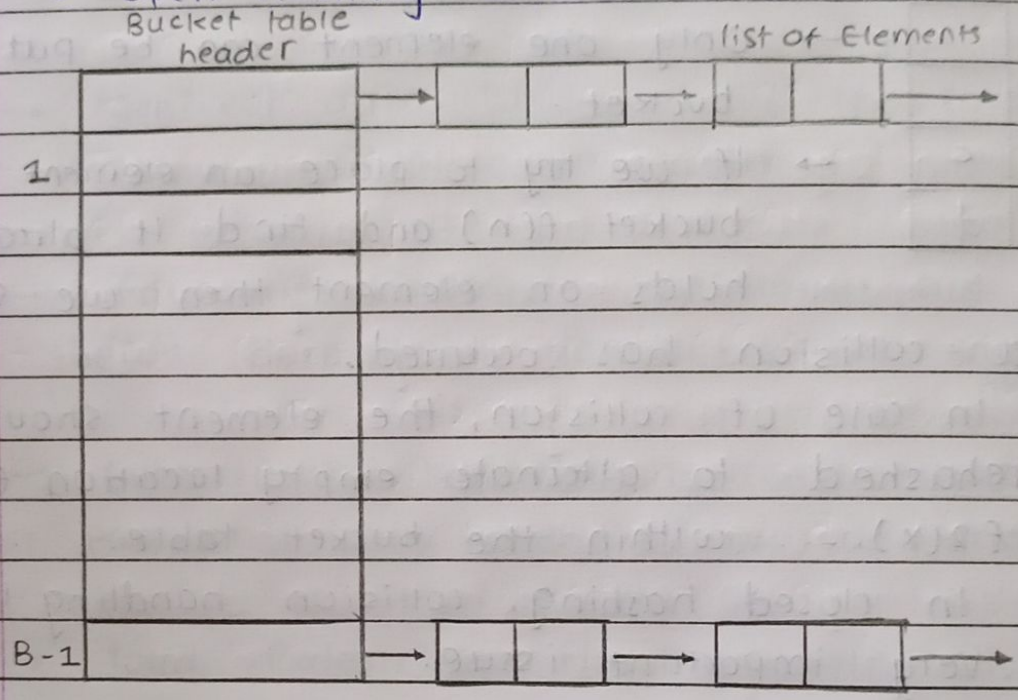
→ A good hash function is easy to compute

- Open hashing or External hashing :-

→ Open or external hashing, allows records to be stored in unlimited space.

→ It places no limitation on the size of the tables.

→ Open hashing Data structure :-



→ The basic idea is that the records [elements] are partitioned into B classes, numbered 0, 1, 2.. B-1

→ A Hashing function f(x) maps a record with key n to an integer value between 0 and B-1.

→ Each bucket in the bucket table is the head of the linked list of records mapped to that bucket.

• close hashing or internal hashing:-

→ closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.

0	b
1	
2	
3	
4	c
5	d

→ A closed hash table keeps the element in the bucket itself.

→ Only one element can be put in the bucket.

→ If we try to place an element in the bucket f(n) and find it already holds an element, then we say that a collision has occurred.

→ In case of collision, the element should be rehashed to alternate empty location f1(x), f2(x), ... within the bucket table.

→ In closed hashing, collision handling is a very important issue.

(4) List out applications of hashing.

→ Data storage and Retrieval: Hash tables enable constant time complexity for insertions, deletions, and lookups, making them efficient for storing key-value pairs.

→ Cryptography: Hashing algorithms like SHA-256 are used in cryptography to generate a fixed-size hash value from input data.

→ It ensures data integrity by verifying if data has been altered.

→ Digital signature: Hash functions are used to generate message digests, ensuring that signatures are unique to the data and verifying the authenticity of documents or software.

→ Data Deduplication: Hashing is used to identify and eliminate duplicate data in storage systems.

→ Identical data blocks have the same hash value, making deduplication faster.

→ Password storage: Passwords are often hashed before storage in databases for security purposes.

→ Even if the database is compromised, the hashed passwords are hard to reverse-engineer.

→ load Balancing: Hashing helps distribute workloads evenly across servers by assigning requests based on the hash of their inputs, ensuring efficient resource utilization.

→ Data Indexing: In databases, hash indexes allow fast data retrieval, especially when searching by unique keys.

→ Checksums and Data Integrity: Hash functions are used to compute checksums or fingerprints for data blocks, ensuring that transmitted or stored data remains unaltered.

→ Blockchain :- Hashing is fundamental to blockchain technology, securing transactions and creating links between blocks, making the chain tamper-resistant.

→ Caching: Hashing is used in web caches and content distribution network (CDNs) to map web requests to cached content for faster access.

(5) Explain significance of data structure.

→ Data structure are fundamental components of computer science and programming bcz they define how data is organized, stored, and accessed efficiently.

→ The choice of data structure affects the efficiency of operations such as data access, insertion, deletion, and searching.

→ For eg, hash tables allow constant-time lookup, while linked lists may require linear time.

→ Choosing the appropriate data structure can improve performance significantly.

→ Data structures help in organizing data logically.

→ Arrays, linked lists, stacks, queues, and trees are all used to organize data differently based on how it needs to be processed or stored.

→ This organization improves data handling in complex programs.

→ Efficient use of memory is crucial in programming.

→ Data structures like arrays require contiguous memory blocks, while dynamic data structures like linked lists can grow as needed, optimizing memory usage.

→ Many algorithms are closely tied to specific data structures.

→ For example, sorting algorithms like quicksort or mergesort require arrays, while graph traversal algorithms need trees or graphs.

→ Certain data structures are better suited for specific types of data manipulation.

→ For example, stacks are used for undo/redo functionality in applications, and trees are useful for hierarchical data, like file systems.

→ As applications grow in size and complexity, efficient data structures ensure that the program remains scalable.

→ Data structures like balanced trees or hash maps ensure that even with large datasets, operations remain fast.

(6) Build a chained hash table of 10 memory locations. Insert the keys 131, 3, 4, 21, 61, 24, 7, 97, 8, 9 in hash table using chaining. Use $h(k) = k \bmod m$. $(m = 10)$.

→ We have 10 memory locations.

→ The ~~locatio~~ hash table will store linked lists at each location to handle collisions via chaining.

→ Insert the keys:
We will insert the following keys: 131, 3, 4, 21, 61, 24, 7, 97, 8, 9. using the hash function $h(k) = k \bmod 10$.

→ 131: $h(131) = 131 \bmod 10 = 1$
Insert 131 at index 1.

→ 3: $h(3) = 3 \bmod 10 = 3$.
Insert 3 at index 3.

→ 4: $h(4) = 4 \bmod 10 = 4$.
Insert 4 at index 4.

→ 21: $h(21) = 21 \bmod 10 = 1$.
Insert 21 at index 1. (chaining with 131,)

→ 61: $h(61) = 61 \bmod 10 = 1$
Insert 61 at index 1 (chaining with 131, 21)

→ 24: $h(24) = 24 \bmod 10 = 4$
Insert 24 at index 4 (chaining with 4).

→ 7: $h(7) = 7 \bmod 10 = 7$.
Insert 7 at index 7.

→ 97: $h(97) = 97 \bmod 10 = 7$
Insert 97 at index 7 (chaining with 7).

→ 8 : h(8) = 8 mod 10 = 8.
Insert 8 at index 8.
→ 9 : h(9) = 9 mod 10 = 9.
Insert 9 at index 9.

→ Hash Table Representation.

Index	chain
0	Null
1	131 → 21 → 61
2	Null
3	3
4	4 → 24
5	Null
6	Null
7	7 → 97
8	8
9	9

(7) Consider the hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81 and 101 into hash table Take $c1 = 1$ and $c2 = 3$.

→ $h'(k,i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$

→ $h(k) = k \bmod m$
→ i is the number of attempts.
→ c_1 and c_2 are constants.
→ $m = 10$ is the size of the hash table.

Page No:
Date:

Keys: 72, 27, 36, 24, 63, 81, 101.

→ Insert key 72:
$h(72) = 72 \mod 10 = 2$

→ Insert key 27:
$h(27) = 27 \mod 10 = 7$

→ Insert key 36:
$h(36) = 36 \mod 10 = 6$

→ Insert key 24:
$h(24) = 24 \mod 10 = 4$

→ Insert key 63:
$h(63) = 63 \mod 10 = 3$

→ Insert key 81:
$h(81) = 81 \mod 10 = 1$

→ Insert key 101:
$h(101) = 101 \mod 10 = 1$

→ for i = 1:
$h'(101, 1) = (1 + 1 \cdot 1 + 3 \cdot 1^2) \mod 10 = (1 + 1 + 3) \mod 10 = 5$

→ Index 5 is empty, so place 101 at Index 5.

→ Hash Table:-

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