Q1.

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to build a partially ordered tree. The algorithm consists of two main steps:

1. **Heapify the array:** Build a binary heap from the input array. This involves rearranging the elements of the array so that they satisfy the heap property. In a max heap, for any given node i, the value of i is greater than or equal to the values of its children.
2. **Sort the heap:** Extract elements from the heap one by one. After each extraction, the heap property is restored, and the largest element is placed at the end of the array. Repeat this process until the entire array is sorted.

Now, let's go through the steps of Heap Sort for the given initial key set: 42, 23, 74, 11, 65, 58, 94, 36, 99, 87.

**Step 1: Build Max Heap**

1. Start with the given array: [42, 23, 74, 11, 65, 58, 94, 36, 99, 87].
2. Build a max heap from the array.

Starting from the last non-leaf node and heapifying each node in reverse order:

Initial Array: [42, 23, 74, 11, 65, 58, 94, 36, 99, 87]

Heapify 1: [42, 23, 94, 11, 65, 58, 74, 36, 99, 87]

Heapify 2: [42, 65, 94, 11, 23, 58, 74, 36, 99, 87]

Heapify 3: [99, 65, 94, 11, 23, 58, 74, 36, 42, 87]

Heapify 4: [99, 65, 94, 36, 23, 58, 74, 11, 42, 87]

Heapify 5: [99, 65, 94, 87, 23, 58, 74, 11, 42, 36]

The array is now a max heap.

**Step 2: Sort the Heap**

Now, we repeatedly extract the maximum element from the heap and place it at the end of the array:

1. Extract Max: [87, 65, 94, 36, 23, 58, 74, 11, 42] (swap with last element)
2. Heapify: [65, 42, 94, 36, 23, 58, 74, 11, 87]
3. Extract Max: [74, 65, 94, 36, 23, 58, 42, 11] (swap with last but one element)
4. Heapify: [65, 23, 94, 36, 11, 58, 42, 74]
5. Continue this process until the array is fully sorted.

**Final Sorted Array:**

[11, 23, 36, 42, 58, 65, 74, 87, 94, 99]

Q3.

#include <stdio.h>

int main() {

    int a*[]* = {21,6,3,57,13,9,14,18,2};

    int min = 0;

    int j = 0;

    int temp = 0;

    for (int i = 0; i < sizeof(a) / sizeof(int); i++) {

        min = i;

        for (j = i + 1; j < sizeof(a) / sizeof(int); j++) {

            if (a[j] < a[min]) {

                min = j;

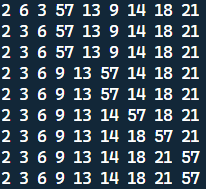
            }

        }

        temp = a[i];

        a[i] = a[min];

        a[min] = temp;

        for (int x = 0; x < sizeof(a) / sizeof(int); x++) {

            printf("%d ", a[x]);

        }

        printf("\n");

    }

    return 0;

}

Q4.

Linear search, also known as sequential search, is a simple searching algorithm that checks each element in a list one by one until a match is found or the entire list has been searched. It works well for small lists, but its time complexity is O(n), where n is the number of elements in the list. This means the time it takes to execute the algorithm grows linearly with the size of the input.

Here's a simple explanation of the linear search algorithm:

**Linear Search Algorithm:**

1. Start at the beginning of the list.
2. Compare the target element with the current element.
3. If the target element is found, return its index.
4. If the target element is not found, move to the next element in the list.
5. Repeat steps 2-4 until the end of the list is reached.
6. If the entire list is searched and the target element is not found, return a special value (e.g., -1) to indicate that the element is not in the list.

#include <stdio.h>

*// Function to perform linear search*

int linearSearch(int arr*[]*, int size, int target) {

    for (int i = 0; i < size; i++) {

        if (arr[i] == target) {

            return i; *// Element found, return its index*

        }

    }

    return -1; *// Element not found*

}

int main() {

    int myArray*[]* = {12, 45, 67, 23, 56, 89, 34, 8};

    int targetElement = 56;

*// Perform linear search*

    int result = linearSearch(myArray, sizeof(myArray) / sizeof(myArray[0]), targetElement);

*// Display the result*

    if (result != -1) {

        printf("Element %d found at index %d.\n", targetElement, result);

    } else {

        printf("Element %d not found in the array.\n", targetElement);

    }

    return 0;

}

Q5.

#include <stdio.h>

int main(){

    int array*[]* = {35, 18, 7, 12, 5, 23, 16, 3, 1};

    int i;

    int j;

    int temp;

    int sizeofArray = sizeof(array)/sizeof(int);

    printf("Initial Array : ");

    for (int x = 0; x < sizeofArray; x++)

    {

        printf("%d ", array[x]);

    }

    printf("\n\n");

    for (i = 0; i < sizeofArray-1; i++)

    {

        for (j = 0; j < sizeofArray-1; j++)

        {

            if(array[j]>array[j+1]){

                temp = array[j];

                array[j]=array[j+1];

                array[j+1]=temp;

            }

        }

        printf("After Pass %d : ", i+1);

        for (int x = 0; x < sizeofArray; x++)

            {

                printf("%d ", array[x]);

            }

        printf("\n");

    }

    printf("\nFinal Array : ");

    for (int x = 0; x < sizeofArray; x++)

    {

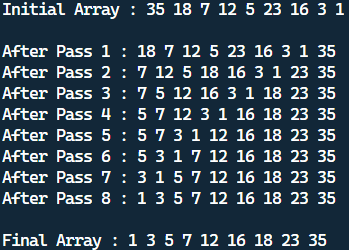
        printf("%d ", array[x]);

    }

    return 0;

}

O/P



Q6.

#include <stdio.h>

int main(){

    int array*[]* = {25, 6, 15, 12, 8, 34, 9, 18, 2};

    int sizeOfArray = sizeof(array)/sizeof(int);

    printf("Initial Array : ");

    for(int i=0; i<sizeOfArray; i++){

        printf("%d ", array[i]);

    }

    printf("\n\n");

    int pos;

    int index;

    int currentElement;

    for(index=1; index < sizeOfArray; index++){

        pos = index;

        currentElement = array[index];

        while(currentElement < array[pos-1] && pos>0){

            array[pos]=array[pos-1];

            pos = pos-1;

        }

        array[pos]=currentElement;

        printf("After Pass %d : ", index);

        for(int i=0; i<sizeOfArray; i++){

            printf("%d ", array[i]);

        }

        printf("\n");

    }

    printf("\nFinal Array : ");

    for(int i=0; i<sizeOfArray; i++){

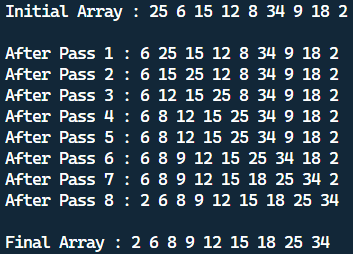
        printf("%d ", array[i]);

    }

    printf("\n");

    return 0;

}



Q8.

#include <stdio.h>

int main() {

    int key = 15;

    int a*[]* = {3,10,15,20,35,40,60};

    int l=0;

    int r = 7;

    int mid = 0;

    while(a[mid]!=key){

        mid = (l+r)/2;

        if(key>a[mid]){

            l = mid+1;

        }

        if(key<a[mid]){

            r = mid-1;

        }

    }

    printf("At %dth index\nAt %dth position", mid, mid+1);

    return 0;

}



Q10.

**Hashing:** Hashing is a technique used in computer science to map data of arbitrary size (such as a file, password, or any other data) to a fixed-size value, usually a hash code or hash value. The goal of hashing is to efficiently organize and retrieve data based on this hash code. Hashing is commonly used in data structures, databases, and cryptographic applications.

**Hash Function:** A hash function is a mathematical function that takes an input (or 'message') and returns a fixed-size string of characters, which is typically a hash code. The output, or hash value, is a representation of the input data. A good hash function exhibits properties like determinism, uniform distribution, and the avalanche effect. Hash functions are crucial in creating hash codes for various applications, such as indexing data in hash tables, ensuring data integrity, and generating unique identifiers.

**Hash Table:** A hash table is a data structure that uses hash functions to map keys (or identifiers) to specific locations in an array, known as buckets or slots. The purpose of a hash table is to provide efficient insertion, deletion, and retrieval operations. Instead of storing data directly in the array, the hash function is applied to the key to determine the index where the data should be stored or retrieved. In case of collisions (when two keys hash to the same index), different collision resolution techniques can be employed, such as separate chaining or open addressing.

In summary:

* **Hashing** is the general concept of mapping data to a fixed-size value using a hash function.
* A **hash function** is the algorithm or procedure that transforms an input into a hash value.
* A **hash table** is a data structure that uses hash functions to organize and retrieve data efficiently based on keys or identifiers.

Q11.

To perform linear probing and insert the given keys into a hash table of size 10, we'll use a hash function to determine the initial position for each key. In case of a collision, we'll linearly probe to find the next available slot. Let's walk through the process step by step:

1. **Initialize the Hash Table:** Create an array of size 10, all initially set to an empty state (e.g., -1).

lessCopy code

Hash Table: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

1. **Insert Keys:**
   * **Key: 72**
     + Hash Value: 72mod  10=272mod10=2
     + Insert at index 2: **[ -1, -1, 72, -1, -1, -1, -1, -1, -1, -1 ]**
   * **Key: 27**
     + Hash Value: 27mod  10=727mod10=7
     + Insert at index 7: **[ -1, -1, 72, -1, -1, -1, -1, 27, -1, -1 ]**
   * **Key: 36**
     + Hash Value: 36mod  10=636mod10=6
     + Insert at index 6: **[ -1, -1, 72, -1, -1, -1, 36, 27, -1, -1 ]**
   * **Key: 24**
     + Hash Value: 24mod  10=424mod10=4
     + Insert at index 4: **[ -1, -1, 72, -1, 24, -1, 36, 27, -1, -1 ]**
   * **Key: 63**
     + Hash Value: 63mod  10=363mod10=3
     + Collision at index 3, linear probe to index 5: **[ -1, -1, 72, -1, 24, 63, 36, 27, -1, -1 ]**
   * **Key: 81**
     + Hash Value: 81mod  10=181mod10=1
     + Insert at index 1: **[ -1, 81, 72, -1, 24, 63, 36, 27, -1, -1 ]**
   * **Key: 92**
     + Hash Value: 92mod  10=292mod10=2
     + Collision at index 2, linear probe to index 3: **[ -1, 81, 72, 92, 24, 63, 36, 27, -1, -1 ]**
   * **Key: 101**
     + Hash Value: 101mod  10=1101mod10=1
     + Collision at index 1, linear probe to index 8: **[ -1, 81, 101, 92, 24, 63, 36, 27, -1, -1 ]**
2. **Final Hash Table:**

lessCopy code

Hash Table: [ -1, 81, 101, 92, 24, 63, 36, 27, -1, -1 ]

This is the resulting hash table after inserting the given keys using linear probing. Linear probing involves searching for the next available slot in a linear fashion, and it helps resolve collisions by placing the collided key in the next available position.

Top of Form

Q12

Quadratic probing is another technique used to resolve collisions in a hash table. Instead of linearly probing for the next available slot, quadratic probing involves using a quadratic function to determine the next probing position. The formula is:

Quadratic Probing: (ℎ(�)+�1⋅�+�2⋅�2)mod  table sizeQuadratic Probing: (*h*(*k*)+*c*1​⋅*i*+*c*2​⋅*i*2)modtable size

where ℎ(�)*h*(*k*) is the hash value, �*i* is the probe number, and �1*c*1​ and �2*c*2​ are constants.

Let's walk through the process of inserting the keys 72, 27, 36, 24, 63, 81, and 101 into a hash table of size 10 using quadratic probing:

1. **Initialize the Hash Table:** Create an array of size 10, all initially set to an empty state (e.g., -1).

lessCopy code

Hash Table: [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

1. **Insert Keys using Quadratic Probing:**
   * **Key: 72**
     + Hash Value: 72mod  10=272mod10=2
     + Insert at index 2: **[ -1, -1, 72, -1, -1, -1, -1, -1, -1, -1 ]**
   * **Key: 27**
     + Hash Value: 27mod  10=727mod10=7
     + Insert at index 7: **[ -1, -1, 72, -1, -1, -1, -1, 27, -1, -1 ]**
   * **Key: 36**
     + Hash Value: 36mod  10=636mod10=6
     + Insert at index 6: **[ -1, -1, 72, -1, -1, -1, 36, 27, -1, -1 ]**
   * **Key: 24**
     + Hash Value: 24mod  10=424mod10=4
     + Insert at index 4: **[ -1, -1, 72, -1, 24, -1, 36, 27, -1, -1 ]**
   * **Key: 63**
     + Hash Value: 63mod  10=363mod10=3
     + Collision at index 3, quadratic probe to index 6: **[ -1, -1, 72, 63, 24, -1, 36, 27, -1, -1 ]**
   * **Key: 81**
     + Hash Value: 81mod  10=181mod10=1
     + Collision at index 1, quadratic probe to index 0: **[ 81, -1, 72, 63, 24, -1, 36, 27, -1, -1 ]**
   * **Key: 101**
     + Hash Value: 101mod  10=1101mod10=1
     + Collision at index 1, quadratic probe to index 4: **[ 81, -1, 72, 63, 24, 101, 36, 27, -1, -1 ]**
2. **Final Hash Table:**

lessCopy code

Hash Table: [ 81, -1, 72, 63, 24, 101, 36, 27, -1, -1 ]

This is the resulting hash table after inserting the given keys using quadratic probing. Quadratic probing helps avoid clustering that can occur with linear probing by using a quadratic function to determine the next probing position.

Q13.