UNIT IV

SEARCHING

Searching is a process used to find the location of a target among list of objects. In case of an array, searching means that given a value we find the location (index) of the first element in the array that contains that value. The algorithm used to search a list depends to a large extent on the structure of the list.

There are two basic searches for array : the sequential search and the binary search. The sequential search can be used to locate an item in any array. The binary search, on the other hand, requires the list to be sorted.

SEQUENTIAL SEARCH (LINEAR SEARCH)

The sequential search is used whenever the list is not ordered. Generally, we use the technique only for small lists or lists that are not searched often.

In the sequential search, we start searching for the target from the beginning of the list, and we continue until we find the target or until we are sure that it not in the list.

Algorithm

1. start
2. declare an array a
3. read n
4. write "input elements "
5. for i = 0 to n do
6. begin
Read a[i]

7. end for
8. read x
call the function

k = linearSearch (int a, int n, int x)

9. if(k==-1)
write" unsuccessful searching"

else

write “element is found at location”,k

10.stop

```
linearSearch(int arr[], int n, int x)
begin
    int i;
    for(i=0; i<n; i++)
    begin
        if(x==a[i])
            return i;
    end for
    return -1;
end
```

//Program to search an element using linear search technique

```
#include<stdio.h>
int linearSearch(int[],int, int);
int main()
{
    int a[30],n,i,x,p;
    printf("Enter size of list :");
    scanf("%d",&n);
    printf("Enter %d element values :",n);
    for(i=0; i<n; i++)
        scanf("%d",&a[i]);
```

```
    printf("Enter element to be searched :");
    scanf("%d",&x);
    p=linearSearch(a,n,x);
    if(p==-1)
        printf("Number not found");
    else
        printf("Number is found %d position",p);
    return 0;
}
int linearSearch(int arr[], int n, int x)
{
    int i;
    for(i=0; i<n; i++)
    {
        if(x==a[i])
            return i;
    }
    return -1;
}
```

Output

Enter size of list : 4

Enter 4 element values : 12 13 14 15

Enter element to be searched : 14

Number is found in 2 position

Time Complexity :

Time complexity of linear search is O(n). At the most n comparisons will be required to search a record in a set of n records.

Advantage

- Linear search is easier to implement than other searching techniques

Disadvantages

- Linear search requires more number of comparisons than other searching techniques.
- Linear search algorithm execution is slow.

BINARY SEARCH

If the array is **sorted**, efficient algorithm named Binary search can be used for searching an element. Binary search starts by testing the data in the element at the middle of the array. This determines if the target is in the first half or second half of the list. If it is in the first half, we do not need to check the second half. In other words, we eliminate half the list. This process is repeated until the target is found or it is not found.

NOTE; In binary search array must be in sorted order

Example 1 : Binary search for **35** in array {11,18,19,22,24,35,37,64,68}

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	
11	**18**	**19**	**22**	**24**	**35**	**37**	**64**	**68**	
11	18	19	22	**24**	35	37	64	68	mid=(0+8)/2 = 4
11	**18**	**19**	**22**	**24**	35	37	64	68	as 35>24, low=4+1=5
11	**18**	**19**	**22**	**24**	35	**37**	64	68	mid=(5+8)/2 = 6
11	**18**	**19**	**22**	**24**	35	37	**64**	**68**	as 35<37, high=6-1=5
11	**18**	**19**	**22**	**24**	**35**	37	**64**	**68**	mid=(5+6)/2 = 5
11	**18**	**19**	**22**	**24**	**35**	**37**	**64**	**68**	as 35==35

Time complexity

The time complexity of binary search is $O(\log_2 n)$. To find a particular record in a set of n records only $\log_2 n$ comparisons are required.

Advantage

- Binary search requires few numbers of comparisons.

Disadvantages

- Binary search works on sorted data only.
- Binary search is little difficult to implement than linear search.

Algorithm

1. start
2. declare an array a
3. read n
4. write "input elements ascending order"
5. *for i = 0 to n do*
   begin

   Read a[i]

   end for
6. read x
7. call the function
8. k = *binarySearch(int a, int n, int x)*
9. *if(k==-1)*
   write" unsuccessful searching"

   else

   write "element is found at location",k
10. stop

binarySearch(int arr[], int n, int x)

```
begin
    int low=0,high=n-1,mid;
    while(low<=high)
    begin
        mid=(low+high)/2;
        if(x<arr[mid])
            high=mid-1;
        else if(x>arr[mid])
            low=mid+1;
        else
            return mid;
    end while
    return -1;
end
```

//Program to search a number in a sorted list using binary search technique

```
#include<stdio.h>
int binarySearch(int[], int, int);
int main()
{
    int a[30],n,i,x,p;
    printf("Enter size of list :");
    scanf("%d",&n);
    printf("Enter %d element values :",n);
    for(i=0; i<n; i++)
        scanf("%d",&a[i]);
    printf("Enter element to be searched :");
    scanf("%d",&x);
```

```
    p= binarySearch (a,n,x);
    if(p==-1)
        printf("Number not found");
    else
        printf("Number is found %d position",p);
    return 0;
}
int binarySearch(int arr[], int n, int x)
{
    int low=0,high=n-1,mid;
    while(low<=high)
    {
        mid=(low+high)/2;
        if(x<arr[mid])
            high=mid-1;
        else if(x>arr[mid])
            low=mid+1;
        else
            return mid;
    }
    return -1;
}
```

Output

```
Enter size of list : 4
Enter 4 element values : 12 13 14 15
Enter element to be searched : 14
Number is found in 2 position
```

HASHING

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.
(or)

- Hashing is a well-known technique to search any particular element among several elements.
- It minimizes the number of comparisons while performing the search.

Advantage-

Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity O(1).

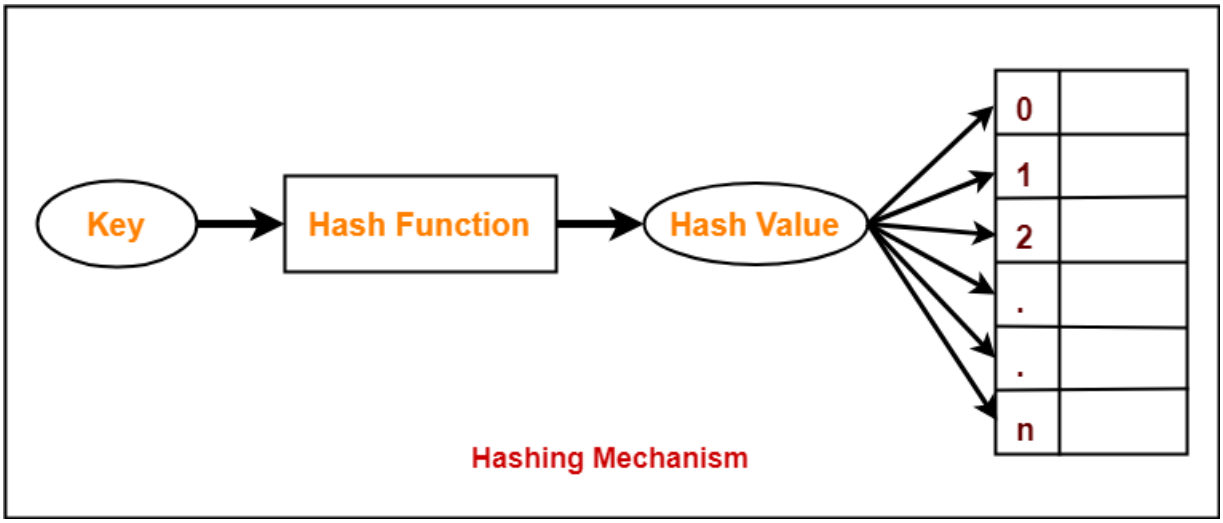
Hashing Mechanism-

In hashing,

- An array data structure called as **Hash table** is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.

Hash Key Value-

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be be stored in the hash table.
- Hash key value is generated using a hash function.



Hash Function-

Hash function is a function that maps any big number or string to a small integer value.

- Hash function takes the data item as an input and returns a small integer value as an output.
- The small integer value is called as a hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.

Types of Hash Functions-

There are various types of hash functions available such as-

1. Mid Square Hash Function
2. Division Hash Function
3. Folding Hash Function etc

It depends on the user which hash function he wants to use.

Properties of Hash Function-

The properties of a good hash function are-

- It is efficiently computable.
- It minimizes the number of collisions.
- It distributes the keys uniformly over the table.

Collision in Hashing-

In hashing,

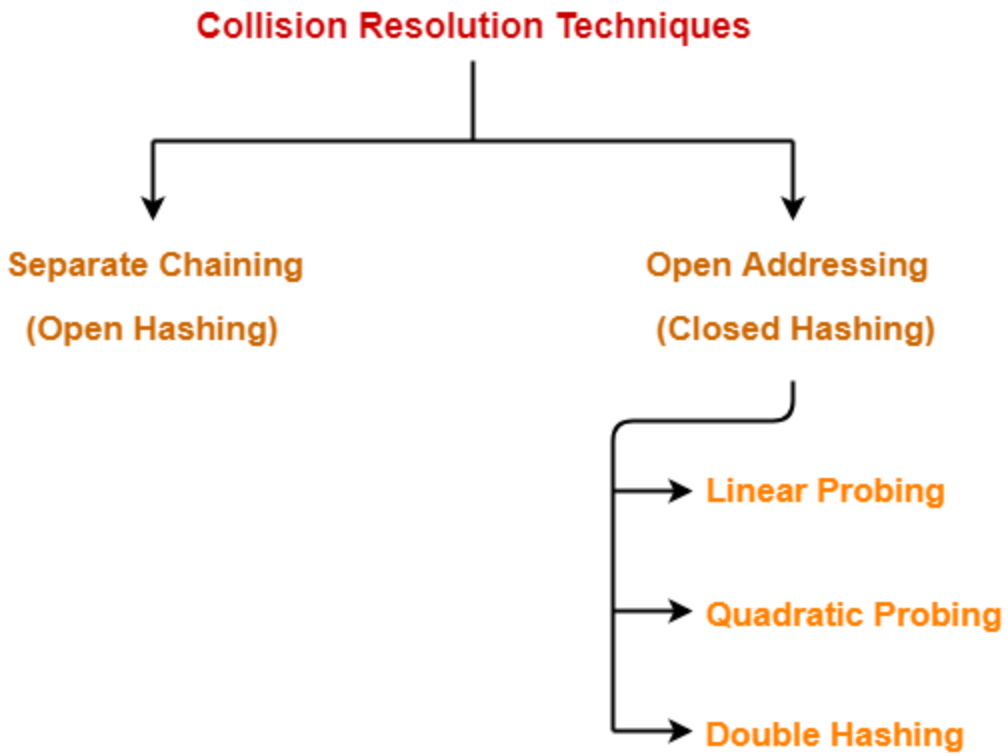
- Hash function is used to compute the hash value for a key.
- Hash value is then used as an index to store the key in the hash table.
- Hash function may return the same hash value for two or more keys.

When the hash value of a key maps to an already occupied bucket of the hash table, it is called as a **Collision**.

Collision Resolution Techniques-

Collision Resolution Techniques are the techniques used for resolving or handling the collision.

Collision resolution techniques are classified as-



1. Separate Chaining
2. Open Addressing

Separate Chaining-

To handle the collision,

- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as **separate chaining**.

Time Complexity-

For Searching-

- In worst case, all the keys might map to the same bucket of the hash table.
- In such a case, all the keys will be present in a single linked list.
- Sequential search will have to be performed on the linked list to perform the search.
- So, time taken for searching in worst case is O(n).

For Deletion-

- In worst case, the key might have to be searched first and then deleted.
- In worst case, time taken for searching is O(n).
- So, time taken for deletion in worst case is O(n).

Load Factor (α)-

Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

If Load factor (α) = constant, then time complexity of Insert, Search, Delete = Θ(1)

PRACTICE PROBLEM BASED ON SEPARATE CHAINING-

Problem-

Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-

50, 700, 76, 85, 92, 73 and 101

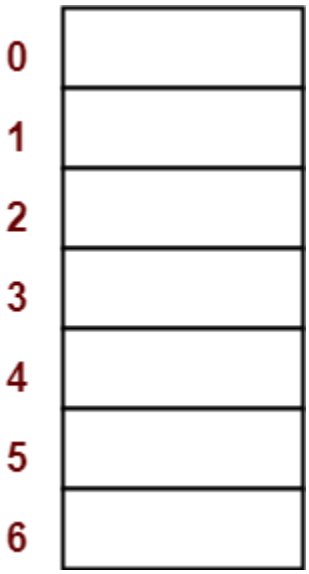
Use separate chaining technique for collision resolution.

Solution-

The given sequence of keys will be inserted in the hash table as-

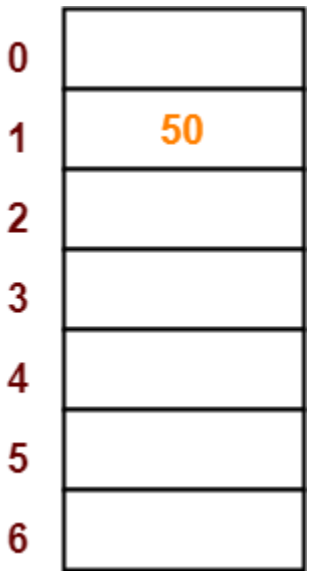
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



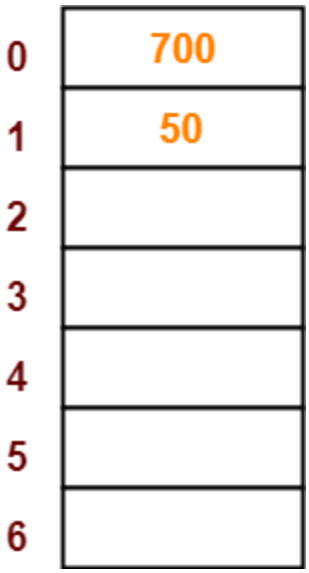
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



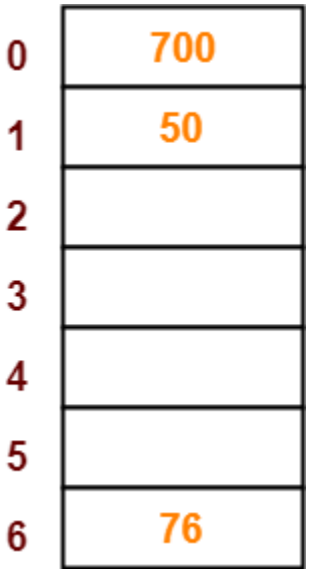
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



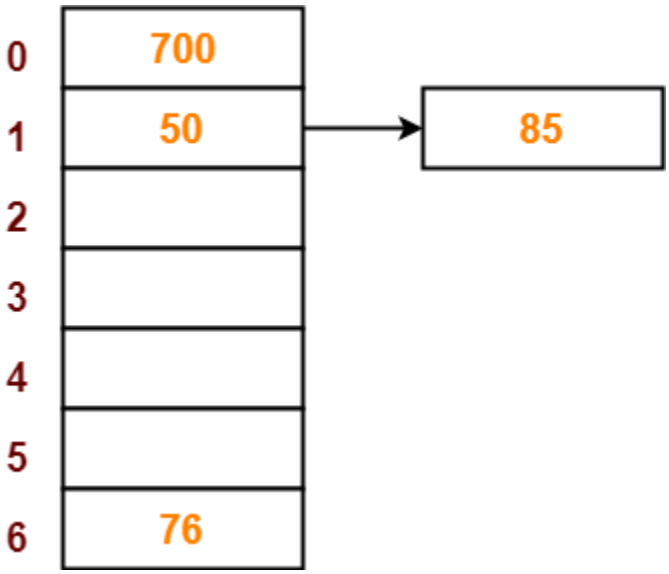
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



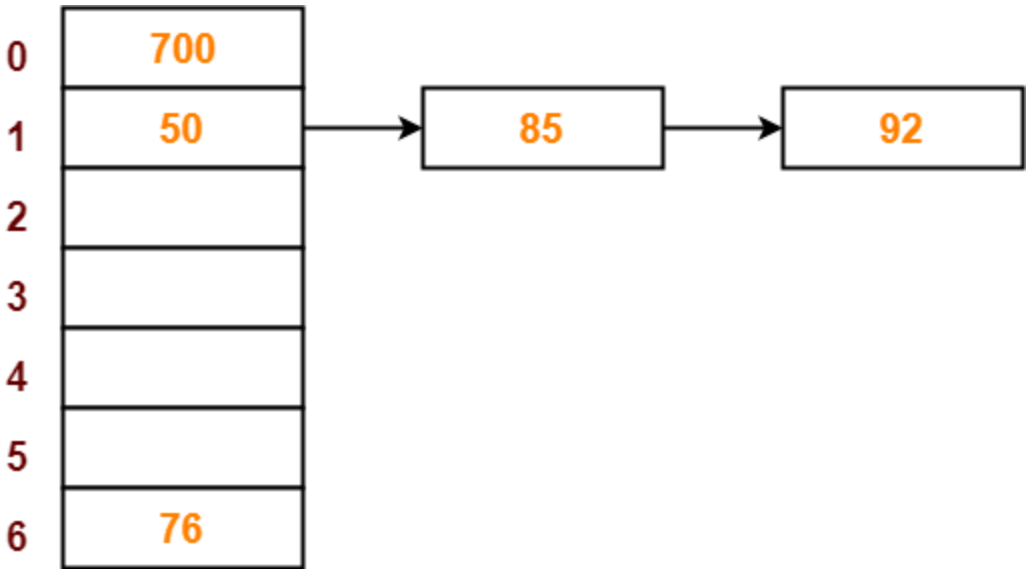
Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



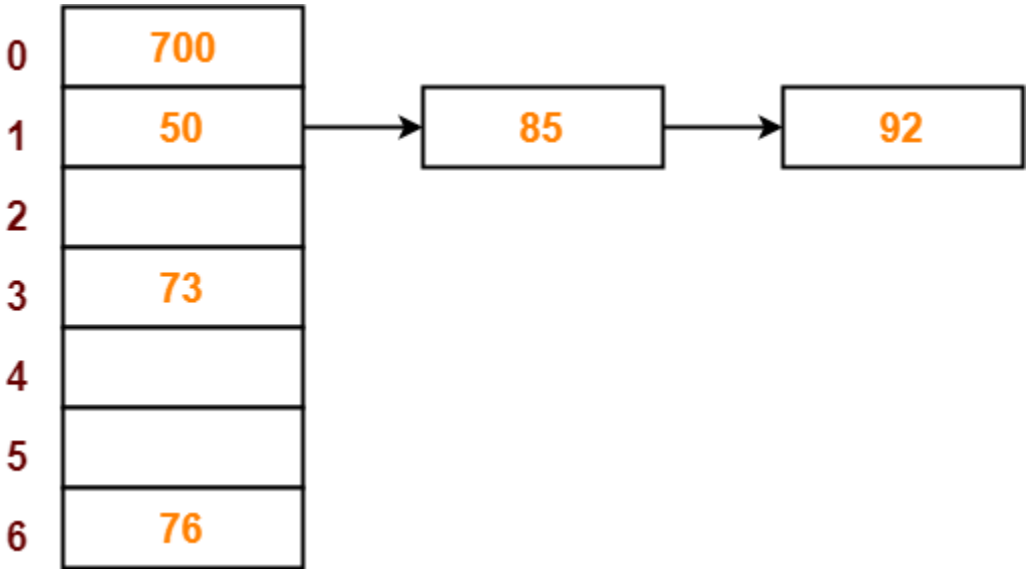
Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



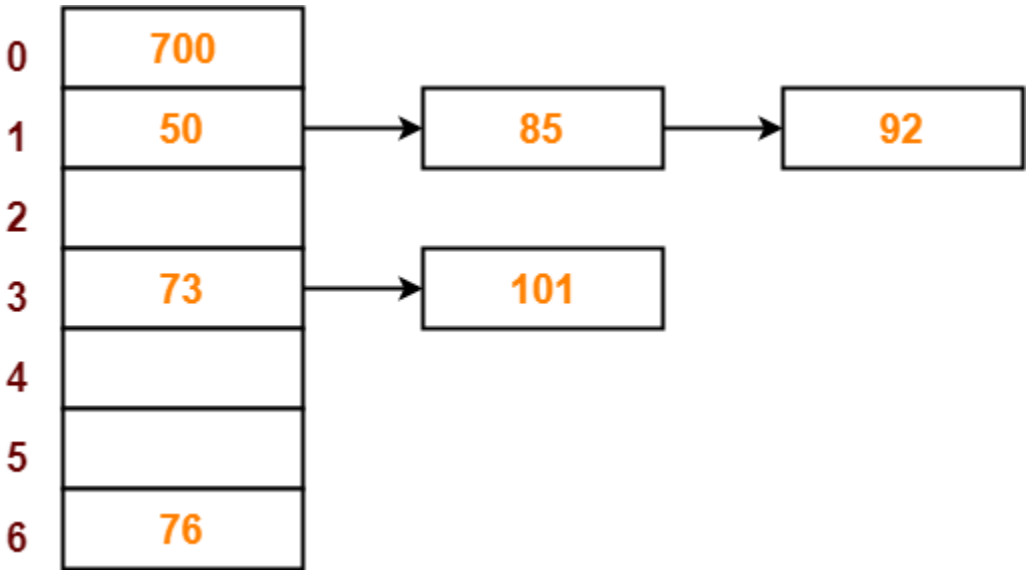
Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- So, key 73 will be inserted in bucket-3 of the hash table as-



Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-



Open Addressing-

In open addressing,

- Unlike separate chaining, all the keys are stored inside the hash table.
- No key is stored outside the hash table.

Techniques used for open addressing are-

- Linear Probing
- Quadratic Probing
- Double Hashing

Operations in Open Addressing-

Let us discuss how operations are performed in open addressing-

Insert Operation-

- Hash function is used to compute the hash value for a key to be inserted.
- Hash value is then used as an index to store the key in the hash table.

In case of collision,

- Probing is performed until an empty bucket is found.
- Once an empty bucket is found, the key is inserted.
- Probing is performed in accordance with the technique used for open addressing.

Search Operation-

To search any particular key,

- Its hash value is obtained using the hash function used.
- Using the hash value, that bucket of the hash table is checked.
- If the required key is found, the key is searched.
- Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
- The empty bucket indicates that the key is not present in the hash table.

Delete Operation-

- The key is first searched and then deleted.
- After deleting the key, that particular bucket is marked as “deleted”.

NOTE-

- During insertion, the buckets marked as “deleted” are treated like any other empty bucket.
- During searching, the search is not terminated on encountering the bucket marked as “deleted”.
- The search terminates only after the required key or an empty bucket is found.

Open Addressing Techniques-

Techniques used for open addressing are-

1. Linear Probing-

In linear probing,

- When collision occurs, we linearly probe for the next bucket.
- We keep probing until an empty bucket is found.

Advantage-

- It is easy to compute.

Disadvantage-

- The main problem with linear probing is clustering.
- Many consecutive elements form groups.
- Then, it takes time to search an element or to find an empty bucket.

Time Complexity-

Worst time to search an element in linear probing is O (table size).

This is because-

- Even if there is only one element present and all other elements are deleted.
- Then, “deleted” markers present in the hash table makes search the entire table.

2. Quadratic Probing-

In quadratic probing,

- When collision occurs, we probe for i^2‘th bucket in i^{th} iteration.
- We keep probing until an empty bucket is found.

3. Double Hashing-

In double hashing,

- We use another hash function hash2(x) and look for i * hash2(x) bucket in i^{th} iteration.
- It requires more computation time as two hash functions need to be computed.

Comparison of Open Addressing Techniques-

	Linear Probing	Quadratic Probing	Double Hashing

Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Conclusions-

- Linear Probing has the best cache performance but suffers from clustering.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double caching has poor cache performance but no clustering.

Load Factor (α)-

Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

In open addressing, the value of load factor always lie between 0 and 1.

This is because-

- In open addressing, all the keys are stored inside the hash table.
- So, size of the table is always greater or at least equal to the number of keys stored in the table.

PRACTICE PROBLEM BASED ON OPEN ADDRESSING-

Problem-

Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-

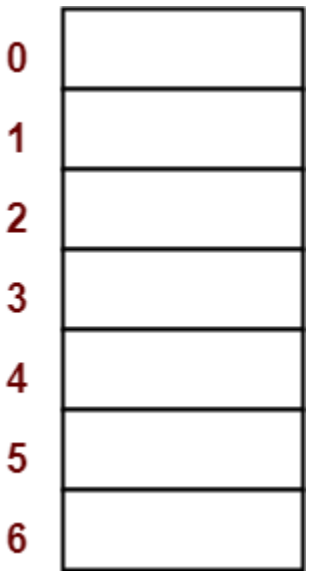
50, 700, 76, 85, 92, 73 and 101

Use linear probing technique for collision resolution.

Solution-

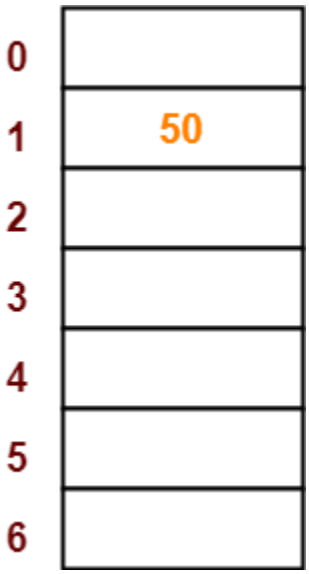
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



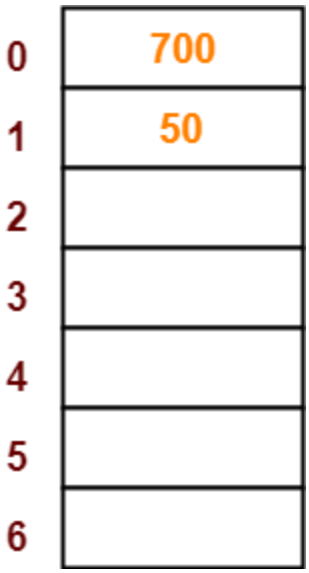
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



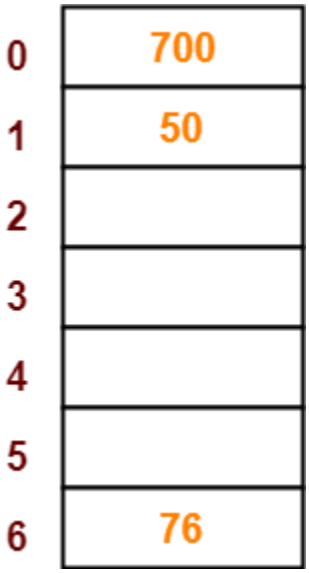
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



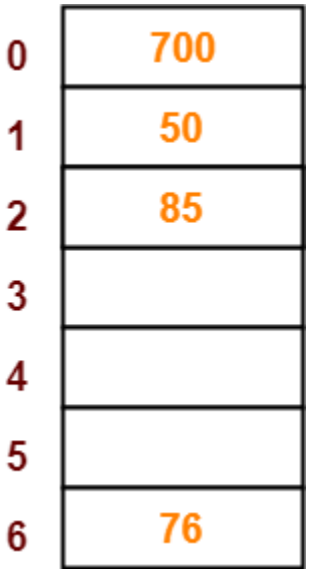
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So, key 85 will be inserted in bucket-2 of the hash table as-



Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So, key 92 will be inserted in bucket-3 of the hash table as-

0	700
1	50
2	85
3	92
4	
5	
6	76

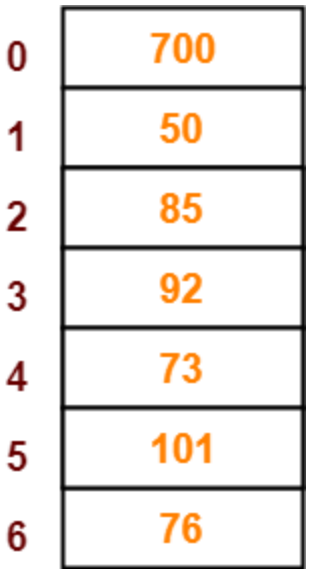
Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-4.
- So, key 73 will be inserted in bucket-4 of the hash table as-

0	700
1	50
2	85
3	92
4	73
5	
6	76

Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So, key 101 will be inserted in bucket-5 of the hash table as-



To gain better understanding about Open Addressing,

Separate Chaining Vs Open Addressing-

Separate Chaining	Open Addressing
Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is present outside the hash table.
The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
Deletion is easier.	Deletion is difficult.
Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.
Some buckets of the hash table are never used which leads to wastage of space.	Buckets may be used even if no key maps to those particular buckets.

Which is the Preferred Technique?

The performance of both the techniques depend on the kind of operations that are required to be performed on the keys stored in the hash table-

Separate Chaining-

Separate Chaining is advantageous when it is required to perform all the following operations on the keys stored in the hash table-

- Insertion Operation
- Deletion Operation
- Searching Operation

NOTE-

- Deletion is easier in separate chaining.
- This is because deleting a key from the hash table does not affect the other keys stored in the hash table.

Open Addressing-

Open addressing is advantageous when it is required to perform only the following operations on the keys stored in the hash table-

- Insertion Operation
- Searching Operation
-

NOTE-

- Deletion is difficult in open addressing.
- This is because deleting a key from the hash table requires some extra efforts.
- After deleting a key, certain keys have to be rearranged.

SORTING

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Sorting algorithms provide an introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, best-, worst- and average-case analysis, time-space tradeoffs, and lower bounds.

Classification:

- **Computational complexity** (worst, average and best behavior) of element comparisons in terms of the size of the list (n). For typical sorting algorithms, a good behavior is $O(n \log n)$ and a bad behavior is $O(n^2)$.
- **Computational complexity of swaps** (for "in place" algorithms).
- **Memory usage** (and use of other computer resources). In particular, some sorting algorithms are "in place". This means that they need only $O(1)$ or $O(\log n)$ memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.
- **Recursion**: Some algorithms are either recursive or non-recursive.
- **Stability**: Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a **comparison sort**. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General methods: **insertion**, **exchange**, **selection**, **merging**, etc.
- **Adaptability**: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Stability

Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list. When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first component:

(4, 2) (3, 7) (3, 1) (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

- (3, 7) (3, 1) (4, 2) (5, 6)
- (3, 1) (3, 7) (4, 2) (5, 6)

Applications of Sorting

- Uniqueness testing

- Deleting duplicates
- Prioritizing events
- Frequency counting
- Reconstructing the original order
- Set intersection/union
- Finding a target pair x, ysuch that x+y= zv
- Efficient searching

INSERTION SORT

Insertion sort is a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(n+d)$, where d is the number of inversions
- More efficient in practice than most other simple quadratic algorithms such as selection sort or bubble sort: the average running time is $n^2/4$, and the running time is linear in the best case
- Stable, i.e., does not change the relative order of elements with equal keys
- In-place, i.e., only requires a constant amount $O(1)$ of additional memory space
- Online, i.e., can sort a list as it receives it

Algorithm

Every iteration of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.

Sorting is typically done in-place. The resulting array after k iterations has the property where the first $k+1$ entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result.

Example: Consider an example of sorting "64 25 12 22 11".

Pseudocode Implementation:

```
insertionSort(array A)
begin
    for i := 1 to length[A] - 1 do
                                begin
                    value := A[i];
            j := i - 1;
```

```
        while j >= 0 and A[j] > value do
                                        begin
                A[j + 1] := A[j];
                j := j - 1;
                                        end;
            A[j + 1] := value;
                                        end;
end;
```

Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$ total, $O(1)$ auxiliary

A Comparison of the $O(n^2)$ Sorting Algorithms

Among simple average-case $O(n^2)$ algorithms, selection sort almost always outperforms bubble sort, but is generally outperformed by insertion sort.

Insertion sort's advantage is that it only scans as many elements as it needs in order to place the k+1st element, while selection sort must scan all remaining elements to find the k+1st element. Experiments show that insertion sort usually performs about half as many comparisons as selection sort. Selection sort will perform identically regardless of the order the array, while insertion sort's running time can vary considerably. Insertion sort runs much more efficiently if the array is already sorted or "close to sorted."

Selection sort always performs $O(n)$ swaps, while insertion sort performs $O(n^2)$ swaps in the average and worst case. Selection sort is preferable if writing to memory is significantly more expensive than reading.

Insertion sort or selection sort are both typically faster for small arrays (i.e., fewer than 10-20 elements). A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" subarrays.

// C program for insertion sort

```
#include <math.h>
#include <stdio.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
```

```
        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

SELECTION SORT

Selection sort is an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

Algorithm

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Effectively, we divide the list into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted.

Example: Consider an example of sorting "64 25 12 22 11".

Java Implementation of the Algorithm

```
void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        if (i != min) {
            int swap = a[i];
            a[i] = a[min];
            a[min] = swap;
        }
    }
}
```

Performance

- Worst case performance: $O(n^2)$
- Best case performance: $O(n^2)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$ total, $O(1)$ auxiliary

How many comparisons does the algorithm need to perform? How many swaps does the algorithm perform in the worst case?

Analysis

Selecting the lowest element requires scanning all n elements (this takes n-1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning all n-1 elements and so on, for (n-1) + (n-2) + ... + 2 + 1 ($O(n^2)$) comparisons. Each of these scans requires one swap for n-1 elements.

```
// C program for implementation of selection sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
```

```
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
```

```
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

RADIX SORT

Algorithm:
For each digit i where i varies from the least significant digit to the most significant digit of a number
Sort input array using countsort algorithm according to ith digit.

We used count sort because it is a stable sort.

Example: Assume the input array is: 10,21,17,34,44,11,654,123
Based on the algorithm, we will sort the input array according to the **one's digit** (least significant digit).

0:			10
1:	21		11
2:			
3:			123
4:	34	44	654
5:			
6:			
7:			17
8:			
9:			

So, the array becomes 10,21,11,123,24,44,654,17
Now, we'll sort according to the **ten's digit**:

0:
1: 10 11 17
2: 21 123
3: 34
4: 44
5: 654
6:
7:
8:
9:

Now, the array becomes : 10,11,17,21,123,34,44,654
Finally , we sort according to the **hundred's digit** (most significant digit):

0: 010 011 017 021 034 044
1: 123
2:
3:
4:
5:
6: 654
7:
8:
9:

The array becomes : 10,11,17,21,34,44,123,654 which is sorted. This is how our algorithm works. The time complexity O(d*(n+b)) d is number of digits is number of numbers is base (10 here, for alphabet base is 26)

Radix Sort Algorithm

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
```

sort the elements according to ith place digits using countingSort

```
countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

// Radix Sort Program

```
#include <stdio.h>

// Function to get the largest element from an array
int getMax(int array[], int n) {
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
  return max;
}

// Using counting sort to sort the elements in the basis of significant places
void countingSort(int array[], int size, int place) {
  int output[size + 1];
  int max = (array[0] / place) % 10;

  for (int i = 1; i < size; i++) {
    if (((array[i] / place) % 10) > max)
      max = array[i];
  }
  int count[max + 1];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  // Calculate count of elements
  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;

  // Calculate cummulative count
  for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];
```

```
  // Place the elements in sorted order
  for (int i = size - 1; i >= 0; i--) {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }

  for (int i = 0; i < size; i++)
    array[i] = output[i];
}

// Main function to implement radix sort
void radixsort(int array[], int size) {
  // Get maximum element
  int max = getMax(array, size);

  // Apply counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}

// Print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

// Driver code
int main() {
  int array[] = {121, 432, 564, 23, 1, 45, 788};
  int n = sizeof(array) / sizeof(array[0]);
  radixsort(array, n);
  printArray(array, n);
}
```

Complexity

Since radix sort is a non-comparative algorithm, it has advantages over comparative sorting algorithms.

For the radix sort that uses counting sort as an intermediate stable sort, the time complexity is $O(d(n+k))$.

Here, d is the number cycle and $O(n+k)$ is the time complexity of counting sort.

Thus, radix sort has linear time complexity which is better than $O(n \log n)$ of comparative sorting algorithms.

If we take very large digit numbers or the number of other bases like 32-bit and 64-bit numbers then it can perform in linear time however the intermediate sort takes large space.

This makes radix sort space inefficient. This is the reason why this sort is not used in software libraries.

Radix Sort Applications

Radix sort is implemented in

- DC3 algorithm (Kärkkäinen-Sanders-Burkhardt) while making a suffix array.
- places where there are numbers in large ranges.

QUICK SORT

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by **C. A. R. Hoare**.

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use **divide_and conquer** strategy. In quick sort, the partition of the list is performed based on the element called ***pivot***. Here pivot element is one of the elements in the list.

The list is divided into two partitions such that **"all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot"**.

Step by Step Process

In Quick sort algorithm, partitioning of the list is performed using following steps...

- **Step 1 -** Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
- **Step 2 -** Define two variables i and j. Set i and j to first and last elements of the list respectively.
- **Step 3 -** Increment i until list[i] > pivot then stop.
- **Step 4 -** Decrement j until list[j] < pivot then stop.
- **Step 5 -** If i < j then exchange list[i] and list[j].
- **Step 6 -** Repeat steps 3,4 & 5 until i > j.
- **Step 7 -** Exchange the pivot element with list[j] element.

Example

Consider the following unsorted list of elements...

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]

Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.
Repeat the same process on the left sublist and right sublist to the number 5.

In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.

Complexity of the Quick Sort Algorithm

To sort an unsorted list with **'n'** number of elements, we need to make **((n-1)+(n-2)+(n-3)+.....+1) = (n (n-1))/2** number of comparisions in the worst case. If the list is already sorted, then it requires **'n'** number of comparisions.

WorstCase:O(n^2)
BestCase:O(nlogn)
Average Case : O (n logn)

Implementaion of Quick Sort Algorithm using C Programming Language

```
#include<stdio.h>
#include<conio.h>

void quickSort(int [10],int,int);

void main(){
  int list[20],size,i;

  printf("Enter size of the list: ");
  scanf("%d",&size);

  printf("Enter %d integer values: ",size);
  for(i = 0; i < size; i++)
    scanf("%d",&list[i]);

  quickSort(list,0,size-1);
```

```
  printf("List after sorting is: ");
  for(i = 0; i < size; i++)
    printf(" %d",list[i]);

  getch();
}

void quickSort(int list[10],int first,int last){
    int pivot,i,j,temp;

     if(first < last){
         pivot = first;
         i = first;
         j = last;

         while(i < j){
             while(list[i]<= list[pivot] && i < last)
                 i++;
             while(list[j] > list[pivot])
                 j--;
             if(i <j){
                  temp = list[i];
                  list[i] = list[j];
                  list[j] = temp;
             }
```

```
        }

        temp = list[pivot];
        list[pivot] = list[j];
        list[j] = temp;
        quickSort(list,first,j-1);
        quickSort(list,j+1,last);
    }
}
```

MERGE SORT

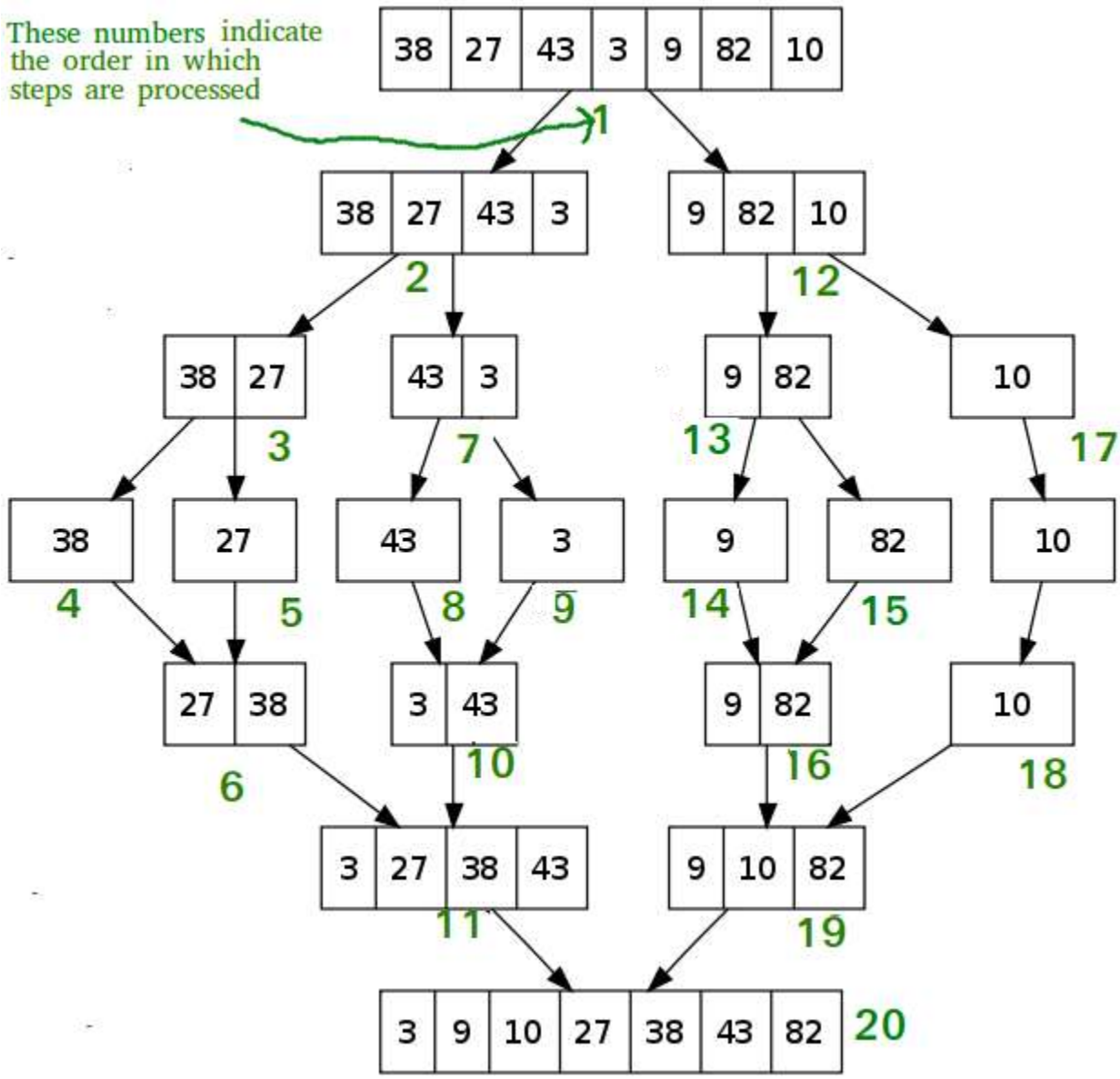
Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Implementation:

MergeSort(arr[], l, r)
If r > l

1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



/* C program for Merge Sort */

```
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];
```

```
    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
```

```
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

/* Driver code */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

HEAP SORT

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

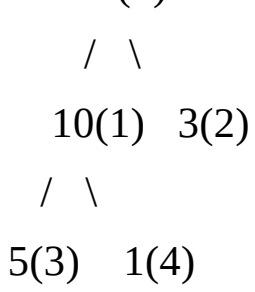
Why array based representation for Binary Heap?
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-

efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while size of heap is greater than 1.

How to build the heap?
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1



The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

10(1) 3(2)

/ \

5(3) 1(4)

Applying heapify procedure to index 0:

10(0)

/ \

5(1) 3(2)

/ \

4(3) 1(4)

The heapify procedure calls itself recursively to build heap in top down manner.

PROGRAM

```
#include <stdio.h>
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
```

```
        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d\n",arr[i]);
}
void swap(int *x,int *y)
{
    int temp;
    temp=*x;
    *x=*y;
```

```
    *y=temp;
}



int main() {
    // Write C code here
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    printf("Sorted array is \n");
    printArray(arr, n);

    return 0;
}
```

Comparison of sorting techniques:

Analysis Type	Selection Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort
Best Case	$O(n^2)$	O(n)	O(log n)	O(log n)	O(logn)
Average Case	$O(n^2)$	$O(n^2)$	O(log n)	O(log n)	O(log n)
Worst Case	$O(n^2)$	$O(n^2)$	O(log n)	$O(n^2)$	O(log n)

some sorting algorithms are non-comparison based algorithm. Some of them are Radix sort, Bucket sort, count sort. These are non-comparison based sort because here two elements are not compared while sorting. The techniques are slightly different. Now we will see the difference between them based on different type of analysis.

Analysis Type	Radix Sort (k is maximum digit)	Counting Sort (k is size of count array)	Bucket Sort (k is number of buckets)
Best Case	O(nk)	O(n + k)	O(n + k)
Average Case	O(nk)	O(n + k)	O(n + k)
Worst Case	O(nk)	O(n + k)	$O(n^2)$