UNIT - 3

Searching:

- Searching is the process of finding the location of given element in the linear array.
- The search is said to be successful if the given element is found, *i.e.*, the element does exist in the array; otherwise unsuccessful.
- There are two searching techniques :
  a. Linear search (sequential)
  b. Binary search
- The algorithm which we choose depends on organization of the array elements.
- If the elements are in random order, then we have to use linear search technique, and if the array elements are sorted, then it is preferable to use binary search.

External searching: When the records are stored in disk, tape, any secondary storage then that searching is known as 'External Searching'.
Internal Searching: When the records are to be searched or stored entirely within the computer memory then it is known as 'Internal Searching'.

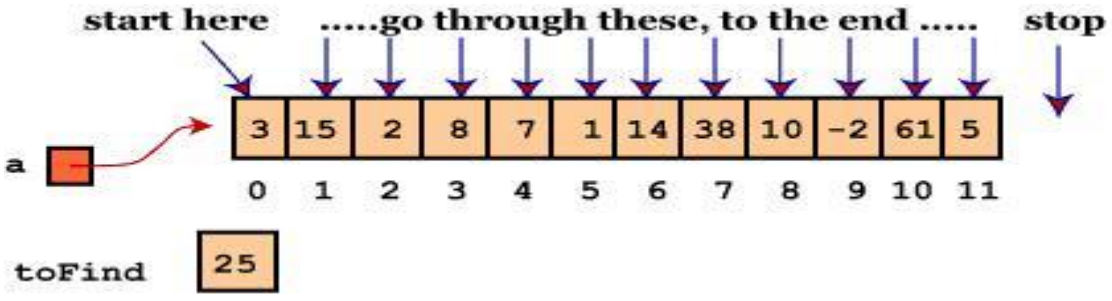
Sequential search:

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; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list.

How Linear Search works

Linear search in an array is usually programmed by stepping up an index variable until it reaches the last index. This normally requires two comparisons for each list item: one to check whether the index has reached the end of the array, and another one to check whether the item has the desired value.



Linear Search Algorithm

1. Repeat For J = 1 to N
2. If (ITEM == A[J]) Then
3. Print: ITEM found at location J
   4. Return [End of If]
   [End of For Loop]
   5. If (J > N) Then
6. Print: ITEM doesn't exist
   [End of If]
7. Exit

Program:

```
#include <stdio.h>
void main()
{
int a[10],i,n,m,c=0, x;
printf("Enter the size of an array: ");
scanf("%d",&n);
printf("Enter the elements of the array:\n ");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("Enter the number to be search:\n ");
scanf("%d",&m);
for(i=0;i<n;i++)
{
   if(a[i]==m)
   {
    x=i;
    c=1;
    break;
   }
}
if(c==0)
printf("The number is not in the list");
else
printf("The number is found at location %d", x);
}
```

Complexity of linear Search

Linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list. However, on average, assuming the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only n/2 elements. In best case the
array is already sorted i.e O(1)

Algorithm	Worst Case	Average Case	Best Case
Linear Search	O(n)	O(n)	O(1)

Index sequential search :

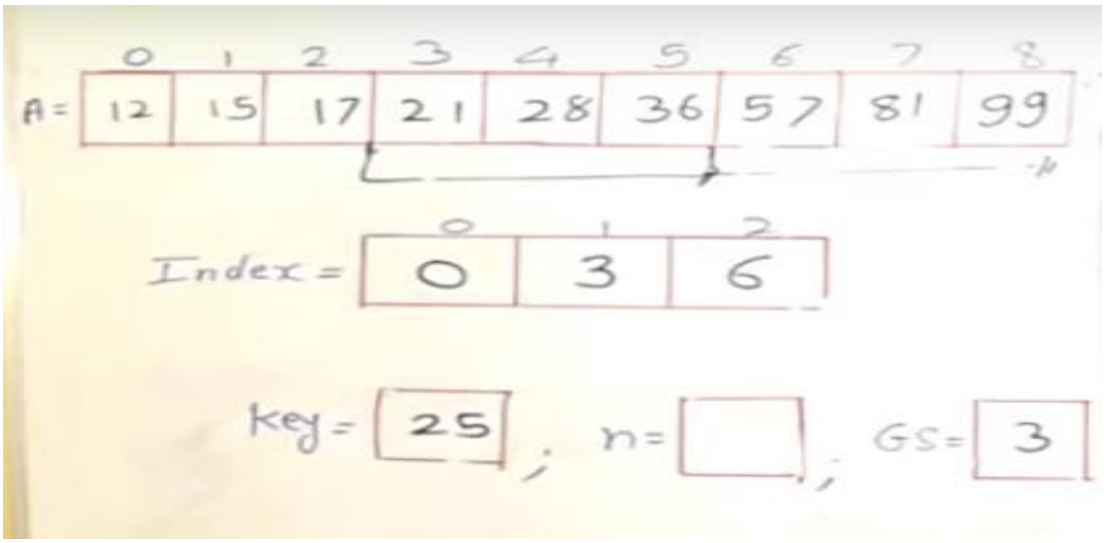
- In index sequential search, an index file is created, that contains some specific group or division of required record, once an index is obtained, then the partial searching of element is done which is located in a specified group.
- In indexed sequential search, a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- First the index is searched that guides the search in the array.
- Indexed sequential search does the indexing multiple times like creating the index of an index.
- When the user makes a request for specific records it will find that index group first where
  - that specific record is recorded.

Steps:

1. Read the search element from the user.
2. Divide the array into groups according to group size.
3. Create index array that contains starting index of groups
4. **If** group is present **and** first element of that group is **less than or equal to** key element
   go to next group
   Else apply linear search in previous group
5. Repeat step 4 for all groups

```
void indexedSequentialSearch(int arr[], int n, int k)
{
    int GN = 3; // GN is group number that is number of
                // elements in a group
    int elements[GN], indices[GN], i, set = 0;
    int j = 0, ind = 0, start, end;
    for (i = 0; i < n; i += 3) {

        // Storing element
        elements[ind] = arr[i];

        // Storing the index
        indices[ind] = i;
        ind++;
    }
    if (k < elements[0]) {
        printf("Not found");
        exit(0);
    }
    else {
        for (i = 1; i <= ind; i++)
            if (k <= elements[i]) {
                start = indices[i - 1];
                end = indices[i];
                set = 1;
                break;
            }
```

```
    }
    if (set == 0) {
        start = indices[GN - 1];
        end = GN;
    }
    for (i = start; i <= end; i++) {
        if (k == arr[i]) {
            j = 1;
            break;
        }
    }
    if (j == 1)
        printf("Found at index %d", i);
    else
        printf("Not found");
}

// Driver code
void main()
{

    int arr[] = { 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Element to search
    int k = 8;
    indexedSequentialSearch(arr, n, k);
}
```

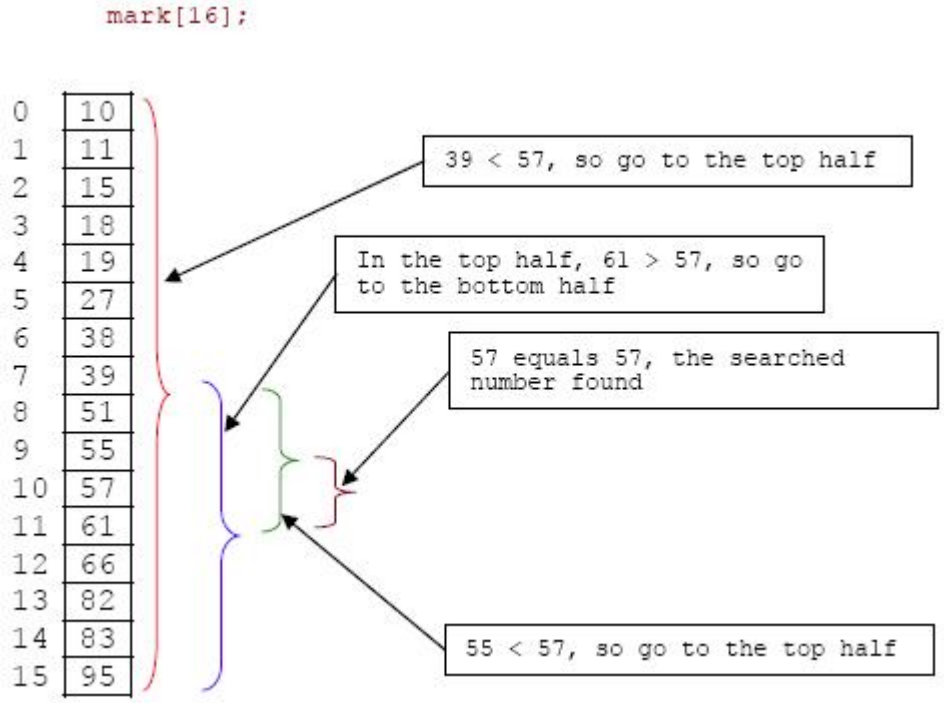


Binary search:

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

How Binary Search Works

Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.



Binary Search Algorithm

1. Set BEG = 1 and END = N
2. Set MID = (BEG + END) / 2
3. Repeat step 4 to 8 While (BEG <= END) and (A[MID] ≠ ITEM)
4. If (ITEM < A[MID]) Then
5. Set END = MID – 1
6. Else
7. Set BEG = MID + 1
   [End of If]
8. Set MID = (BEG + END) / 2

9. If (A[MID] == ITEM) Then
10. Print: ITEM exists at location MID
11. Else
12. Print: ITEM doesn't exist
[End of If]
13. Exit

Complexity of Binary Search

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time.

	Worst Case	Average Case	Best Case
Binary Search	O(nlogn)	O(nlogn)	O(1)

Program:

```
#include <stdio.h>
void main()
{
int ar[10],val,mid,low,high,size,i;
printf("\nenter the no.s of elements  in array\n");
scanf("%d",&size);
for(i=0;i<size;i++)
{
printf("input the element no %d\n",i+1);
scanf("%d",&ar[i]);
}
printf("the arry inputed is \n");
for(i=0;i<size;i++)
{
 printf("%d\t",ar[i]);
}
low=0; high=size-1;
printf("\ninput the no. for search \n");
scanf("%d",&val);
while(val!=ar[mid]&&high>=low)
{
mid=(low+high)/2;
if(ar[mid]==val)
{
printf("value found at %d position",mid+1);
}
if(val>ar[mid])
{
low=mid+1;
}
else
{
high=mid-1;
}}}
```

Complexity of Binary Search

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time.

S. No.	Sequential (linear) search	Binary search
1.	No elementary condition *i.e.*, array can be sorted or unsorted.	Elementary condition *i.e.*, array should be sorted.
2.	It takes long time to search an element.	It takes less time to search an element.
3.	Complexity is O(n).	Complexity is O($\log_2 n$).
4.	It searches data linearly.	It is based on divide and conquer method.

Hashing:

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.

A **Hash Function** is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function **maps** a significant number or string to a small integer that can be used as the **index** in the hash table.

The pair is of the form **(key, value)**, where for a given key, one can find a value using some kind of a "function" that maps keys to values. The key for a given object can be calculated using a function called a hash function. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i].

Types of Hash functions

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. **Division Method.**
2. **Mid Square Method.**
3. **Folding Method.**
4. **Multiplication Method.**

Let's begin discussing these methods in detail.

1. Division Method:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

h(K) = k mod M

Here,

***k** is the key value, and*

***M** is the size of the hash table.*

It is best suited that **M** is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

k = 12345

M = 95

h(12345) = 12345 mod 95
= 90

Pros:

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:

1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

2. Mid Square Method:

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k^2
2. Extract the middle **r** digits as the hash value.

Formula:

h(K) = h(k x k)

Here,

***k** is the key value.*

The value of **r** can be decided based on the size of the table.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60
k x k = 60 x 60
= 3600
h(60) = 60

The hash value obtained is 60

Pros:

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

1. The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

3. Digit Folding Method:

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:

k = k1, k2, k3, k4, ….., kn
s = k1+ k2 + k3 + k4 +….+ kn
h(K)= s

Here,
s** is obtained by adding the parts of the key **k
Example:
k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3
= 12 + 34 + 5
= 51
h(K) = 51
Note:
The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

4. Multiplication Method

This method involves the following steps:

1. Choose a constant value A such that 0 < A < 1.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:
h(K) = floor (M (kA mod 1))
Here,
***M** is the size of the hash table.*
***k** is the key value.*
***A** is a constant value.*
Example:
k = 12345
A = 0.357840
M = 100

*h(12345) = floor[100 (12345*0.357840 mod 1)]*
= floor[100 (4417.5348 mod 1)]
= floor[100 (0.5348)]
= floor[53.48]
= 53

Pros:
The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

Cons:
The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast

.

Collision in Hashing:

The hash function is used to find the index of the array. The hash value is used to create an index for the key in the hash table. The hash function may return the same hash value for two or more keys. When two or more keys have the same hash value, a collision happens. To handle this collision, we use collision resolution techniques.

Collision Resolution Techniques

There are two types of collision resolution techniques.

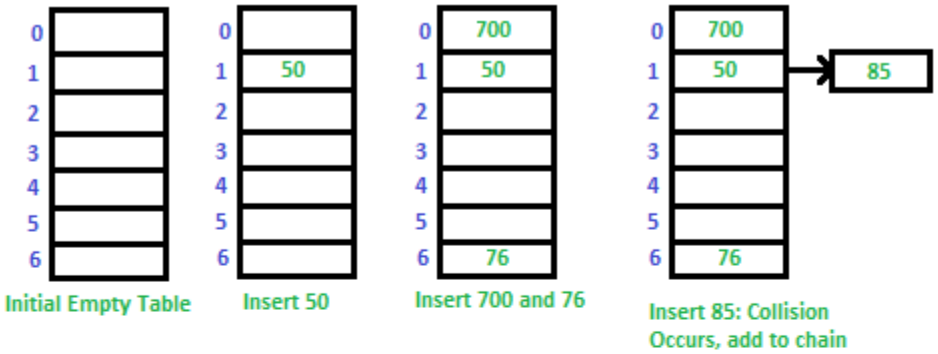
- Separate chaining (open hashing)
- Open addressing (closed hashing)

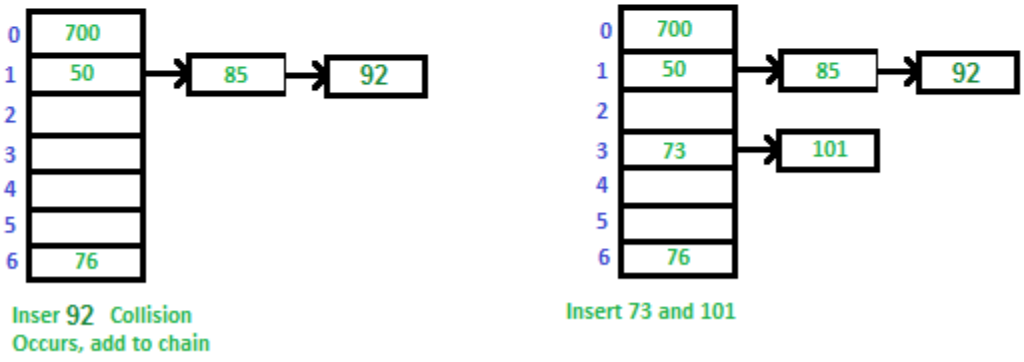
Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

*The **linked list** data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*

Example: Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101





Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.

- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Open Addressing:

In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:

- ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
- ***Search(k):*** *Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.*
- ***Delete(k)**: **Delete operation is interesting**. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".*
  The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Different ways of Open Addressing:

1. **Linear Probing:**

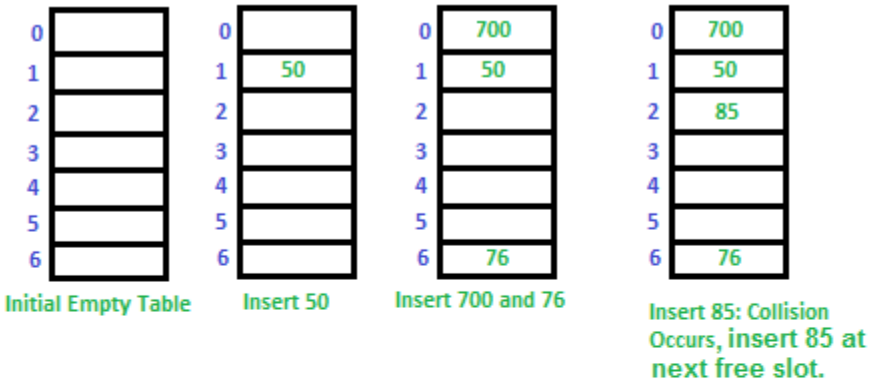
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

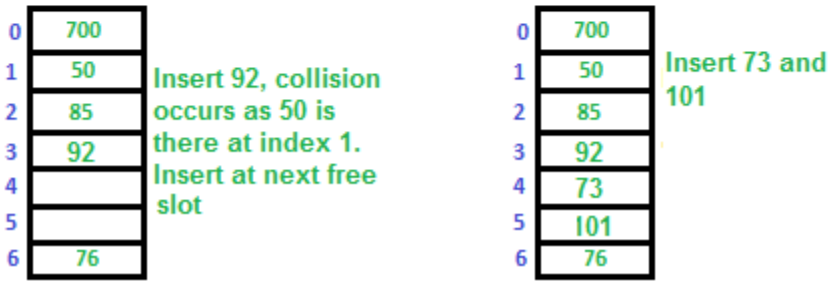
The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.

For example, The typical gap between two probes is 1 as seen in the example below:
*Let **hash(x)** be the slot index computed using a hash function and **S** be the table size*
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
……..
……..

Example:*Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101,*

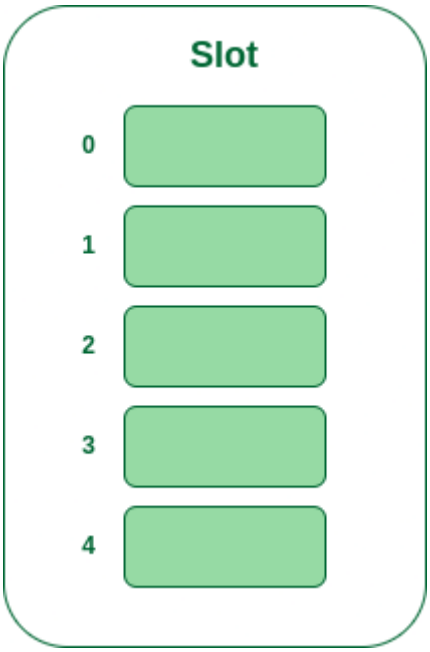
which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.We can define the hash function as per our choice if we want to create a hash table,although it is fixed internally with a pre-defined formula.





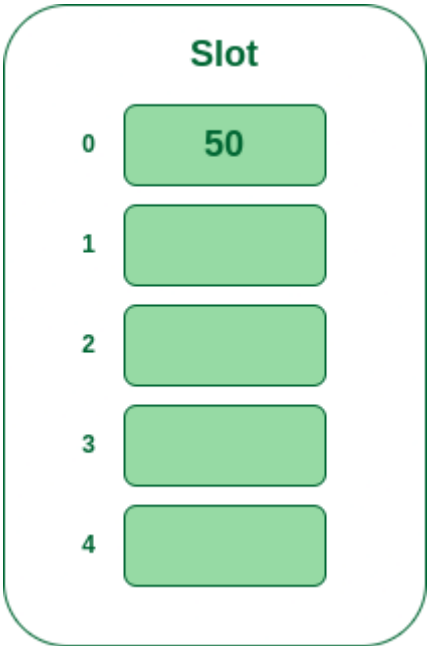
Example: Let us consider a simple hash function as "key mod 5" and a sequence of keys that are to be inserted are 50, 70, 76, 93.

- **Step1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



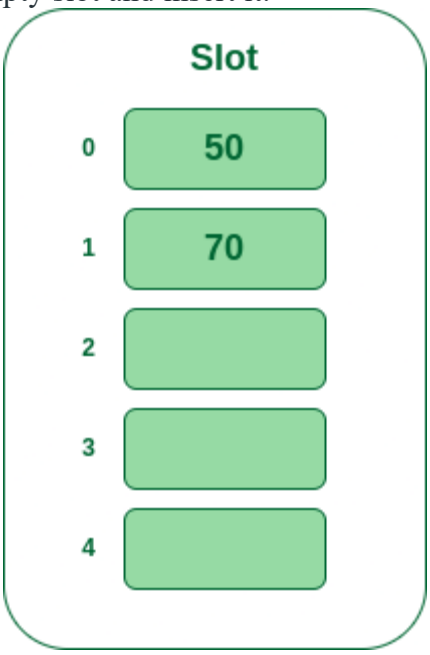
Hash table

- **Step 2:** Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.



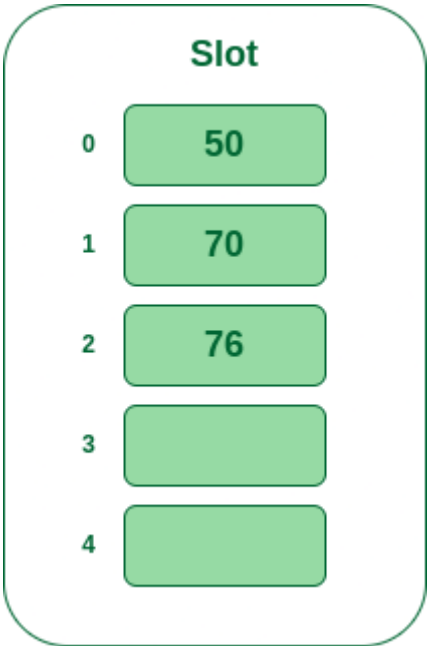
Insert 50 into hash table

- **Step 3:** The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.



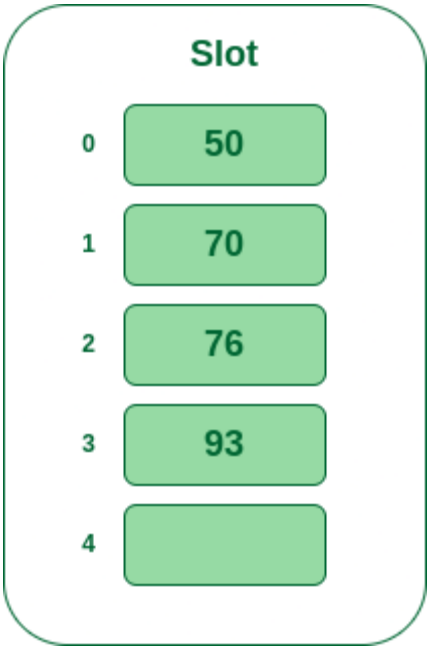
Insert 70 into hash table

- **Step 4:** The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.



Insert 76 into hash table

- **Step 5:** The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.



Insert 93 into hash table

2. Quadratic Probing

If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value. Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed above. This method is also known as the **mid-square** method. In this method, we look for the $\mathbf{i^2}$**'th** slot in the $\mathbf{i^{th}}$ iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.
let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*

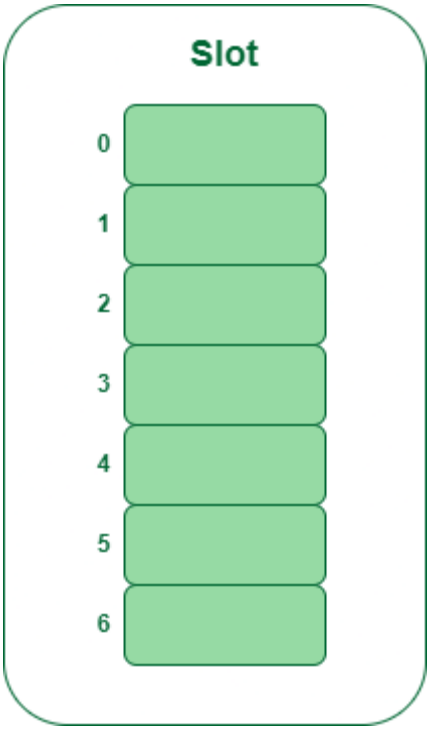
*If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S*

……

……

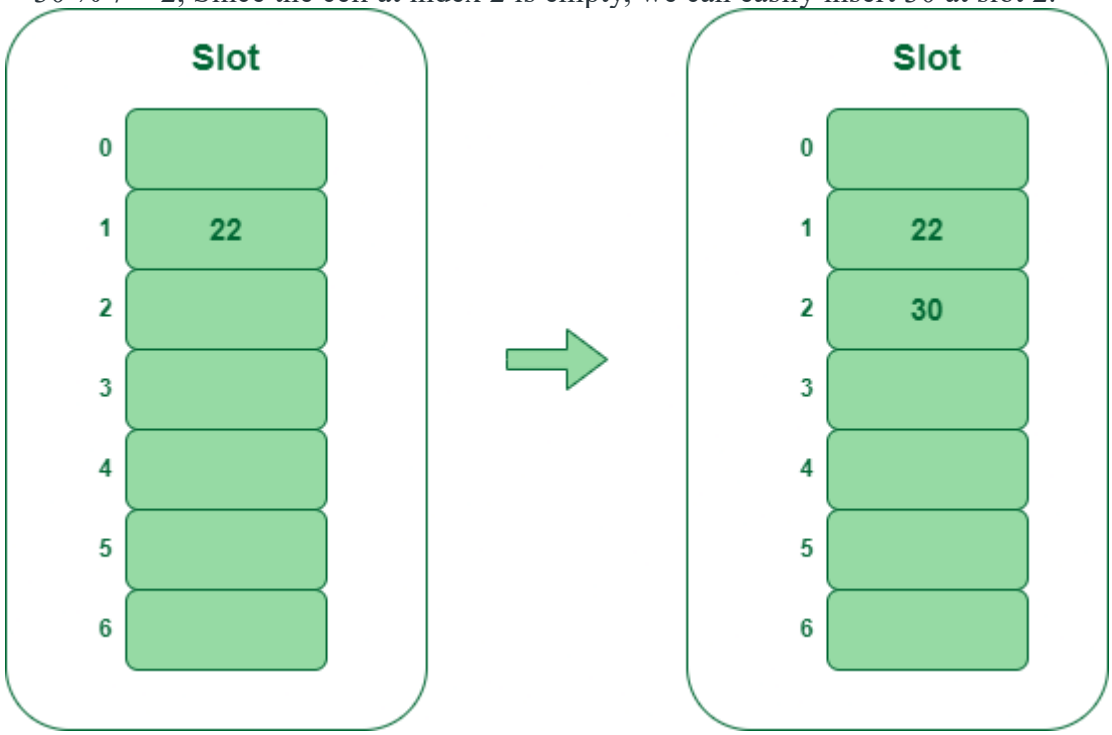
Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, and 50.

- **Step 1:** Create a table of size 7.



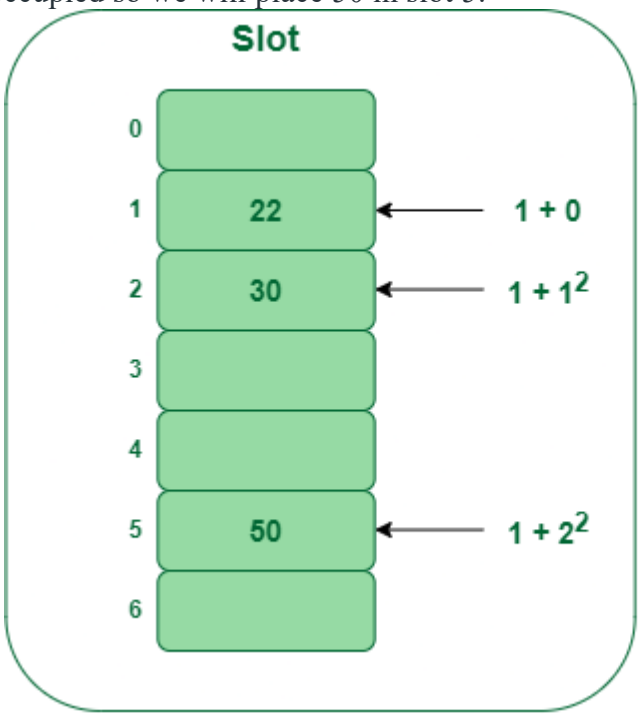
Hash table

- **Step 2** – Insert 22 and 30
- Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
- Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert keys 22 and 30 in the hash table

- **Step 3:** Inserting 50
- Hash(50) = 50 % 7 = 1
- In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
- Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
- Now, cell 5 is not occupied so we will place 50 in slot 5.



Insert key 50 in the hash table

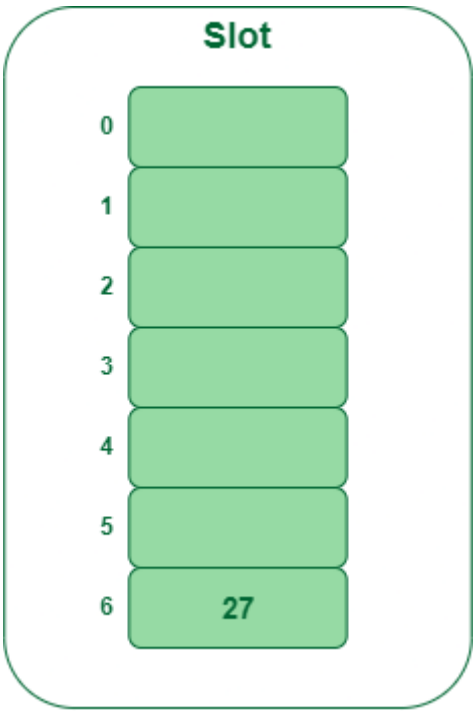
3. Double Hashing

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **ith** rotation.
let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*
*If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S*
*If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S*
……
……

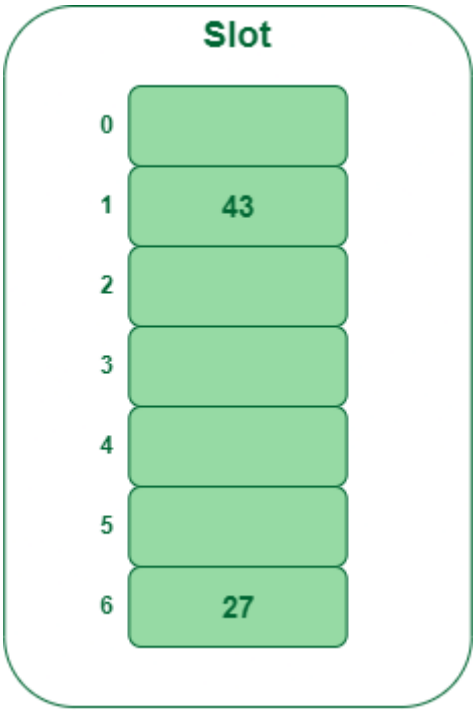
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1 (k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
- 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Insert key 27 in the hash table

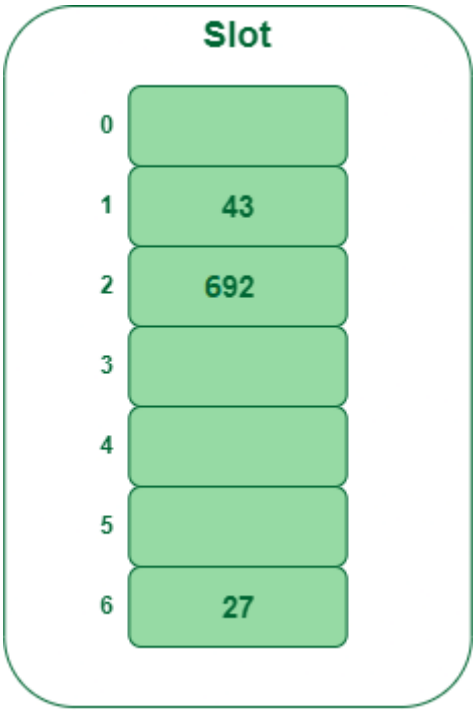
- **Step 2:** Insert 43
- 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



Insert key 43 in the hash table

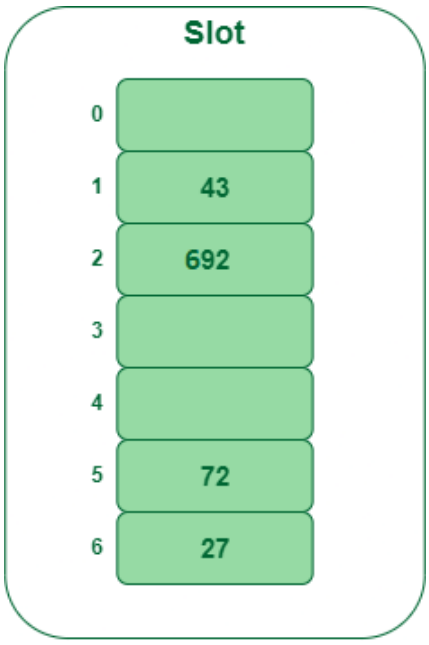
- **Step 3:** Insert 692
- 692 % 7 = 6, but location 6 is already being occupied and this is a collision
- So we need to resolve this collision using double hashing.
  h_{new} = *[h1(692) + i * (h2(692)] % 7*
  *= [6 + 1 * (1 + 692 % 5)] % 7*

= 9% 7
= 2
Now, as 2 is an empty slot, so we can insert 692 into 2nd slot.



Insert key 692 in the hash table

- **Step 4:** Insert 72
- 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
- So we need to resolve this collision using double hashing.
  h_{new} *= [h1(72) + i * (h2(72)] % 7*
  *= [2 + 1 * (1 + 72 % 5)] % 7*
  = 5 % 7
  = 5,
  Now, as 5 is an empty slot, so we can insert 72 into 5th slot.



Insert key 72 in the hash table

Difference between Separate Chaining and open addressing

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn’t map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

INTRODUCTION TO SORTING

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:
Roll No. Name
Age
Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Sorting Efficiency

Sorting techniques mainly depends on two parameters.
First parameter is the execution time of program, which means time taken for execution of program.
Second is the space, which means space taken by the program.

TYPES OF SORTING

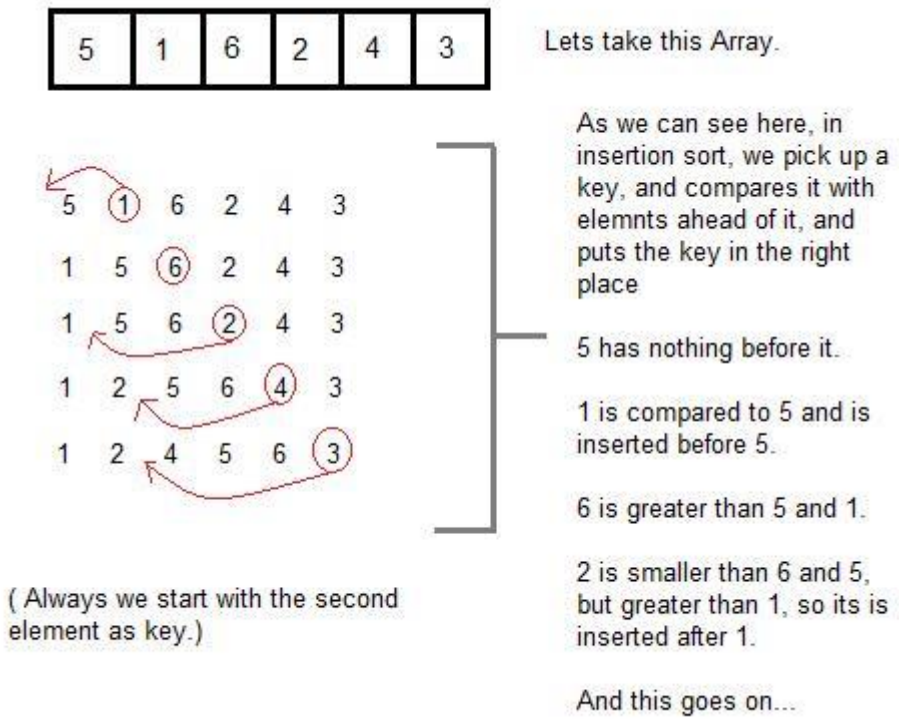
- An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.
- External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub files are combined into a single larger file.
- We can say a sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

Insertion sort

It is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. This algorithm is 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
- 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

How Insertion Sort Works



Insertion Sort Algorithm

This algorithm sorts the array A with N elements.

1. Repeat step 3 to 5 for i=1 to n
2. Set key=A[i] And j=i-1
3. Repeat while (J>=0 & key<A[j]
   Set A[j+1]=A[j]
   j=j-1

5. Set A[j+1]=key
   6. Return

Program:

```
#include <stdio.h>
void Insert(int A[], int n)
{
int i, j, key;
for(i=1; i<n; i++)
{
key = A[i];
j = i-1;
while(j>=0 && key < A[j])
{
A[j+1] = A[j];
 j--;
}
A[j+1] = key;
}
printf("sorted array");
for(i=0;i<n;i++)
printf(" %d", A[i]);
}

int main()
{
    int A[10] ,I , n;
    printf(" enter size of array");
    scanf("%d",&n);
    printf(" enter elements of array");
    for(i=0;i<n;i++)
    scanf("%d",&A[i]);
    Insert(A,n);
   return 0;
}
```

Complexity of Insertion Sort

The number f(n) of comparisons in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K-1 of comparisons. Hence

$F(n)= 1+2+3+\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots+(n-1)=n(n-1)/2= O(n^2)$

Furthermore, One can show that, on the average, there will be approximately (K-1)/2 comparisons in the inner loop. Accordingly, for the average case.

$F(n)=O(n^2)$

Thus the insertion sort algorithm is a very slow algorithm when n is very large.

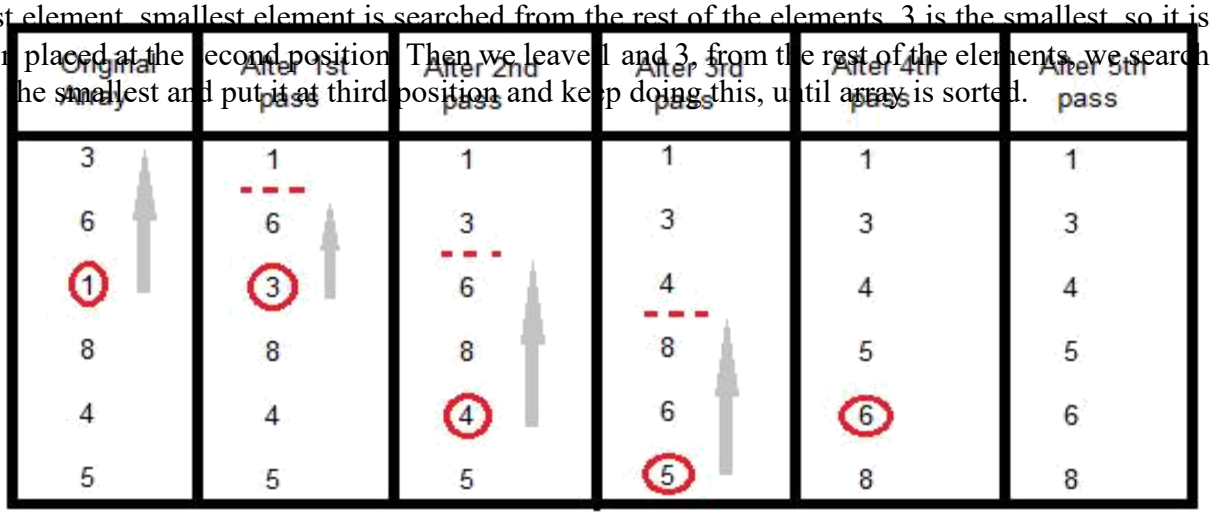
Algorithm	Worst Case	Average Case	Best Case
Insertion Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/4 = O(n^2)$	O(n)

Selection Sort

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted

How Selection Sort works

In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.



Selection Sort Algorithm

1. Repeat For J = 0 to N-1
2. Set MIN = J
3. Repeat For K = J+1 to N
4. If (A[K] < A[MIN]) Then
5. Set MIN = K
   [End of If]
   [End of Step 3 For Loop]
   6. Interchange A[J] and A[MIN] [End of Step 1 For Loop]
7. Exit

Program:

```
include <stdio.h>
void selectionSort(int a[], int size)
{
int i, j, min, temp;
for(i=0; i < size-1; i++ )
{
min = i; //setting min as i
for(j=i+1; j < size; j++)
{
if(a[j] < a[min])  //if element at j is less than element at min position
{
min = j;  //then set min as j
}
}
temp = a[i];
a[i] = a[min];
a[min] = temp;
}
printf("sorted array");
for(i=0;i<6;i++)
printf(" %d", a[i]);
}


int main()
{
    int A[10], i, n;
    printf(" enter size of array");
    scanf("%d", &n);
```

```
    printf(" enter elements of array");
    for(i=0;i<n;i++)
    scanf("%d",&A[i]);
     selectionSort(A, n);
    return 0;
}
```

Complexity of Selection Sort Algorithm

The number of comparison in the selection sort algorithm is independent of the original order of the element. That is there are n-1 comparison during PASS 1 to find the smallest element, there are n-2 comparisons during PASS 2 to find the second smallest element, and so on. Accordingly

$F(n)=(n-1)+(n-2)+\ldots\ldots\ldots\ldots\ldots\ldots\ldots+2+1=n(n-1)/2 = O(n^2)$

Algorithm	Worst Case	Average Case	Best Case
Selection Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$O(n^2)$

Bubble Sort

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.
It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

How Bubble Sort Works

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

First Pass:
(5 1 4 2 8) $\rightarrow$ (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 5 4 2 8) $\rightarrow$ (1 4 5 2 8), Swap since 5 > 4 (
1 4 5 2 8) $\rightarrow$ (1 4 2 5 8), Swap since 5 > 2
(1 4 2 5 8) $\rightarrow$ (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm

does not swap them.

Second Pass:
(1 4 2 5 8) → (1 4 2 5 8)
(1 4 2 5 8) → (1 2 4 5 8), Swap since 4 >
2 (1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:
(1 2 4 5 8) → (1 2 4 5 8) (1 2 4 5 8) → (1 2 4 5 8) (1 2 4 5 8) → (1 2 4 5 8) (1 2 4 5 8) → (1 2 4 5 8)

Bubble Sort Algorithm

1. Repeat Step 2 and 3 for i=0 to n
2. Repeat for j=0 to n-1-i
   If (A[j] > A[j+1]) Then Interchange A[j] and A[j+1]
   [End of If]
   [end of step 2 loop]
   [end of step 1 loop]
4. Exit

Program:

```
#include <stdio.h>
void Bubblesort(int A[], Int n)
{
int i, j, temp;
for(i=0; i<n; i++)
{
for(j=0; j<n-i-1; j++)
{
if( a[j] > a[j+1])
{
temp = a[j];
a[j] = a[j+1];
a[j+1] = temp;
}}}
printf("sorted array");
for(i=0;i<n;i++)
printf(" %d", a[i]);
}
```

```
int main()
{
    int A[10],I,n;
    printf(" enter size of array");
    scanf("%d",&n);
    printf(" enter elements of array");
    for(i=0;i<n;i++)
    scanf("%d",&A[i]);
    Bubblesort(A, 6);
    return 0;
}
```

Complexity of Bubble Sort Algorithm

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

$F(n)=(n-1)+(n-2)+\ldots\ldots\ldots\ldots\ldots\ldots\ldots+2+1=n(n-1)/2 = O(n^2)$

Algorithm	Worst Case	Average Case	Best Case
Bubble Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	O(n)

Quick Sort

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called partition-exchange sort). This algorithm divides the list into three main parts
Elements less than the Pivot
element Pivot element
Elements greater than the pivot element

How Quick Sort Works

In the list of elements, mentioned in below example, we have taken 25 as pivot. So after the first pass, the list will be changed like this.
6 8 17 14 **25** 63 37 52
Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

Quick Sort Algorithm

```
QUICKSORT (A, p, r)
1 if p < r
2 then q ← PARTITION (A, p, r)
3 QUICKSORT (A, p, q - 1)
4 QUICKSORT (A, q + 1, r)
```

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A[p r] in place.

```
PARTITION (A, p, r)
1 pivot← A[r]
2 i ← p - 1
3 for j ← p to r - 1
      do if A[j] ≤ pivot
      then i ← i + 1
      exchange A[i] ↔ A[j]
      [end IF]
  [end for loop]
7 exchange A[i + 1] ↔ A[r]
8 return i + 1
```

Program:

```
#include<stdio.h>

void Quicksort(int A[], int p, int r){

        if(p<r){
                int q = Partition(A, p, r);
                Quicksort(A, p, q-1);
                Quicksort(A, q+1, r);
        }}

int Partition(int A[], int p, int r)
{
        int j, temp;
        int pivot = A[r];
        int i = p -1;

        for(j = p; j<r; j++){
```

```
            if(A[j]<= pivot)
            {
                  i++;
                  temp = A[i];
                  A[i] = A[j];
                  A[j] = temp;
            }
      }
      temp = A[r];
      A[r] = A[i +1];
      A[i+1] = temp;
      return i+1;  }


int main()
{
int i,n,a[20],p=0;
printf("Enter the size of array");
scanf("%d",&n);
printf("Enter the element of array");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
Quicksort(a,p,n-1);
printf("After sorting the array:");
for(i=0;i<n;i++)
printf("  %d",a[i]);
return 0;
}
```

Complexity of Quick Sort Algorithm

The Worst Case occurs when the list is sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have n-1 elements. Accordingly the second element require n-1 comparisons to recognize that it remains in the second position and so on.

$F(n)= n+(n-1)+(n-2)+\ldots\ldots\ldots\ldots\ldots\ldots\ldots+2+1=n(n+1)/2 = O(n^2)$

Algorithm	Worst Case	Average Case	Best Case
Quick Sort	$n(n+1)/2 = O(n^2)$	O(n logn)	O(n logn)

Merge Sort

Merge Sort follows the rule of Divide and Conquer. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into N sub lists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sub lists, to produce new sorted sub lists, and at lasts one sorted list is produced.
Merge Sort is quite fast, and has a time complexity of O(n log n). It is also a stable sort, which means the equal elements are ordered in the same order in the sorted list.

How Merge Sort Works

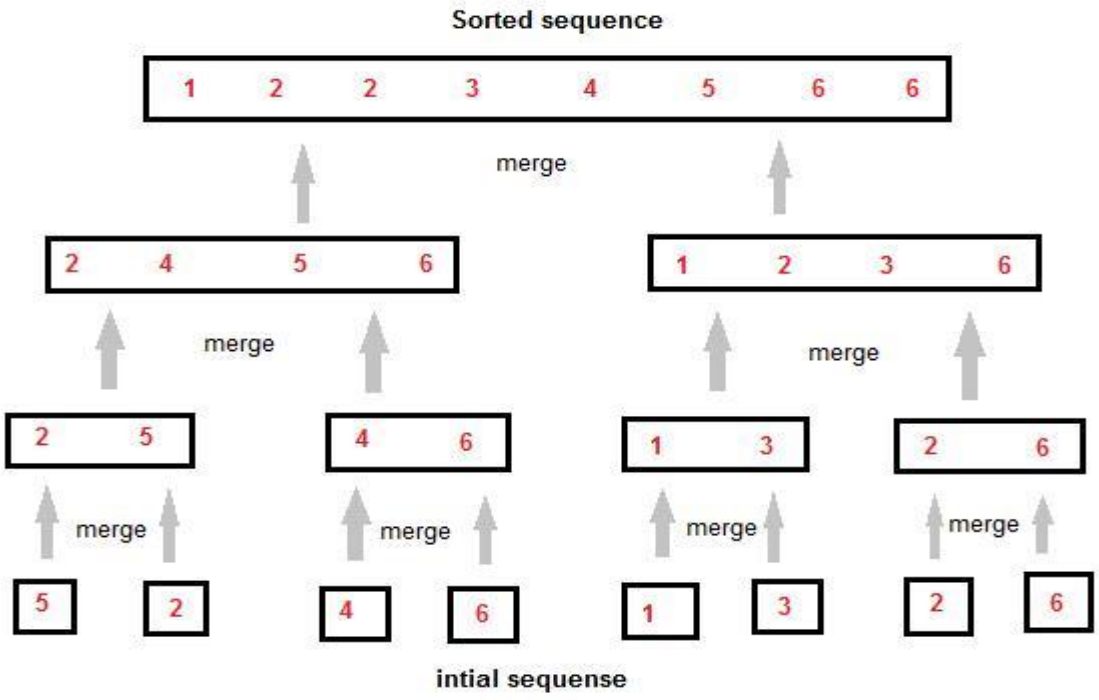
Suppose the array A contains 8 elements, each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows.

PASS 1. Merge each pair of elements to obtain the list of sorted pairs.

PASS 2. Merge each pair of pairs to obtain the list of sorted quadruplets.

PASS 3. Merge each pair of sorted quadruplets to obtain the two sorted subarrays.

PASS 4. Merge the two sorted subarrays to obtain the single sorted array.



Merge Sort Algorithm

```
MERGE_SORT (a, p, r)
1. if p < r
2. then q ← (p + r)/2⌋
3.  MERGE-SORT (A, p, q)
    MERGE-SORT (A, q + 1, r)
.   MERGE (A, p, q, r)
4 .return

MERGE (A, p, q, r)
 1.   n1 = q – p + 1
 2.   n2 = r – q
 3.  Create arrays L [n1] and R [n2]
 4. for i = 0 to n1-1
    do L[i] = A [p + i ]
    endfor
5. for j = 0 to n2-1
    do R[j] = A[q + j+1]
    endfor
6.  set  i = 0, j = 0, k=p
7. while(i<n1 and  j<n2)
   do if L[i] ≤ R[j]
         then A[k] = L[i]
           i = i + 1
        else
           A[k] = R[j]
           j = j + 1
        endif
      k=k+1
   end while
     8.   while (i< n1)
          a[k] = L[i]
          i++, k++
   end while

     9.    while (j< n2)
          a[k] = R[j]
          j++, k++
   end while

10. exit
```

Program:

```
#include <stdio.h>

void merge(int arr[], int start, int mid, int end)
```

```
{
  int len1 = mid - start + 1;
  int len2 = end - mid;
  int leftArr[len1], rightArr[len2];
  for (int i = 0; i < len1; i++)
    leftArr[i] = arr[start + i];
  for (int j = 0; j < len2; j++)
    rightArr[j] = arr[mid + 1 + j];
  int i, j, k;
  i = 0;
  j = 0;
  k = start;
  while (i < len1 && j < len2) {
    if (leftArr[i] <= rightArr[j]) {
      arr[k] = leftArr[i];
      i++;
    } else {
      arr[k] = rightArr[j];
      j++;
    }
    k++;
  }

  while (i < len1) {
    arr[k] = leftArr[i];
    i++;
    k++;
  }

  while (j < len2) {
    arr[k] = rightArr[j];
    j++;
    k++;
  }
}

void mergeSort(int arr[], int start, int end)
{
  if (start < end)
{
    int mid = start + (end - start) / 2;
    mergeSort(arr, start, mid);
    mergeSort(arr, mid + 1, end);
    merge(arr, start, mid, end);
  }
}
```

```
void display(int arr[], int size)
{
  for (int i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

int main()
{
    int i, n, a[20];
printf("Enter the size of array");
scanf("%d",&n);
printf("Enter the element of array");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
  printf("Original array\n");
  display(a, n);
    mergeSort(a, 0, n - 1);
  printf("Sorted array\n");
  display(a, n);
}
```

Complexity of Merge Sort Algorithm

Let f(n) denote the number of comparisons needed to sort an n-element array A using merge-sort algorithm. The algorithm requires at most logn passes. Each pass merges a total of n elements and each pass require at most n comparisons.

Thus the time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

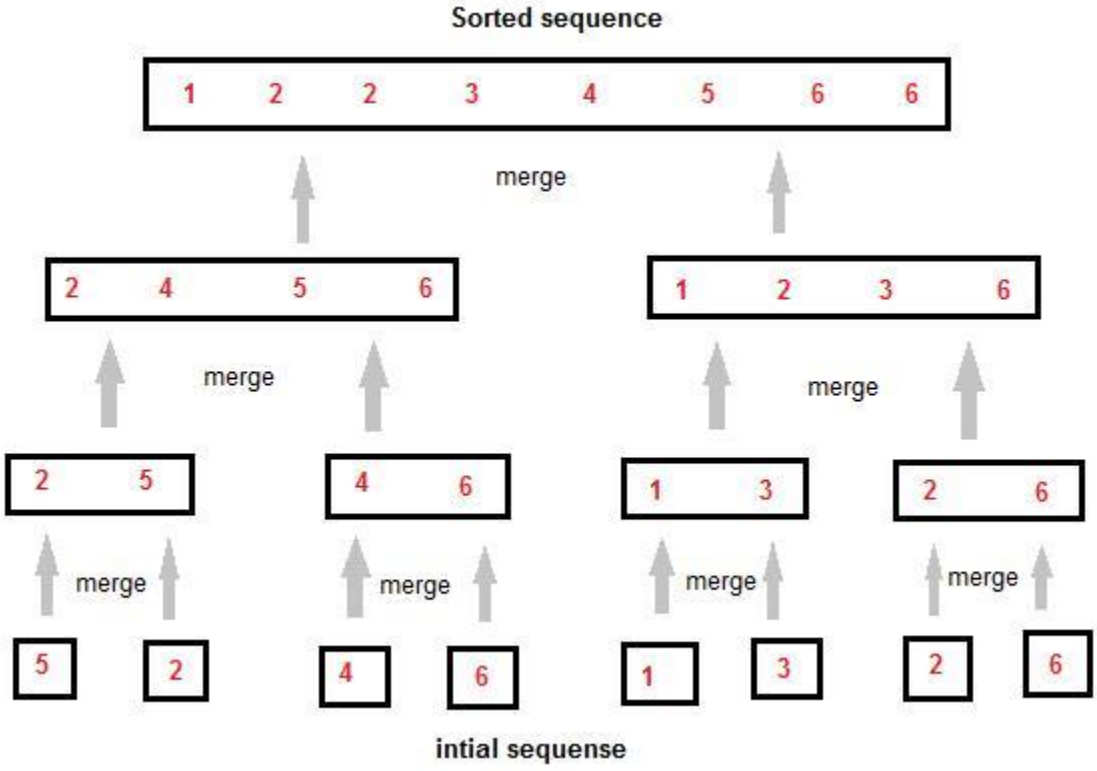
Algorithm	Worst Case	Average Case	Best Case
Merge Sort	O(n logn)	O(n logn)	O(n logn)

PASS 1. Merge each pair of elements to obtain the list of sorted pairs.

PASS 2. Merge each pair of pairs to obtain the list of sorted quadruplets.

PASS 3. Merge each pair of sorted quadruplets to obtain the two sorted subarrays.

PASS 4. Merge the two sorted subarrays to obtain the single sorted array.



Merge Sort Algorithm

/* Sorting using Merge Sort Algorithm
a[] is the array, p is starting index, that is 0, and r is the last index
of array. */

Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

```
void mergesort(int a[], int p, int r)
{
int q; if(p < r)
{
q = floor( (p+r) / 2);
mergesort(a, p, q);
```

```
mergesort(a, q+1, r);
 merge(a, p, q, r);
}
}

void merge (int a[], int p, int q, int r)
{
int b[5]; //same size of a[] int i, j, k;
k = 0; i = p;
j = q+1;
while(i <= q && j <= r)
{
if(a[i] < a[j])
{
b[k++] = a[i++];    // same as b[k]=a[i]; k++; i++;
}
else
{
b[k++] = a[j++];
}
}

while(i <= q)
{
b[k++] = a[i++];
}

while(j <= r)
{
b[k++] = a[j++];
}

for(i=r; i >= p; i--)
{
a[i] = b[--k];           // copying back the sorted list to a[]
}
}
```

Complexity of Merge Sort Algorithm

Let f(n) denote the number of comparisons needed to sort an n-element array A using merge-sort algorithm. The algorithm requires at most logn passes. Each pass merges a total of n elements and each pass require at most n comparisons. Thus for both the worst and average case

$F(n) \leq n \log n$

Thus the time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

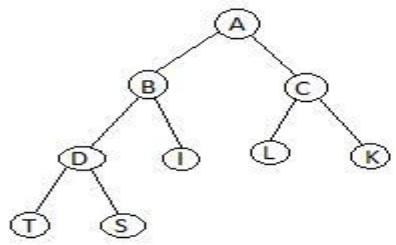
Algorithm	Worst Case	Average Case	Best Case
Merge Sort	O(n logn)	O(n logn)	O(n logn)

5.3.6 Heap Sort

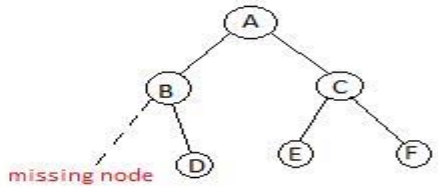
Heap Sort is one of the best sorting methods being in -place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts
Creating a Heap of the unsorted list.
Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.
What is a Heap?
Heap is a special tree-based data structure that satisfies the following special heap properties **Shape Property**: Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.
Heap Property: All nodes are either greater than or equal to or less than or equal to each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smaller than their child nodes, heap is called Min-Heap.

How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.



Complete Binary Tree



In-Complete Binary Tree

Heap Sort Algorithm

- HEAPSORT(A)

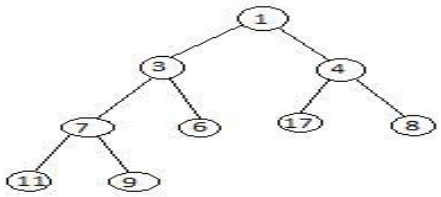
1. BUILD-MAX-HEAP(A)
2. for i ← length[A] downto 2
3. do exchange A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. MAX-HEAPIFY(A, 1)

- BUILD-MAX-HEAP(A)

1. heap-size[A] ← length[A]
2. for i ← length[A]/2 downto 1
3. do MAX-HEAPIFY(A, i)

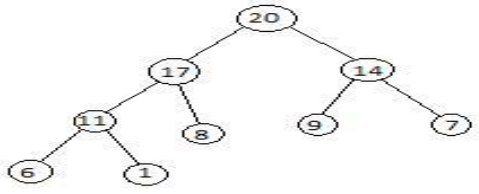
- MAX-HEAPIFY(A, i)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. if l ≤ heap-size[A] and A[l] > A[i]
4. then largest ←l
5. else largest ←i
6. if r ≤ heap-size[A] and A[r] > A[largest]
7. then largest ←r
8. if largest = i
9. then exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest)



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

//CODE

In the below algorithm, initially heapsort() function is called, which calls buildmaxheap() to build heap, which inturn uses maxheap() to build the heap.

```
void heapsort(int[], int);
void buildmaxheap(int [], int); void
maxheap(int [], int, int);

void main()
{
int a[10], i, size;
printf("Enter size of list"); // less than 10, because max size of array is 10
scanf(“%d”,&size);
printf( "Enter" elements"); for( i=0; i <
size; i++)
{
Scanf(“%d”,&a[i]);
}
heapsort(a, size); getch();
}

void heapsort (int a[], int length)
{
buildmaxheap(a, length); int heapsize, i,
temp; heapsize = length - 1;
for( i=heapsize; i >= 0; i--)
{
temp = a[0];
a[0] = a[heapsize]; a[heapsize] =
temp; heapsize--;
maxheap(a, 0, heapsize);
}
for( i=0; i < length; i++)
{
Printf("\t%d" ,a[i]);
}
}

void buildmaxheap (int a[], int length)
{
int i, heapsize; heapsize = length - 1;
```

```
for( i=(length/2); i >= 0; i--)
{
maxheap(a, i, heapsize);
}
}

void maxheap(int a[], int i, int heapsize)
{
int l, r, largest, temp; l = 2*i;
r = 2*i + 1;
if(l <= heapsize && a[l] > a[i])
{
largest = l;
}
else
{
largest = i;
}
if( r <= heapsize && a[r] > a[largest])
{
largest = r;
}
if(largest != i)
{
temp = a[i]; a[i] =
a[largest];
a[largest] = temp; maxheap(a, largest,
heapsize);
}
}
```

Complexity of Heap Sort Algorithm

The heap sort algorithm is applied to an array A with n elements. The algorithm has two phases, and we analyze the complexity of each phae separately.

Phase 1. Suppose H is a heap. The number of comparisons to find the appropriate place of a new element item in H cannot exceed the depth of H. Since H is complete tree, its depth is bounded by $\log_2 m$ where m is the number of elements in H. Accordingly, the total number g(n) of comparisons to insert the n elements of A into H is bounded as

$g(n) \leq n \log_2 n$

Phase 2. If H is a complete tree with m elements, the left and right subtrees of H are heaps and L is the root of H Reheaping uses 4 comparisons to move the node L one step down the tree H. Since the depth cannot exceeds $\log_2 m$, it uses $4\log_2 m$ comparisons to find the appropriate place of L in the tree H.

$h(n) \leq 4n\log_2 n$

Thus each phase requires time proportional to $n\log_2 n$, the running time to sort n elements array A would be $n\log_2 n$

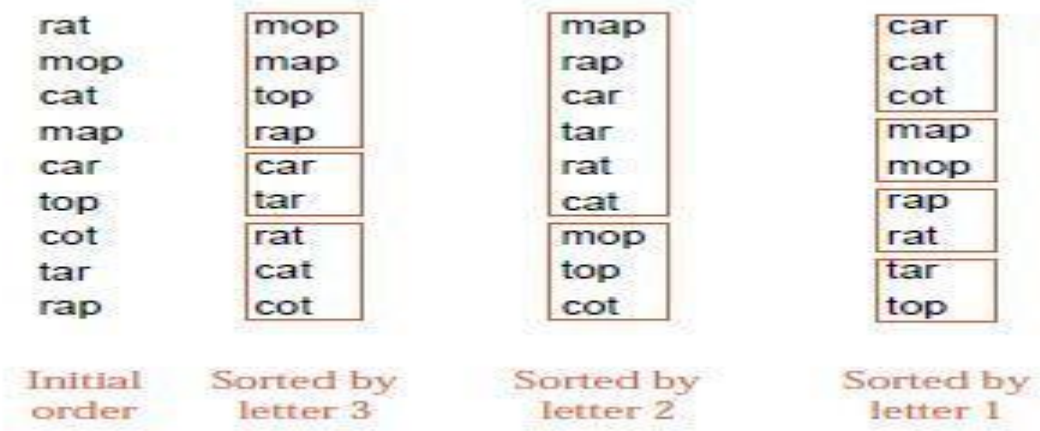
Algorithm	Worst Case	Average Case	Best Case
Heap Sort	O(n logn)	O(n logn)	O(n logn)

5.3.7 Radix Sort

The idea is to consider the key one character at a time and to divide the entries, not into two sub lists, but into as many sub lists as there are possibilities for the given character from the key. If our keys, for example, are words or other alphabetic strings, then we divide the list into 26 sub lists at each stage. That is, we set up a table of 26 lists and distribute the entries into the lists according to one of the characters in the key.

How Radix Sort Works

A person sorting words by this method might first distribute the words into 26 lists according to the initial letter (or distribute punched cards into 12 piles), then divide each of these sub lists into further sub lists according to the second letter, and so on. The following idea eliminates this multiplicity of sub lists: Partition the items into the table of sub lists first by the least significant position, not the most significant. After this first partition, the sub lists from the table are put back together as a single list, in the order given by the character in the least significant position. The list is then partitioned into the table according to the second least significant position and recombined as one list. When, after repetition of these steps, the list has been partitioned by the most significant place and recombined, it will be completely sorted. This process is illustrated by sorting the list of nine three-letter words below.



Radix Sort Algorithm

Radixsort(A,d)

1. For i←1 to d
2. Do use a stable sort to sort array A on digit i

Share this:

Complexity of Radix Sort Algorithm

The list A of n elements $A_1, A_2, \ldots\ldots\ldots\ldots A_n$ is given. Let d denote the radix(e.g d=10 for decimal digits, d=26 for letters and d=2 for bits) and each item A_i is represented by means of s of the digits:
$A_i = d_{i1}\ d_{i2}\ldots\ldots\ldots\ldots\ldots\ d_{is}$

The radix sort require s passes, the number of digits in each item . Pass K will compare each d_{ik} with each of the d digits. Hence

$C(n) \leq d*s*n$

Algorithm	Worst Case	Average Case	Best Case
Radix Sort	$O(n^2)$	d*s*n	O(n logn)

```
include <stdio.h>
#include<stdio.h>
int Max_value(int Array[], int n) // This function gives maximum value in array[]
{
   int i;
   int maximum = Array[0];
   for (i = 1; i < n; i++){
      if (Array[i] > maximum)
         maximum = Array[i];
   }
   return maximum;
}

void radixSortalgorithm(int Array[], int n) // Main Radix Sort sort function
{
   int i,digitPlace = 1;
   int result_array[n]; // resulting array
   int largest = Max_value(Array, n); // Find the largest number to know number of digits
   printf("\nlargest = %d", largest);
   while(largest/digitPlace >0){
      int count_array[10] = {0};
      for (i = 0; i < n; i++) //Store the count of "keys" or digits in count[]
         count_array[ (Array[i]/digitPlace)%10 ]++;
      for (i = 1; i < 10; i++)
         count_array[i] += count_array[i - 1];
      for (i = n - 1; i >= 0; i--) // Build the resulting array

      {
```

```
            result_array[count_array[ (Array[i]/digitPlace)%10 ] - 1] = Array[i];
            count_array[ (Array[i]/digitPlace)%10 ]--;
        }
        for (i = 0; i < n; i++) // numbers according to current digit place
            Array[i] = result_array[i];
            digitPlace *= 10; // Move to next digit place
    }
}

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

int main()
{
    int array1[] = {204,343,40,939,607,100,586,730};
    int n = sizeof(array1)/sizeof(array1[0]);
    printf("Unsorted Array is : ");
    displayArray(array1, n);
    radixSortalgorithm(array1, n);
    printf("\nSorted Array is: ");
    displayArray(array1, n);
    return 0;
}
```