Chapter 5 :- Sorting & Hashing

1. SELECTION SORT:

Description of selection sort:

Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

How Selection sort works ?

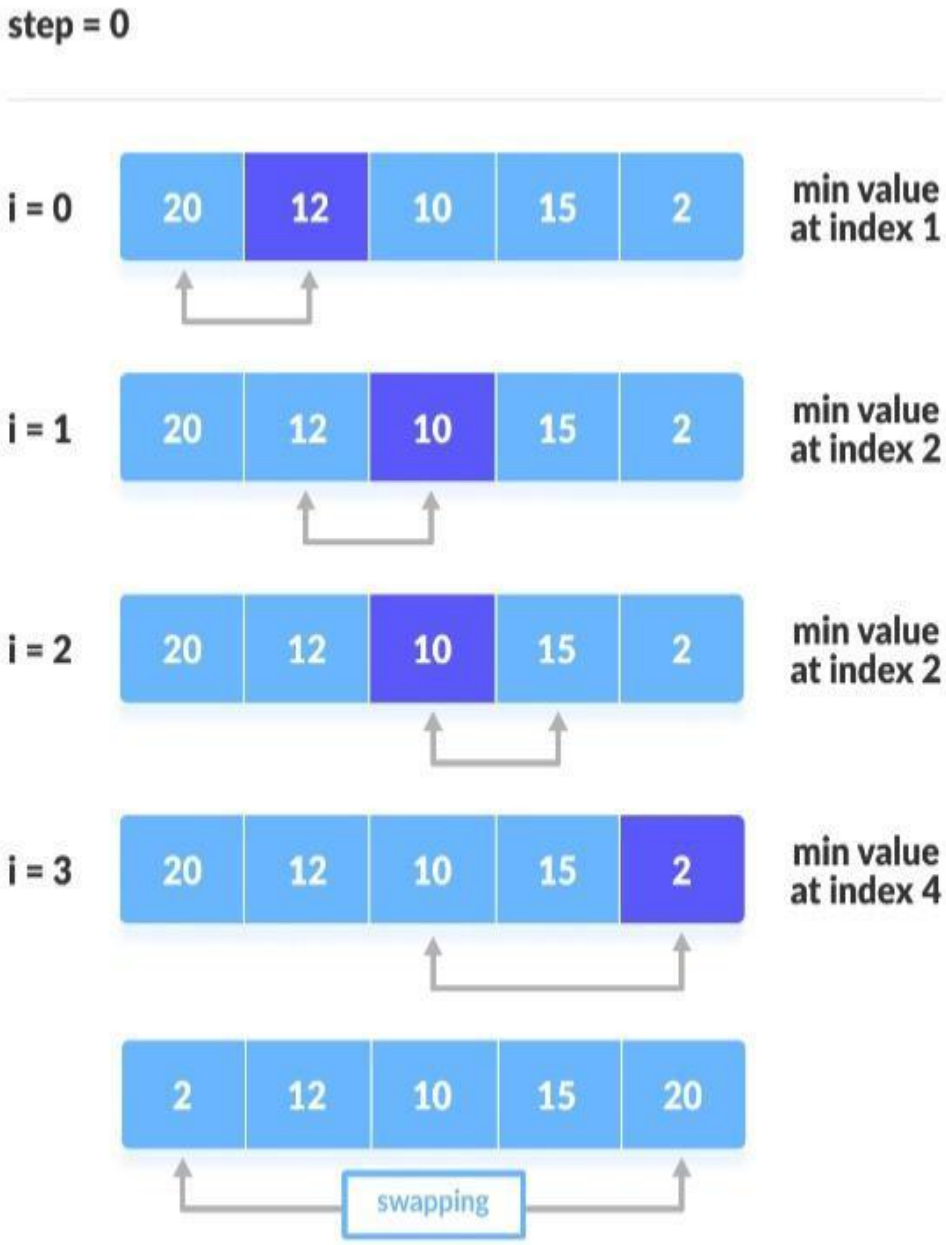
1. Set the first element as minimum.

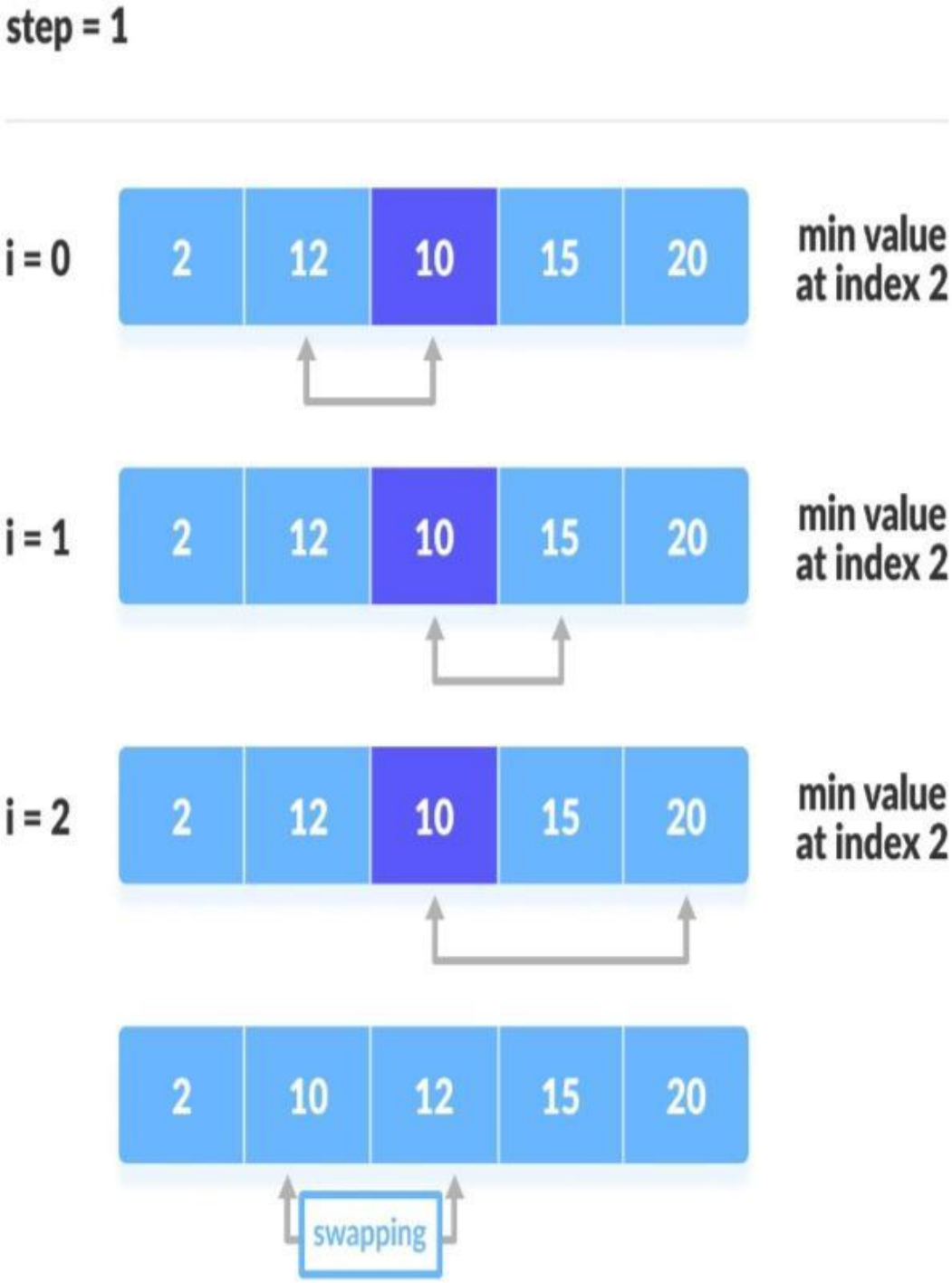
2. Compare minimum with the second element. If the second element is smaller than minimum, assign second element as minimum. Compare minimum with the third element. Again, if the third element is smaller ,then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

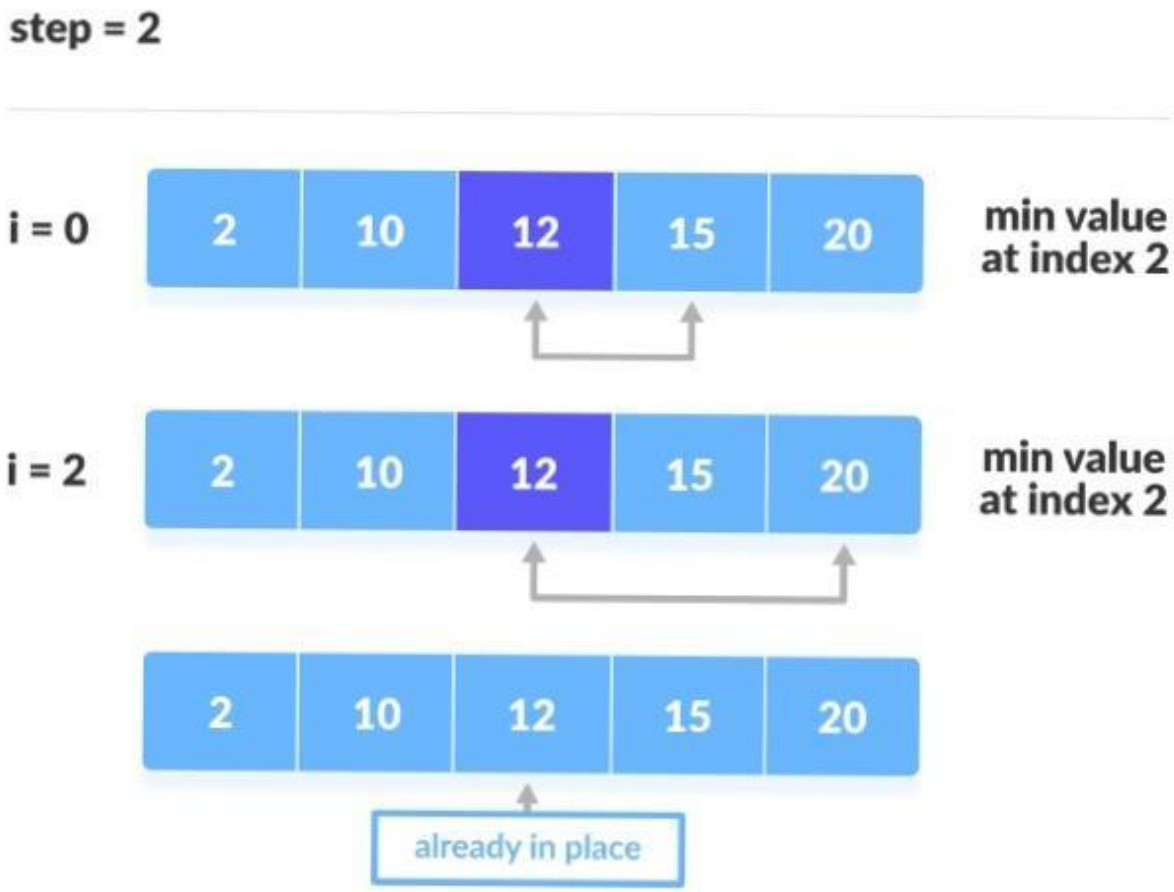
3. After each iteration, minimum is placed in the front unsorted list.

4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

Example:


step = 0
i = 0
20 12 10 15 2
min value at index 1
i = 1
20 12 10 15 2
min value at index 2
i = 2
20 12 10 15 2
min value at index 2
i = 3
20 12 10 15 2
min value at index 4
2 12 10 15 20
swapping


step = 1
i = 0
2
12
10
15
20
min value
at index 2
i = 1
2
12
10
15
20
min value
at index 2
i = 2
2
12
10
15
20
min value
at index 2
2
10
12
15
20
swapping


step = 2
i = 0
2
10
12
15
20
min value
at index 2
i = 2
2
10
12
15
20
min value
at index 2
2
10
12
15
20
already in place

ALGORITHM:-

```
For i<-0 to n-2 do
{
        Min<-i
For j<- i+1 to n-1 do
{
        If(a[j]<a[min])
                Min<-j

}

Temp<- a[j]
```

A[j]<-a[min]

A[min]<-temp

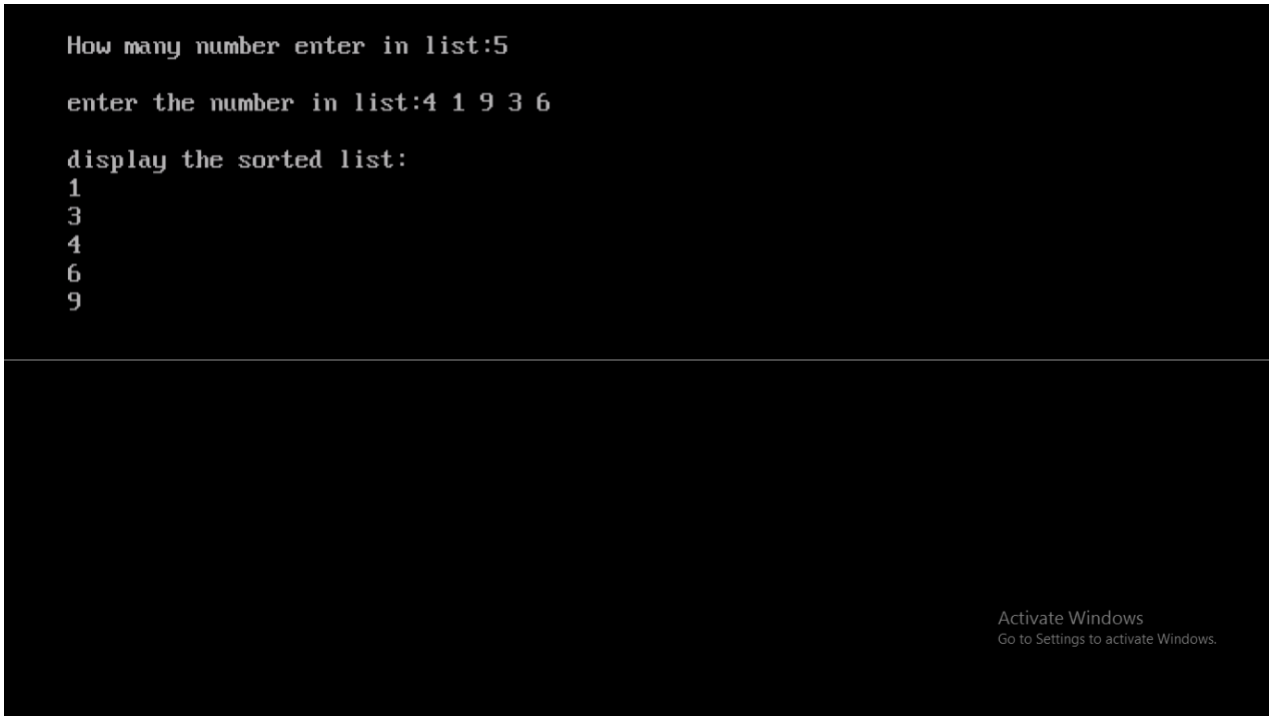
}

Program:- Straight selection sort program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int a[100],n,i,j,min,temp;
  clrscr();
  printf("\nHow many number enter in list:");
  scanf("%d",&n);
  printf("\nenter the number in list:");
  for(i=0;i<n;i++)
  {
          scanf("%d",&a[i]);
  }
  for(i=0;i<n-1;i++)
  {
          min=i;
          for(j=i+1;j<n;j++)
          {
                  if(a[min]>a[j])
                          min=j;
          }
          if(min!=i)
          {
                   temp=a[i];
                   a[i]=a[min];
                   a[min]=temp;
          }
  }
  printf("\ndisplay the sorted list:");
  for(i=0;i<n;i++)
  {
          printf("\n%d",a[i]);
```

```
        }
    getch();
}
```

OUTPUT:-



Application of selection sort:-

1. Small list is to be sorted.
2. Cost of swapping does not matter.
3. Checking of all the elements is compulsory.
4. Cost of writing to a memory matters like in flash memory .

2. INSERTION SORT

ALGORITHM

Step 1:[Intialization]
i==0, array[size], Val

Step 2:[read data in array]
while(I<size)
read (array(i))
I++

Step 3:[Insertion Opertaion]
I=1
while(i<size)
{
Val=array[i]
j=i+1
while(j>0 && array[j]>val)
{
array[j]=array[j-1]
j--;
}
}

Step 4:[if j is not equal]
then,
array[j]=val

Step 5:[Display the data]
i=0
while(I<size)
{ printer(array[i])
i++
}

Step 6:[END]
return(0)

EXAMPLE

Sorting Type

It's sorting Type is stable

Why insertion sort is stable ?

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. ... In this case, unstable sort generates problems.
- Example:

A "stable" sorting algorithm keeps the items with the same sorting key in order. Suppose we have a list of 5-letter words:

peach, straw, apple, spork

If we sort the list by just the first letter of each word then a stable-sort would produce:

apple ,peach ,straw ,spork

1. Wherever array size is less (typically less than 20)
2. If array is almost sorted – the better sorted the array is faster is insertion sort
3. Whenever space available space is less as it uses in-place sorting or else temporary storage would be required.
4. For high performance in assembly or C when dealing cache memory.
5. If data is to be read from a slower peripheral device and then sorted, then time and CPU cycles are wasted by using insertion sort this an be solved as insertion sort starts sorting parallelly to data reading.
6. For short sub sections of data generated after quick sort.
7. To sort the linked list as nodes of the linked list are added one at a time in other words it works with controlled rate-of-growth.

PROGRAM

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

```
 void main()
{
 int a[10],temp,i,j,n;
clrscr();
printf("enter number : ");
scanf("%d",&n); for(i=0;i<n;i++)
   {
         printf("a[%d]=",i);
         scanf("%d",&a[i]);
   }
 for(i=1;i<n;i++)
   {.
 for(j=i;j>0 && a[j-1]>a[j];j--)
{
temp=a[j];
 a[j]=a[j-1];
 a[j-1]=temp;
 }
printf("sorted array!\n");
for(j=0;j<n;j++)
   {
printf("%d\n",a[j]);
   }
getch();
}
```

OUTPUT

3. QUICK SORT

QUICKSORT TECHNIQUE :

Quick sort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements.

Why quick sort is better than merge sort?

This a common question asked in DS interviews that despite of better worst-case performance of merge sort, quicksort is considered better than merge sort. There are certain reasons due to which quicksort is better especially in case of arrays:

Auxiliary Space : Merge sort uses extra space, quicksort requires little space and exhibitsgood cache locality. Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array

to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space.

Worst Cases : The worst case of quicksort O(n2) can be avoided by using randomizedquicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort.

Algorithm:

Step-1:[Function call]

Quicksort_sort(k,lb,ub)

Step-2:[Flag]

flag=1,lb=0,ub=size-1

Step-3:[Perform Sorting]

if(lb>ub) then

i=lb

j=ub

key=k[lb](First pivot element) repeat while(i<j)

repeat while(k[i]<key)

i=i+1 repeat while (k(j)>key)

j=j-1;

if(i<j)

then (Swap) k[i] and k[j]

else

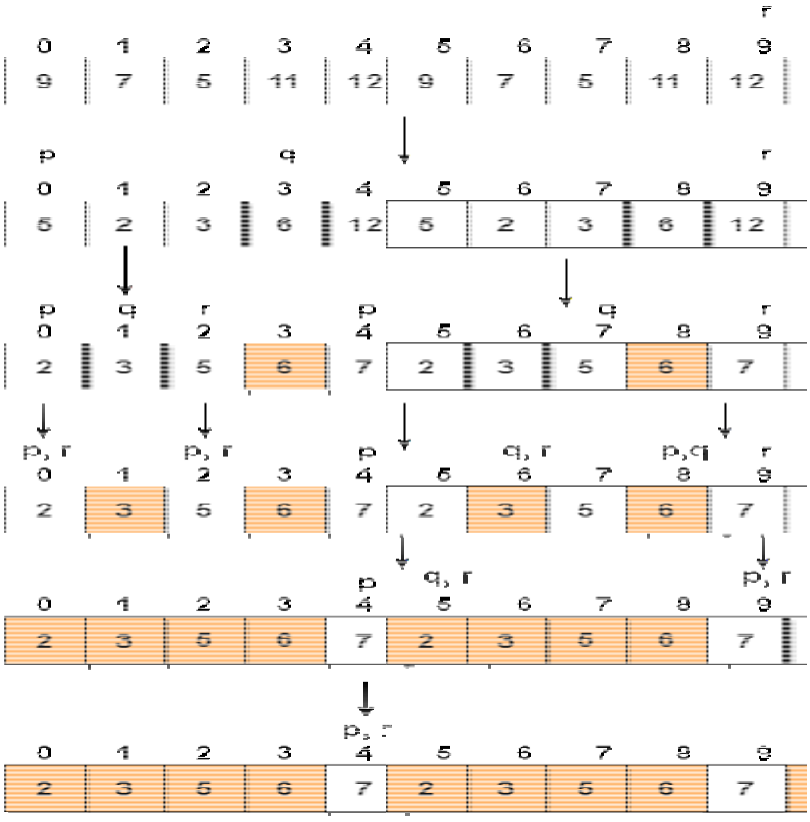
flag=0 k[lb]=k[j]

callQuick_sort(k,lb,j-1) call Quick_sort(k,J+1,ub)

Step-4:[End]

return0;

TRACING :



PROGRAM:

```
#include<stdio.h>
void quicksort(int number[25],intfirst,int last)
{
        Int i, j, pivot, temp;
        if(first<last)
        {
                pivot=first; i=first; j=last;
                while(i<j)
                {
while(number[i]>=number[pivot]&&i<last) i++;
while(number[j]<=number[pivot]) j--;
if(i<j)
{
temp=number[i];
number[i]=number[j];
 number[j]=temp;
}
}
temp=number[pivot];
 number[pivot]=number[j];
 number[j]=temp;
 quicksort(number,first,j-1);
 quicksort(number,j+1,last);
```

```
}

}

int main()

{

        inti, count, number[25];

        printf("How many elements are u going to enter?: "); scanf("%d",&count);

        printf("Enter %d elements: ", count); for(i=0;i<count;i++){

        scanf("%d",&number[i]);

}

        quicksort(number,0,count-1); printf("Order of Sorted elements:
        "); for(i=0;i<count;i++){

        printf(" %d",number[i]);

}

return 0;

}

        printf("Enter %d elements: ", count); for(i=0;i<count;i++){

        scanf("%d",&number[i]);

}

        quicksort(number,0,count-1); printf("Order of Sorted elements:
        "); for(i=0;i<count;i++){

        printf(" %d",number[i]);

}

        return 0;
```

}

APPLICATION:

CommercialUse

no additionalmemory

It runs fast

Life-critical

Medicalmonitoring

life support in aircraft andspacecraft

Want to search very fast in an array?

Use binary search for it. But it has a pre requirements sorting through quick sort is able to deal well with a huge list of items.

Amazon, EBay, Flipkart can show items sorted on the basis of price, ratings, availability etc. Sorting

SORTING TYPE:

Quick sort is unstable and in place type of sorting.

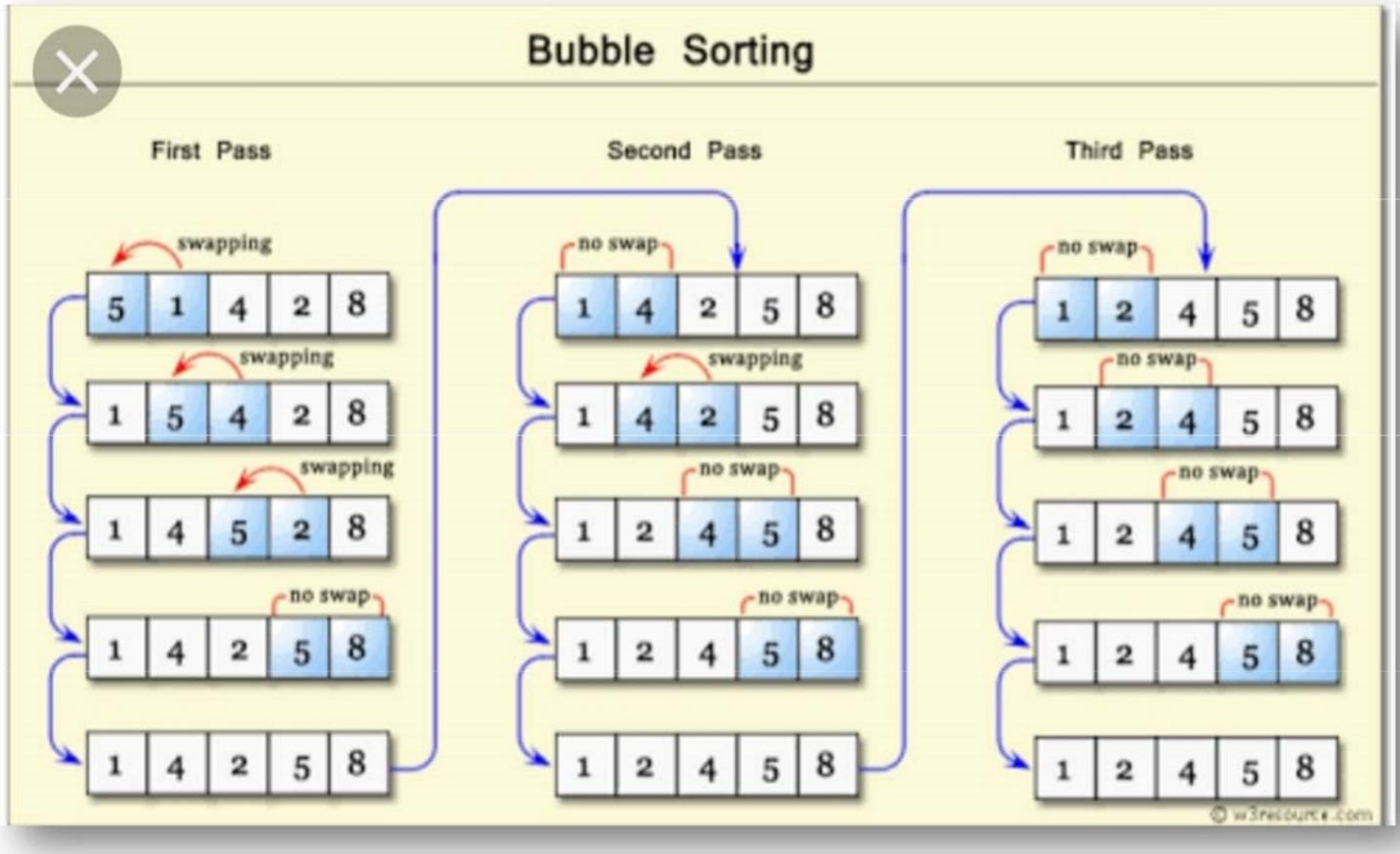
So the space efficiency of Quicksort is O(log(n)). This is the space required to maintain the call stack.

Now, according to the Wikipedia page on Quicksort, this qualifies as an in-place algorithm, as the algorithm is just swapping elements within the input data structure. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort etc. Quicksort is an unstable algorithm because we do swapping of elements according to pivot's position (without considering their original positions).

4. BUBBLE SORT

The most widely known among beginning students of programming is the bubble sort..It is easy to understand and simple sorting technique. but this sorting technique is not efficient in comparison to other sorting technique.

The fundamental thing in this technique is, for ascending order sorting during every pass the elements in the list comparing the adjacent elements and moves the small elements to top of the list. For example, a table name list have n elements, so there are at most(n-1)passes are required.



EXAMPLE

In given example, n=5 (total elements in list) so, total 4 passes(n-1) required in first pass the elements list[1] and list[2] are compared, if list[1] is bigger than list[2] then elements list[1] and list[2] are interchanged, other wise not exchange. then go for other elements list[2] and list[3] are compared and interchange if list[3] are bigger then list[2].the process is continued with all elements in the list, this is called first pass.

After one pass, the largest elements will be reach at location(position)on each successive pass, the elements with next largest value will be placed in position n-1,n-2 etc. This method will causes for small elements to move or bubble up.

ALGORITHM :

BUBBLE_SORT (LIST,N)

Step 1: [initialize]

i<-0

Step 2: while (i<n-1)repeat thru step 7

Step 3: j<-0

Step 4: while (j<n-i-1)repeat thru step 7

Step 5: if(list[j]>list(j+1))

then

(i)temp<-list[j]

(ii)list[j]<-list[j+1]

(iii)list[j+1]<-temp

Step 6: j j +1

Step 7: i i+1

Step 8: exit

PROGRAM :

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    Void bubble_sort(int[],int);
Int a[20],n,I;
Clrscr();
Printf("\n how many number enter in list:");
Scanf("%d",&n);
Printf("\n enter the number in list:");
     For(i=0;i<n;i++)
Scanf("%d",&a[i]);
Bubble_sort(a,n);
}
  Void bubble_sort(int b[],int n)
{
IntI,j,temp;
    For(inti=0; i<n-1; i++)
    {
    For(int j=0;j<n-i-1; j++)
    {
          If(b[j]>b[j+1])
           {
                Temp=b[j];
                b[j] = b[j=1];
```

```
                b[j+1] = temp;
            }
        }
}
Printf("\n display the sorted list:");
For(inti=0;i<n; i++)
Printf("\n %d",b[i];
Getch();
}
```

5. MERGE SORT:

INTRODUCTION –

Merging is the process of combining two or more tables or lists into third list or table.

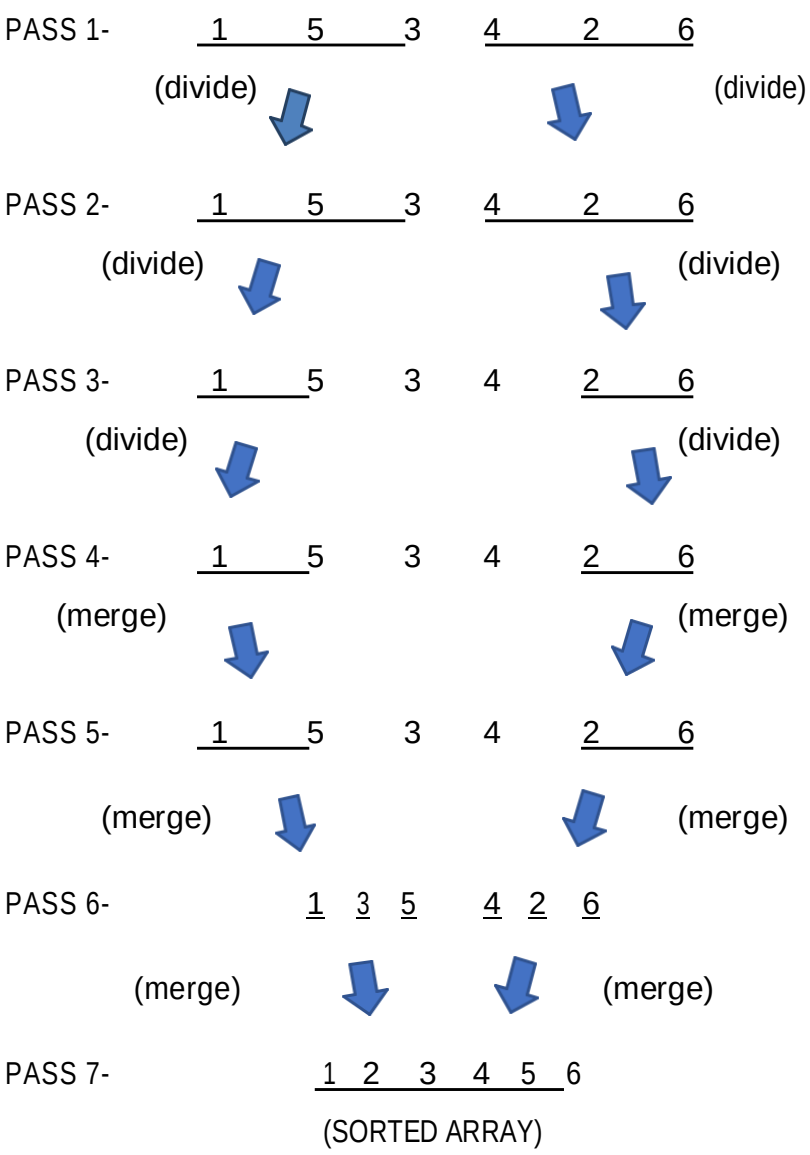
Merge sort is a sorting algorithm, which is commonly used in computer science.

It works by recursively breaking down a problem into two or more sub-problems of the Same or related types, until it become a simple enough to be solved directly. the solutions of sub-problems are then combined to give a solution to the original problem. So, merge Sort first divides the array into equal halves and then combines them in a sorted manner.

WORING OF MERGE SORT-

1. It is only one element in the list is already sorted, return.
2. Divide the list recursively into two halves until it can no more be divided.
3. Merge the smaller lists into new list in sorted order.

EXAMPLE –



Use of merge sort-

1. Merge sort is used in sorting linked list.
2. Overall time complexity of merge sort is 0. It is more efficient as it is in worst case also the runtime is 0.
3. The space complexity of its 0(n). its means is takes more space and may slower down operations for the last data sets.

Algorithm of merge sort –

```
Sorted array: output –Combine (a [0……n-1])
If (low< high) then
{
        Mid <- (low+ high)/2
        Merge sort (a, low, mid)
        Merge sort (a, mid+1, high)
        Combine (a, low, mid, high)
}
```

ALGORITHM:

```
combine (a, [0……n-1]), low, mid, high
        {
                K<-low;
                I<-low;
                J<-mid+1
                While (i<=mid & j<=high) do
        {
                If (a[i]<=a[j]) then
           {
                Temp[k]<-a[j]
                j<-j+1
                k<-k+1
           }
        Else
           {
                Temp[k]<-a[j]
                j<-j+1
                k<-k+1
           }
        }
        While (i<=mid) do
```

```
        {
                Temp[k]<-a [ j]

                i<-i+1
                k<-k+1
        }
        While (j<=high) do
        {
                Temp[k]<-a[j]
                j<-j+1
                k<-k+1
        }
```

PROGRAM

```
#include<conio .h>
#include<stdio .h>
Void main ()
{
        Void merge _sort (int [], int, int [], int, int []);
        Int a [20], b [20], c [20], n, m, I, j;
        clrscr ();
        Printf ("enter 1st lists:");
        Scanf ("%d", &n);
        Printf ("enter 2nd lists:");
        Scanf ("%d", &m);
        Printf ("enter the no of 1st list:");
        For (i=0; i<m; i++)
        Scanf ("%d", &a[i]);
        Printf ("enter the no of 2nd list:");
        For (j=o; j<m; j++)
        Scanf ("%d", &a[j]);
```

```
        Merge sort (a, n, b, m, c);
}
Void merge sort (int a [], int n, int b [], int m, int c [])
{
        Void merge sort (int list1 [], Int n, int list2 [], int m, int list3 [])
        {
                Int i=j =k =0, x;
                While ((i< n) && (j<m))
                {
                        If (list1 [i] < list2 [j])
                        {
                           List3 [k] = list[i];
                           I++;
                           K++;
                        }

If (list1 [i] > list2[j])
{
        List3 [k] = list2 [j];
        J++;
        K++;
}
Else
{
        List3 [k] = list1 [i];
        I++;
        K++;
}
If (I < n)
{
        For (int x = I; x < n; x++)
        {
```

```
                List3 [k] = list1 [x];
                k++;
            }
        }
        Else
        If (j < m)
        {
            For (int y=j; y < n; y++)
            {
              List3[k] = list2[y];
              k++;
            }
        }
}
        Printf (“display sorted list:”);
        For (int I = 0; I < k; i++)
        Printf (“%d”, list[k]);
Getch();
}
```

6. RADIX SORT :

INTRODUCTION -

Radix sort is sorted by making pockets sorted according to unit’s digit, tens digit, hundred digit and so on.

It avoids the comparison by creating and distributing elements to into buckets according their radix.

Radix sort is also called **Bucket Sort** and **Digital Sort**.

USE -

Practically used in card sorter machine.

Also used when large list or large digit of number are sorted.

METHOD -

In this method, elements are sorted **digit by digit**.

In first pass : Lists are sorted according to the unit's digit,

In second pass : Lists are sorted according to the tens digit, and

In third pass : Lists are sorted according to the hundred digit,

And this process is repeated upto N digits.

After this, combine the pockets to complete the radix sort.

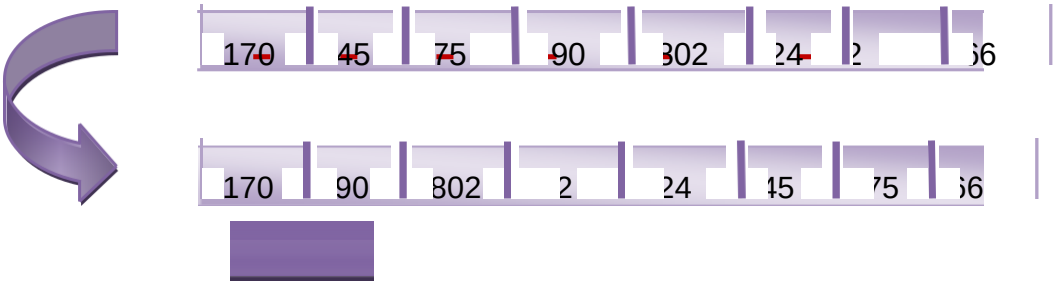
In this process, the numbers are placed in the pockets according to the value in the unit position [i.e. 23, 12 : 12, 23] and then numbers are collected from the pockets and once again are placed in the pockets.

EXAMPLE -

Consider an input array :

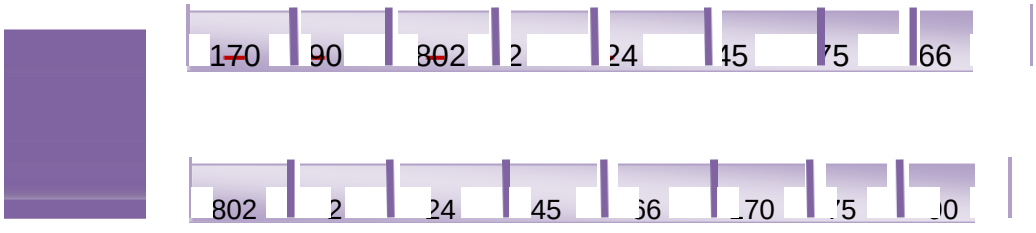
Pockets :

First consider the **1's** place [Pass 1] :

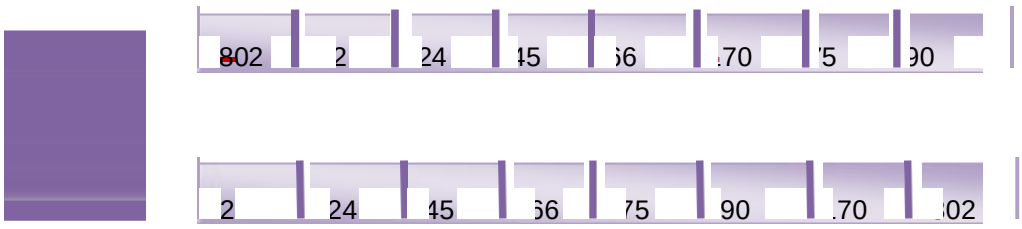


Observe that**170** has come before **90.**This is because it appeared before in the original list.

Now consider the **10's** place [Pass 2] :



Now consider **100's** place [Pass 3] :



Hence, we get the **sorted elements** :

NOTE :- The elements that come first in sequence is placed first in the array.

Highest no.of digits = no. of passes

EXAMPLE :- 170 45 75 90

Then, **170** is placed first instead of **90 [1's place]**

	90	**75**
i.e :	170	45

Elements are : 170 90 45 75

ALGORITHM :-

First address of first record

R store address of rear record pocket F store address of front record pocket

Step 1: Start

Step 2: Input N elements

Step 3: Repeat thru st for j = 1, 2 . . . . . .N

Step 4: Repeat for i = 1, 2. . . . . .9 [for pockets]

F[i]r[i] Null

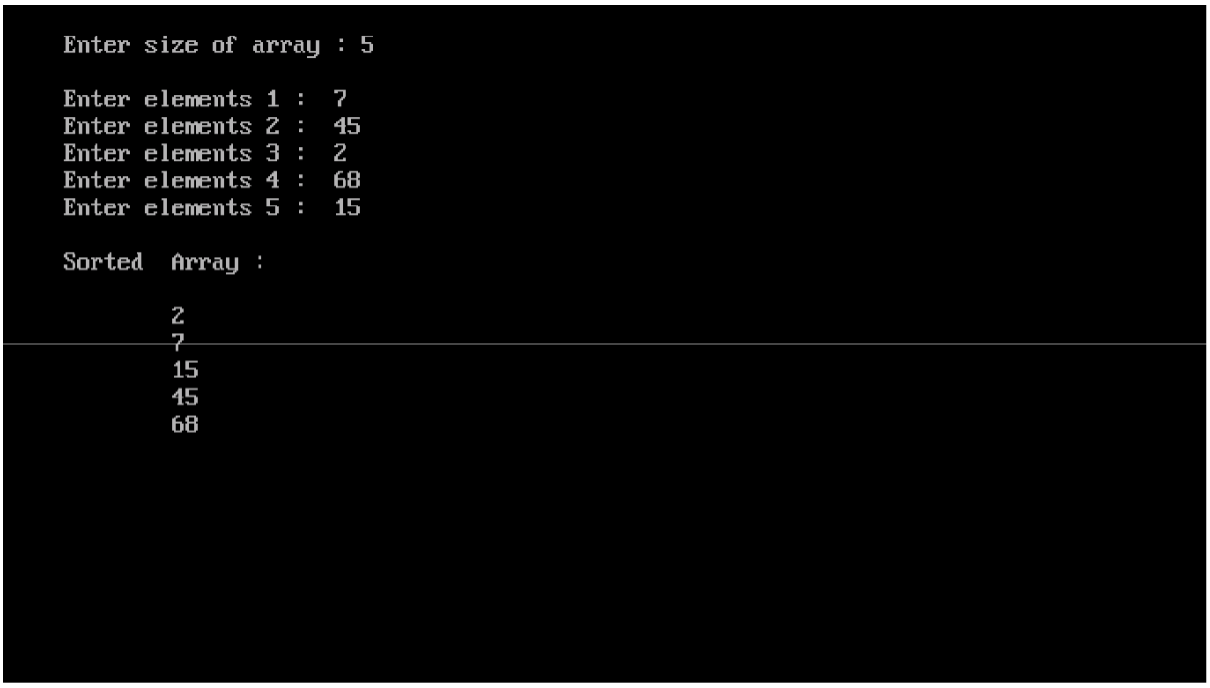
```
Step 5 : C      first    [ C is the address of current record ]
         D      d
         Next   a[C]
                If ( r(D) = Null)
                        Then
                            R(D)    F(D)    C
                        Else
                            A( R[D]     C)
                            R[D]    C
                            A[C]    Null
                            C       Next
Step 6 :  P     0                                   [ P is the temporary index value ]
Repeat while ( F[P] = Null)
            P   P + 1 First
            F [ P ]
Repeat for I = P + 1, P + 2 . . . .9
If ( R [I] <> Null)
                    A(PREV)     F[I]
                    Else
                        R[I]    PREV
Step 7 : End
```

C PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
clrscr();
int a[10][10],r,c,i,n,b[5],temp;
printf("\nEnter size of array : ");
scanf("%d",&n);
```

```
printf("\n");
            for(r=0;r<10;r++)
            {
              for(c=0;c<10;c++)
              a[r][c]=10;
             }
for(i=0;i<n;i++)
{
printf("Enter elements %d:",i+1);
scanf("%d",&b[i]);
r=b[i]/10;
c=b[i]%10;
a[r][c]=b[i];
}
printf("\nSorted  Array :\n");
                  for(r=0;r<10;r++)
{ for(c=0;c<10;c++)
{ for(i=0;i<n;i++)
{ if(a[r][c]==b[i]) {
printf("\n\t");
printf("%d",a[r][c]);
                        }
                      }
                    }
                  }
getch();
}
```

OUTPUT



HASHING :

A new data structure called a Hash table , whose search can be independent of the number of entries in a table.

For this, the position of a particular entry in the table is determined by the values of key for that entry, this can be achieved by HASHING FUNCTION.

A Hash tables are a common data structures, they consists of an array and a mapping. The hash functions maps the key in to hash values. It can stored in hash table and must have keys. The hash function maps of a key of an item to Hash value, and that hash value is used as an index in to the hash table for that item, this allow item to be inserted and located quickly.

Hash function OR Hash table method :

A function that transforms a key in to a table index is called a HASH FUNCTION.

IF H is a hash function and key is a m then H(m) is called a Hash of key m and is the index at which a record with the key m should be placed.

There are different Hash table methods to build various Hash functions, such as :

1. THE DIVISION METHOD :

- This approach is to divide a key value of record by an appropriate number then to use the remainder of division as the relative address for the records, such hash function is known as the division remainder method or simply the DIVISION method.
- The hash function is defined as :

$$H(X) = X \bmod m + 1$$

- It has been found that division method is best when table size is prime.

2. THE MID-SQUARE METHOD :

- The another hash function, known as the mid square method, the key is multiplied by itself and the middle few digits of the square are used as the index.
- If the square is considered as a decimal number, the table size must be a power of 10, whereas if it is considered as a Binary number, the table size must be a power of 2/
- Example :

Key value	(squared values) Key values	Relative address (position 7-10 from right side)
123456789	15241578750190521	8750
987654321	975461055789971041	5789
123456790	15241578997104100	8997
000000472 `	00000000000222784	0000

- Unfortunately this method does not yield uniform hash values and does not perform as well as previous technique

3. THE FOLDING METHOD :

- The folding method breaks up a key into several segments that are added or exclusive ordered together to form a hash values.
- For example, suppose that the internal bit string representation of a key is 01011 10010 10110 and that 5 bits are allowed in the index, the 3 binary string 01011, 10010, 10010, 10110 are exclusive ordered to produce 01111, which is 15 as binary integer.

COLLISION RESOLUTION TECHNIQUE :

- **COLLISION** :A failed attempt to insert an item in a table, because there is alreadyan item in the slot when the new item hashed to.

THERE ARE DIFFERENT METHODS FOR SOLVING THE PROBLEM OF COLLISION :

(1). Open addressing

(2). Chaining

(3). Re-Hashing

(4). Using neighboring slot

(5). Quadratic probing

(6). Random probing

(1) - Linear Probing :

- This is one of the simplest method. When collision occurs, look in the neighbouring slot in the table, if slot is empty, it calculates the new address extremely quickly.
- This technique will do sequencial search for the new address for collide record, so this method searches in a straight line, so it is called linear probing.
- DRAWBACK : When the table becomes full, there is tendency towards clustering.
- Clustering means, two keys that hash in two in to different values compete with each other in successive rehashes, is called clustering.

(2) - Random Probing :

- This method generates a random sequence of positions rather than an ordered sequence in Case of linear probing.

(3) - Quadratic Probing :

- If there is collision at hash address H, this method probes the location at h+1, h+4, h+9…..for

  I = 1,2,3……

- A quadratic probes suffer from a different and more subtle clustering problem. This occurs because all the keys that hash to a particular cell follow the same sequence in trying to find a vacant space.
- The secondary clustering is not a serious problem, but Quadratic probing is not used to eliminate secondary clustering.

(4) - Rehashing (Double hashing) :

- Re-hashing technique use a secondary hashing operation when there is a collision. If there is a further collision, we re-hash until an empty slot is found.
- The re-hashing function can either be a new function or a reapplication of the original one.
- Using re-hashing to eliminate primary as well as secondary clustering.