Design Of Data Structures

Target = 40

Course Code : 21SC1202

L - T - P - S : 3 - 0 - 2 - 4

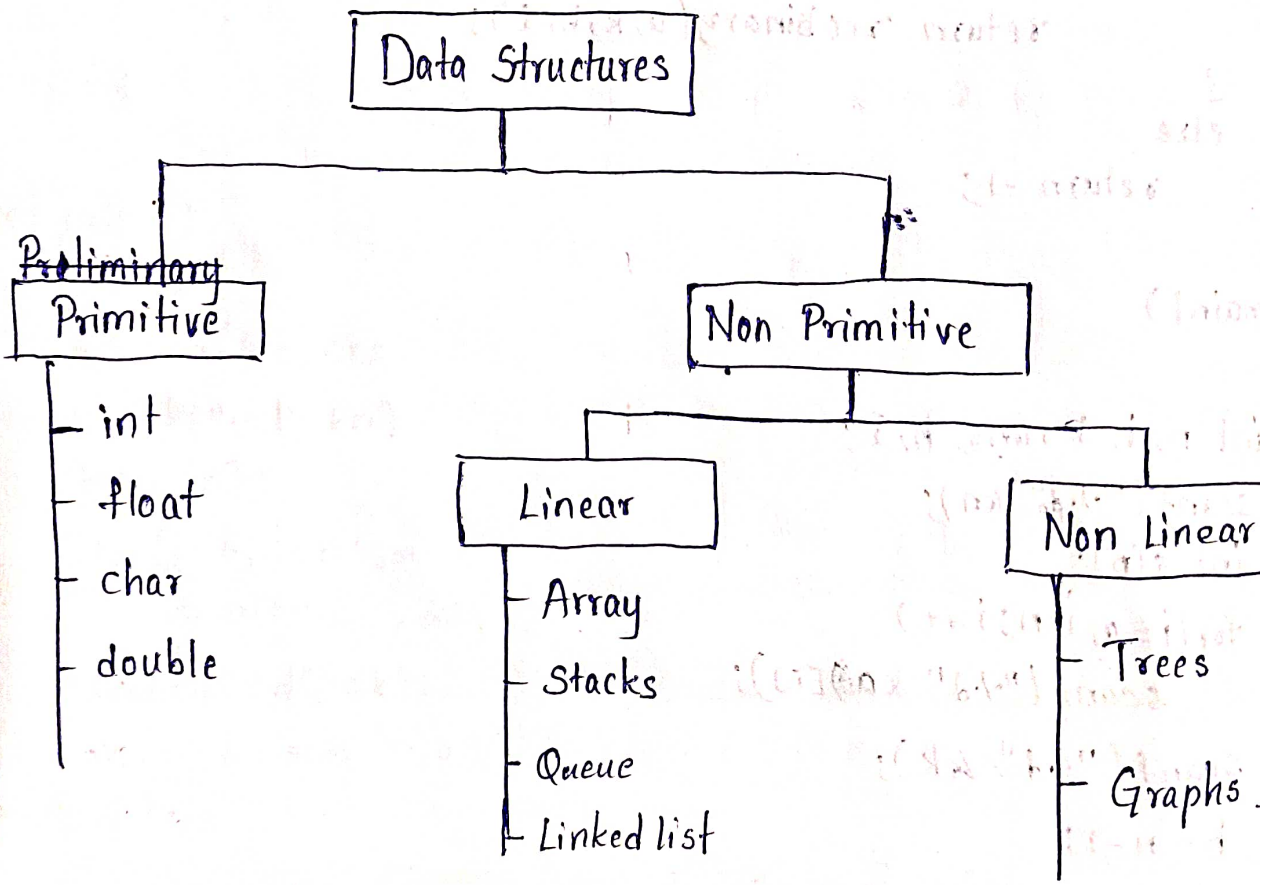
CO-1 : Algorithm Analysis & Sorting Techniques.

CO-2 : Linear Data Structures

CO-3 : Hashing, & Trees

CO-4 : Graphs

Credits : 5



Data Structures:

Organisation of data in well mannered order to

CO - 1

Algorithm Analysis and Sorting Techniques.

→ **Objective:**

To implement different sorting techniques and evaluate their efficiency.

1. Mathematical Background
2. Model
3. Analyse
4. Running time calculations.
5. Introduction to sorting techniques
6. Insertion Sort
7. Shell Sort
8. Heap Sort
9. Merge Sort
10. Quick Sort
11. Bucket Sort
12. External Sort.

→ **Introduction to Sorting Techniques**

Sorting: Arranging elements in ascending order or descending order in an array is called Sorting.

Time complexity of linear search is $O(n)$. where n represents no of elements in array

$O(n)$ → big O of n.

Time complexity of binary search is $O(\log n)$.

1. Insertion Sort

- Insertion sort is inplace comparision sorting algorithm.
- Shell, insertion, bubble, selection sort comes under inplace comparision.
- In in-place comparision, we don't use additional memory for sorting.

→ **Algorithm**

Step 1: Start

Step 2: If the array is having single element, it is already sorted.

Step 3: First element is already sorted, so pick the next element

Step 4: Compare this element with the elements in sorted sublist

Step 5: If the element is less than the elements in sorted sublist. then bring it into it's place.

Step 6: Repeat step 3 to step 5 until the array gets sorted.

Step 7: Stop

Eg:

4	1	23	17	27	10	31	18	26
0	1	2	3	4	5	6	7	8

Pass 1: 4 | ① 23 ~~17~~ ~~27~~ 10 31 18 26.

1 4 23 17 27 10 31 18 26.

Pass 2: 1 4 | (23) 17 27 10 31 18 26

No swap

1 4 ~~23~~ ~~17~~ 27 10 31 18 26.

Pass 3: 1 4 23 | (17) 27 10 31 18 26.

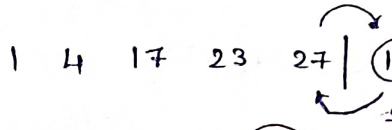
1 4 17 23 27 10 31 18 26

Pass 4: 1 4 17 23 | (27) 10 31 18 26

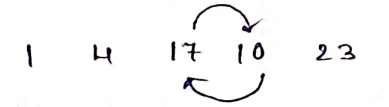
no swap

1 4 17 23 27 10 31 18 26

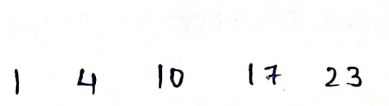
Pass 5: 1 4 17 23 27 | (10) 31 18 26



1 4 17 23 10 27 31 18 26



1 4 17 10 23 27 31 18 26



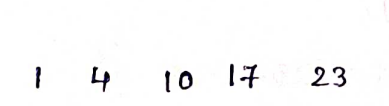
1 4 10 17 23 27 31 18 26

Pass 6: 1 4 10 17 23 27 | (31) 18 26

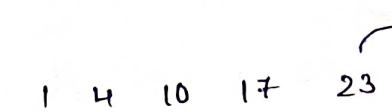
no swap

1 4 10 17 23 27 31 18 26.

Pass 7: 1 4 10 17 23 27 31 | (18) 26



1 4 10 17 23 27 18 31 26

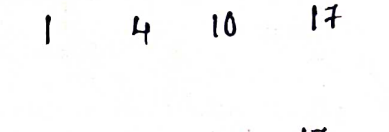


1 4 10 17 23 18 27 31 26

1 4 10 17 18 23 27 31 26

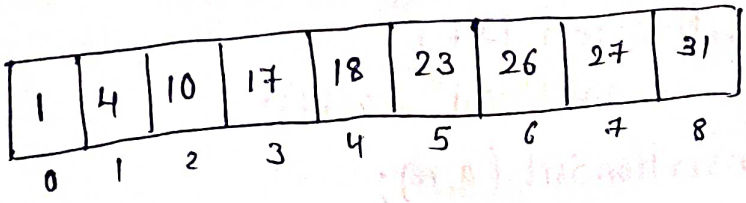
Pass 8: 1 4 10 17 ~~18~~ ~~23~~ ~~27~~ 31 | (26)

1 4 10 17 18 23 27 26 31



1 4 10 17 18 23 26 27 31

Final Sorted Array -



→ Implementation

```
#include <stdio.h>
void insertionSort(int a[], int n)
{
    int t, i, j, k;
    for(i=1; i<n; i++)
    {
        for(j=i; j>=0; j=j-1)
        {
            if(a[j]<a[j-1])
            {
                t = a[j];
                a[j] = a[j-1];
                a[j-1]=t;
            }
        }
        printf("\n After %d pass: ", i);
        for(k=0; k<n; k++)
            printf("%3d", a[k]);
    }
    printf("\nAfter Sorting: ");
    for(i=0; i<n; i++)
        printf("%3d", a[i]);
}
int main()
{
    int n,i;
    printf("Enter size of array:");
    scanf("%d", &n);
    int a[n];
    printf("Enter elements : ");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    insertionSort(a,n);
    return 0;
}
```

→ Insertion sort is inplace comparision sorting algorithm.

→ Insertion sort is flexible when list is having less no of elements.

→ It is adaptive if the array is partially sorted.

→ It is space efficient as it is using single variable (temp) for swapping.

→ The worst case time complexity of insertion sort is $O(n^2)$

Eg: Given an array of elements 30 -14 29 26 -37 11 -9 10 90
Represent the sorted list for every pass and give the final sorted array using insertion sort

30	-14	29	26	-37	11	-9	10	90
0	1	2	3	4	5	6	7	8

Pass 1 : 30 | -14 29 26 -37 11 -9 10 90

-14 30 29 26 -37 11 -9 10 90

Pass 2 : -14 30 | 29 26 -37 11 -9 10 90

-14 29 30 26 -37 11 -9 10 90

Pass 3 : -14 29 30 | 26 -37 11 -9 10 90

-14 29 26 30 -37 11 -9 10 90

-14 26 29 30 -37 11 -9 10 90

Pass 4 : -14 26 29 30 | -37 11 -9 10 90

-14 26 29 -37 30 11 -9 10 90

-14 26 -37 29 30 11 -9 10 90

-14 -37 26 29 30 11 -9 10 90

-37 -14 26 29 30 11 -9 10 90

Pass-5 : -37 -14 26 29 30 | (11) -9 10 90

-37 -14 26 29 11 30 -9 10 90

-37 -14 26 11 29 30 -9 10 90

-37 -14 11 26 29 30 -9 10 90

Pass-6 : -37 -14 11 26 29 30 | (-9) 10 90

-37 -14 11 26 29 -9 30 10 90

-37 -14 11 26 -9 29 30 10 90

-37 -14 11 -9 26 29 30 10 90

-37 -14 -9 11 26 29 30 10 90

Pass-7: -37 -14 -9 11 26 29 30 | (10) 90

-37 -14 -9 11 26 29 10 30 90

-37 -14 -9 11 26 10 29 30 90

-37 -14 -9 11 10 26 29 30 90

-37 -14 -9 10 11 26 29 30 90

Pass-8 : -37 -14 -9 10 11 26 29 30 | (90)

no swap

-37 -14 -9 10 11 26 29 30 90

Final Sorted Array

-37	-14	-9	10	11	26	29	30	90
0	1	2	3	4	5	6	7	8

2. Shell Sort

. Shell sort is inplace comparision sorting algorithm, it is generalized version of insertion sort.

O(nlogn)

→

9	8	3	7	5	6	4	1
0	1	2	3	4	5	6	7

n=8

$n = 8$

$gap = n/2 = 8/2 = 4$

9 / 8 / 3 / 7

Pass - 1 9 8 3 7 5 6 4 1

5 8 3 7 9 6 4 1

5 6 3 7 9 8 4 1 no swap

5 6 3 7 9 8 4 1

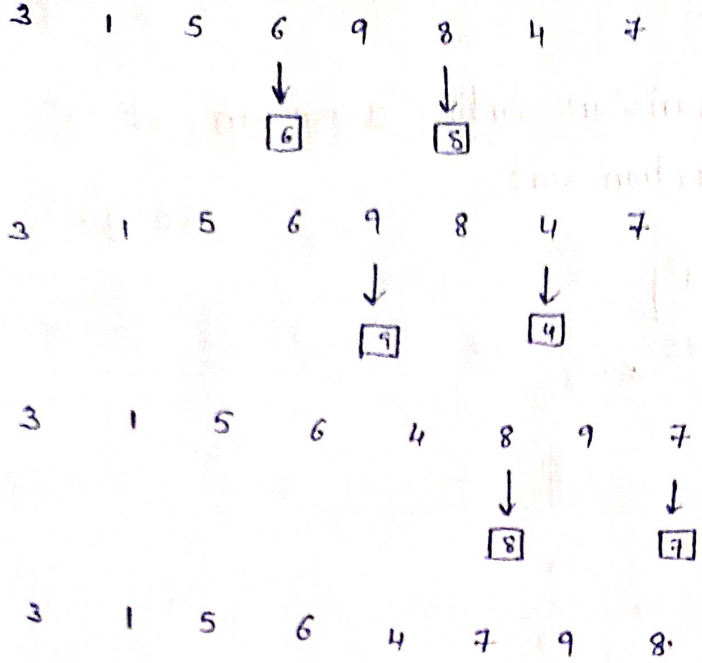
5 6 3 1 9 8 4 7

Pass 2 → $gap = gap/2 = 4/2 = 2$

5 6 3 1 9 8 4 7

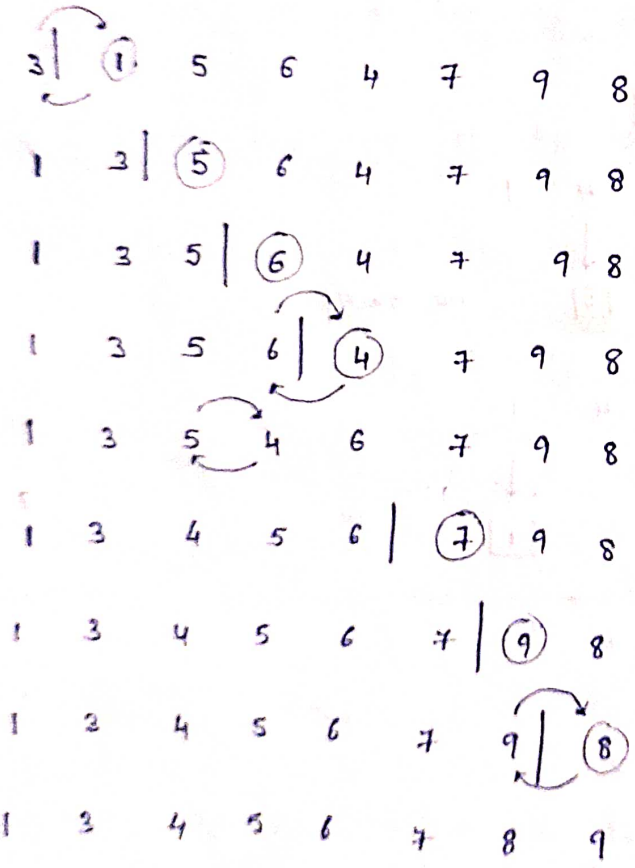
3 6 5 1 9 8 4 7

3 6 5 6 9 8 4 7



Pass - 3 : gap = gap/2 = 2/2 = 1

When gap = 1, insertion sort take place.



∴ Final Sorted Order : 1 3 4 5 6 7 8 9

→

10	7	35	9	25	23	86.
0	1	2	3	4	5	6

$n = 7$

$gap = n/2 = 7/2 = 3$.

Pass-1:

10 7 35 9 25 23 86

↓ ↓

[10] [9]

9 7 35 10 25 23 86

↓ ↓

[7] [25]

9 7 35 10 25 23 86.

↓ ↓

[35] [23]

9 7 23 10 25 35 86.

↓ ↓

[10] [86]

9 7 23 10 25 35 86.

Pass-2: $gap = gap/2 = 3/2 = 1$

When gap = 1, insertion sort take place

9 | (7) 23 10 25 35 86

7 9 | (23) 10 25 35 86

7 9 23 | (10) 25 35 86

7 9 10 23 | (25) 35 86

7 9 10 23 25 | (35) 86

7 9 10 23 25 35 | (86)

7 9 10 23 25 35 86

∴ Final Sorted Array: 7 9 10 23 25 35 86.

→

35	33	42	10	14	19	27	44
0	1	2	3	4	5	6	7

$n = 8$

$gap = n/2 = 8/2 = 4$

Pass-1: 35 33 42 10 14 19 27 44

14 33 42 10 35 19 27 44

14 19 42 10 35 33 27 44

14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

Pass-2: $gap = gap/2 = 4/2 = 2$

14 19 27 10 35 33 42 44

14 19 27 10 35 33 42 44

14 10 27 19 35 33 42 44

14 10 27 19 35 33 42 44

14 10 27 19 35 33 42 44

35 42

14 10 27 19 35 33 42 44.

33 44

14 10 27 19 35 33 42 44.

Pass - 3 : gap = gap/2 = 2/2 = 1

When gap = 1, insertion sort take place.

14 | 10 27 19 35 33 42 44.

10 14 | 27 19 35 33 42 44

10 14 27 | 19 35 33 42 44

10 14 19 27 | 35 33 42 44.

10 14 19 27 35 | 33 42 44

10 14 19 27 33 35 | 42 44

10 14 19 27 33 35 42 | 44

10 14 19 27 33 35 42 44.

∴ Final Sorted Array : 10 14 19 27 33 35 42 44.

Implementation

```
#include <stdio.h>
void shellSort (int a[], int n)
{
    int gap, i, j, t;
    for ( gap = n/2 ; gap >= 1; gap = gap/2 )
    {
        for ( i = gap; i<n; i++)
        {
            for (j = i ; j >= gap ; j = j-gap)
            {
                if ( a [j] < a [j- gap] )
                {
                    t = a [j];
                    a [j] = a [j- gap ];
                    a [j-gap] = t;
                }
            }
        }
    }
}
int main()
{
    int n, i;
    scanf ("%d", &n);
    int a[n];
    for (i=0; i<n; i++)
    {   scanf ("%d", &a [i]);
    shellSort (a,n);
    return 0;
}
```

```
for (i=0; i<n; i++)
    printf ("%d", a[i]);
```

3. Quick Sort (Partition Exchange)

$O(n \log n)$

$O(n^2)$

Algorithm for Quick Sort

Step 1: Start

Step 2: Set the pivot element as a[0].

Step 3: Set two pointers i and j, i = a[0] & j = a[n-1]

Step 4: Move i pointer towards right till the no is greater than pivot

Step 5: Move j pointer towards left till the no is less than pivot.

Step 6: if i and j cross each other. swap a[j] and pivot

Step 7: if i and j don't cross each other swap a[i] and a[j]

Step 8: Stop.

Eg:

35	33	42	10	14	19	27	44
pivot 0	1	2	3	4	5	6	7

$\uparrow_i$ $\uparrow_j$

35 33 42 10 14 19 27 44

$\uparrow_i$ $\uparrow_j$

(i & j didn't cross each other, so a[j] & a[j] swapping take place)

35 33 27 42 10 14 19 44

$\uparrow_j$ $\uparrow_i$

(i & j crossed each other, so a[j] & pivot swapped)

less than 35 | 35 | greater than 35

19 33 27 10 14 | 35 | 42 44

pivot $\uparrow_i$ $\uparrow_j$

19 33 27 10 14

$\uparrow_i$ $\uparrow_j$

→ i & j didn't cross

19 14 27 10 33

$\uparrow_i$ $\uparrow_j$

→ i & j didn't cross

19 14 10 27 33.

↑j ↑i

→ i & j crossed

pivot 10 14 19 27 33

↑i ↑j

[From here, no changes will be appeared.]

∴ Final sorted array obtained is.

10 / 14 ~~19~~ ~~27~~ ~~33~~ ~~35~~ ~~42~~ ~~44~~,

pivot 10 14 19 27 33.

↑j ↑i

Here j becomes -1

∴ Process stopped.

Final Sorted Array 10 14 19 27 33 35 42 44.

→ Quick Sort is a Divide and conquer algorithm.

It picks an element as pivot and partitions the given array around the picked pivot.

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

Quick Sort Algorithm

1. if n<=1, then return.
2. Pick any element V in a[]. This is called the pivot.
3. Rearrange elements of the array by moving all elements $x_i > V$ right of V and all elements $x_i <= V$ left of V.
4. If the place of the V after re-arrangment is j, the array is partioned as below.

all elements with value less than V, appear in a[0], a[1], a[2]. a[j-1]. and those with value greater than V appear in a[j+1] .... a[n-1]

5. Apply quick sort recursively to a[0] .... a[j-1] and to a[j+1] .... a[n-1].

Eg: 35, 33, 42, 10, 14, 19, 27, 44, 26, 31

Eg: 10, 80, 30, 90, 40, 50, 70

Eg: 2, 8, 7, 1, 3, 5, 6, 4

Implementation

```
#include<stdio.h>
void quickSort(int arr[100], int, int);
int partition(int arr[100], int, int);
int main()
{
    int arr[100], n, i;
    printf("enter no of elements in array: ");
    scanf("%d", &n);
    printf("\n Enter %d elements: \n\n", n);
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);
    quickSort(arr, 0, n-1);
    printf("\n The Sorted Array is: \n\n");
    for(i=0; i<n; i++)
        printf("%4d", arr[i]);
    return 0;
}
void quickSort(int arr[], int low, int high)
{
    int j;
    if(low < high)
    {
        j = partition(arr, low, high);
        quickSort(arr, low, j-1);
        quickSort(arr, j+1, high);
    }
```

```
int partition(int arr[100], int low, int high)
{
    int pivot, j, temp, i, k;
    if(low < high)
    {
        pivot = arr[high];
        i = low;
        j = high;
        while(i<j)
        {
            while((arr[i] <= arr[pivot]) && (i < high))
            {
                i++;
            }
            while(arr[j] > arr[pivot])
            {
                j--;
            }
            if(i<j)
            {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        temp = arr[pivot];
        arr[pivot] = arr[j];
        arr[j] = temp;
    }
    return j;
}
```

2 5 1 3 7 9 8

② 5 1 3 7 9 8

i i j

i

0<6

Eg

→ ~~35~~ ~~33~~ ~~42~~

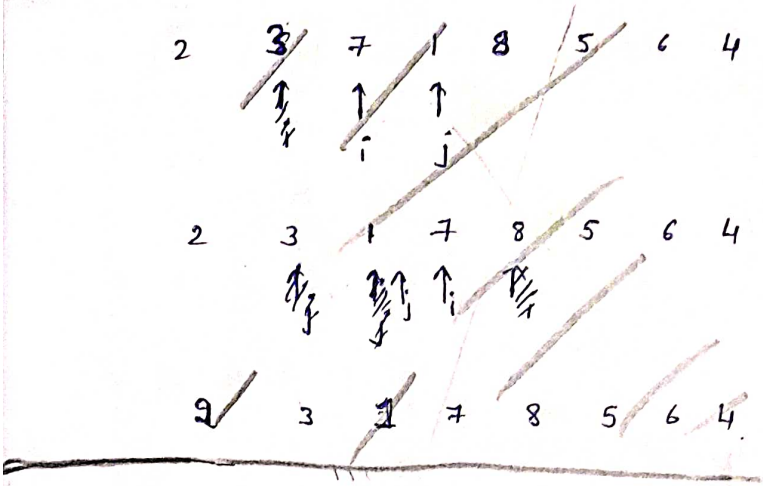
2 8 7 1 3 5 6 4

pivot i j

2 8 7 1 3 5 6 4

↑ ↑

i j



→

35	33	42	10	14	19	27	44	26	31
0	1	2	3	4	5	6	7	8	9

35 33 42 10 14 19 27 44 26 (31)P

(31)P 33 42 10 14 19 27 44 26 35

26 33 42 10 14 19 27 44 (31)P 35

26 (31)P 42 10 14 19 27 44 33 35

26 27 42 10 14 19 (31)P 44 33 35

26 27 (31)P 10 14 19 42 44 33 35

26 27 19 10 14P (31)P 42 44 33 (35)P

(14)P 27 19 10 26 | (35)P 44 33 42

10 27 19 (14)P 26 | 33 44 (35)P 42

10 (14)P 19 27 (26)P | 33 (35)P 44 (42)P

10 14 19 26 27 31 33 35 42 44

Final Sorted Array obtained = 10 14 19 26 27 31 35 42 44

→

10	80	30	90	40	50	70
0	1	2	3	4	5	6

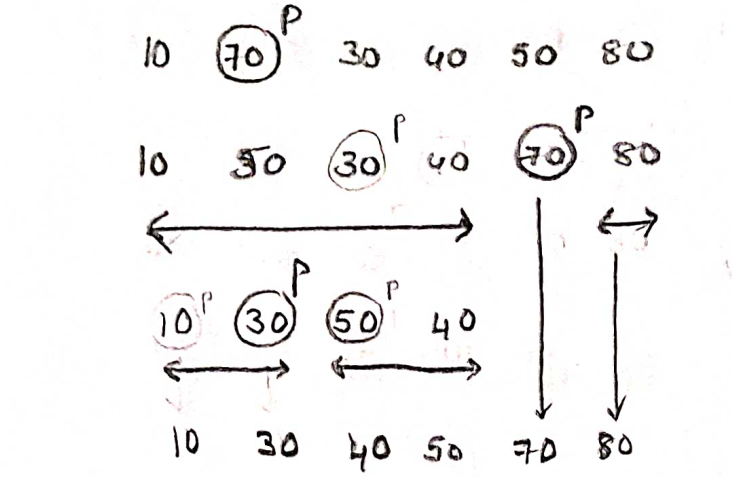
10 80 30 (90)P 40 50 70

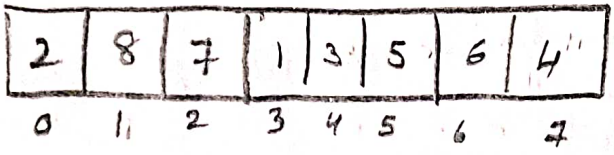
10 80 30 70 40 50 (90)P

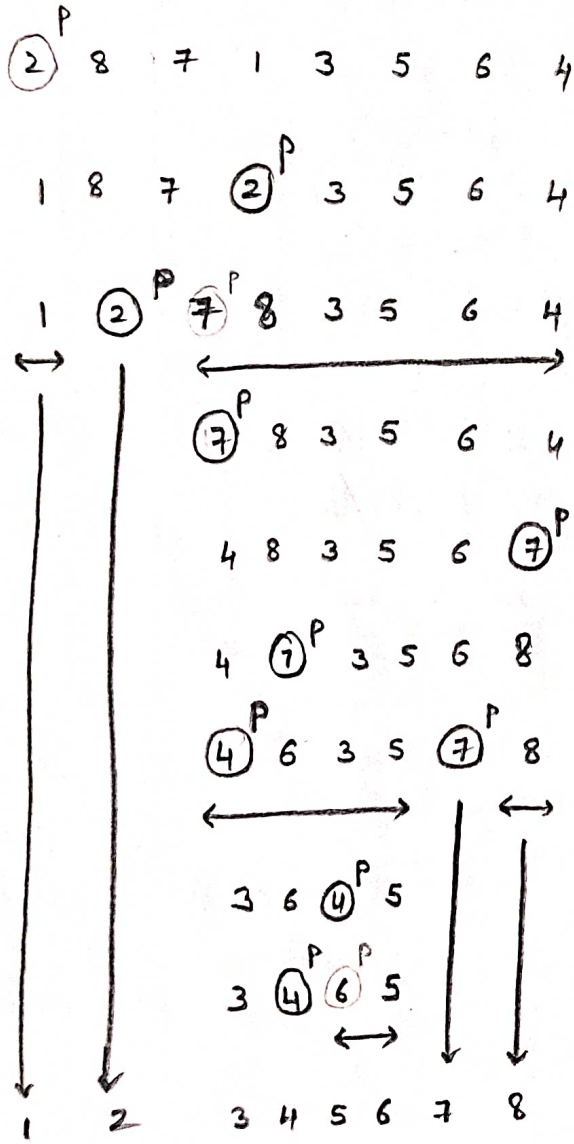
10 80 30 (70)P 40 50

10 80 30 40 (70)P 50

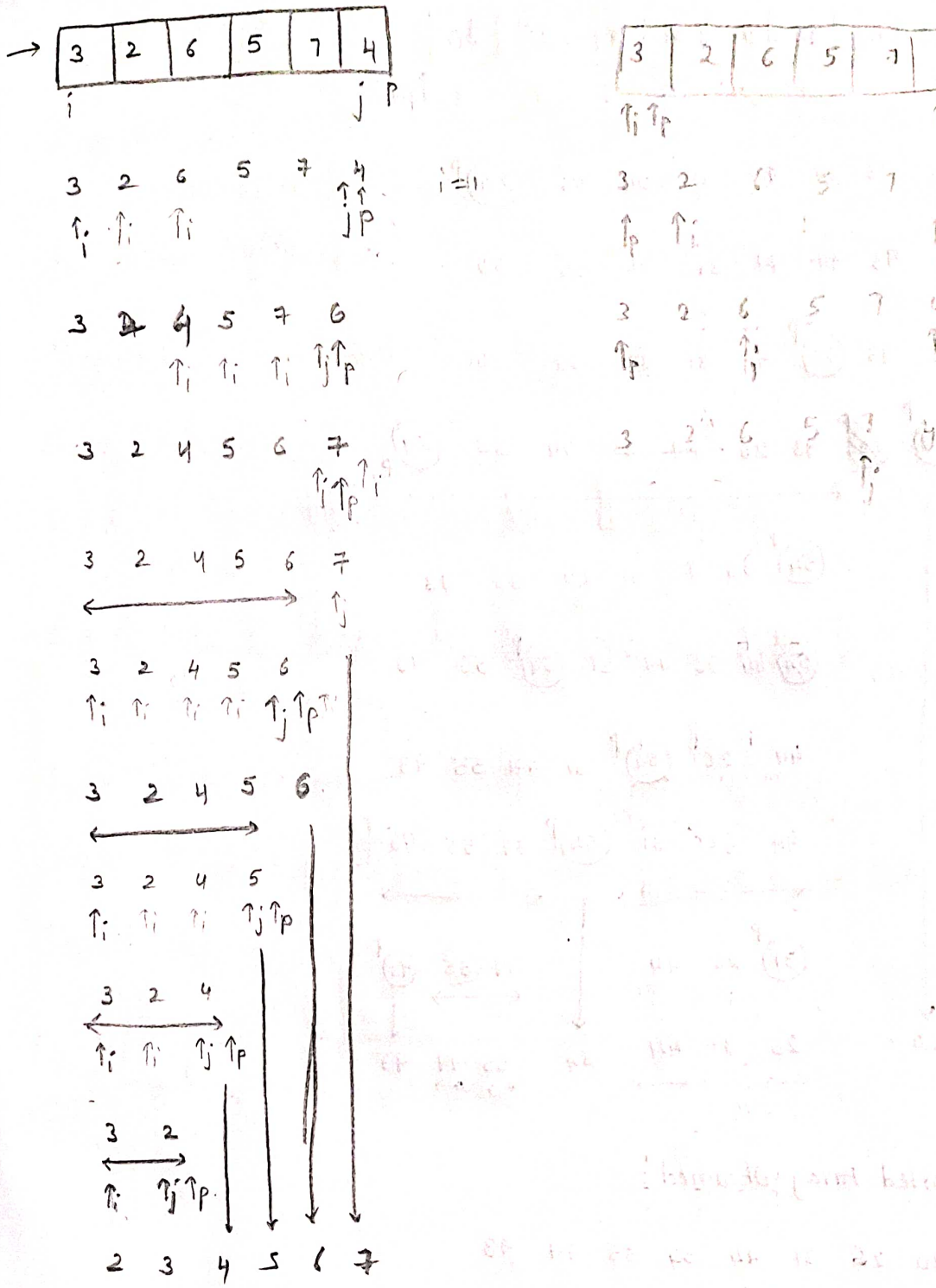
10 80 30 40 50 (70)P







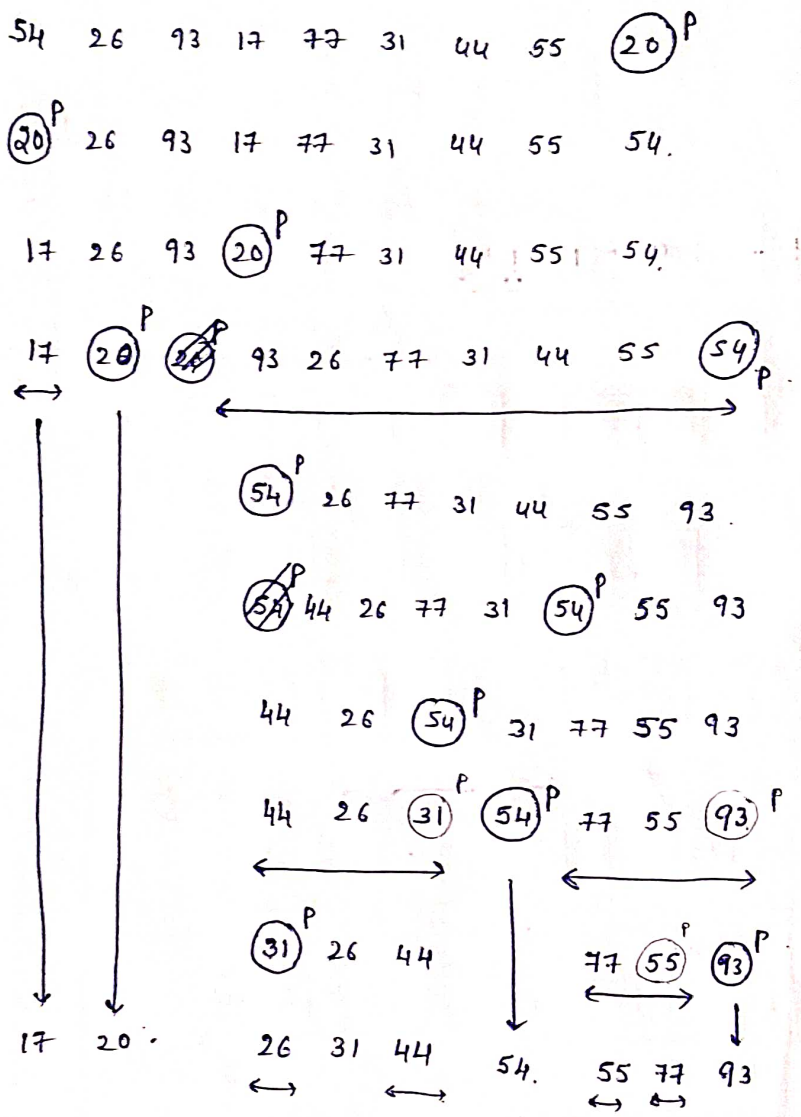
Final Sorted Array obtained : 1 2 3 4 5 6 7 8


3 2 6 5 7 4
i j p
i=1
2 3 4 5 6 7

→

54	26	93	17	77	31	44	55	20
0	1	2	3	4	5	6	7	8

↑ pivot



Final Sorted Array Obtained :

17 20 26 31 44 54 55 77 93.

→

→ Merge Sort

$O(n^2)$

Merge sort is a divide & conquer algorithm, it divides the input array into 2 halves, calls itself for the 2 halves and then merges the sorted halves.

Algorithm (divide → sort → merge)

Step 1: Start

Step 2: if the array is of length 0 or 1 then it is already sorted.

Step 3: Otherwise, divide the unsorted array into 2 sub arrays of about half the size

Step 4: Use merge sort algorithm recursively to sort each sub array.

Step 5: Merge two sub arrays to form a single sorted list.

Step 6: Stop

Eg:

5	3	8	9	1	7	0	2	6	4
0	1	2	3	4	5	6	7	8	9

$n = 10$

$l = 0$

$h = 9$

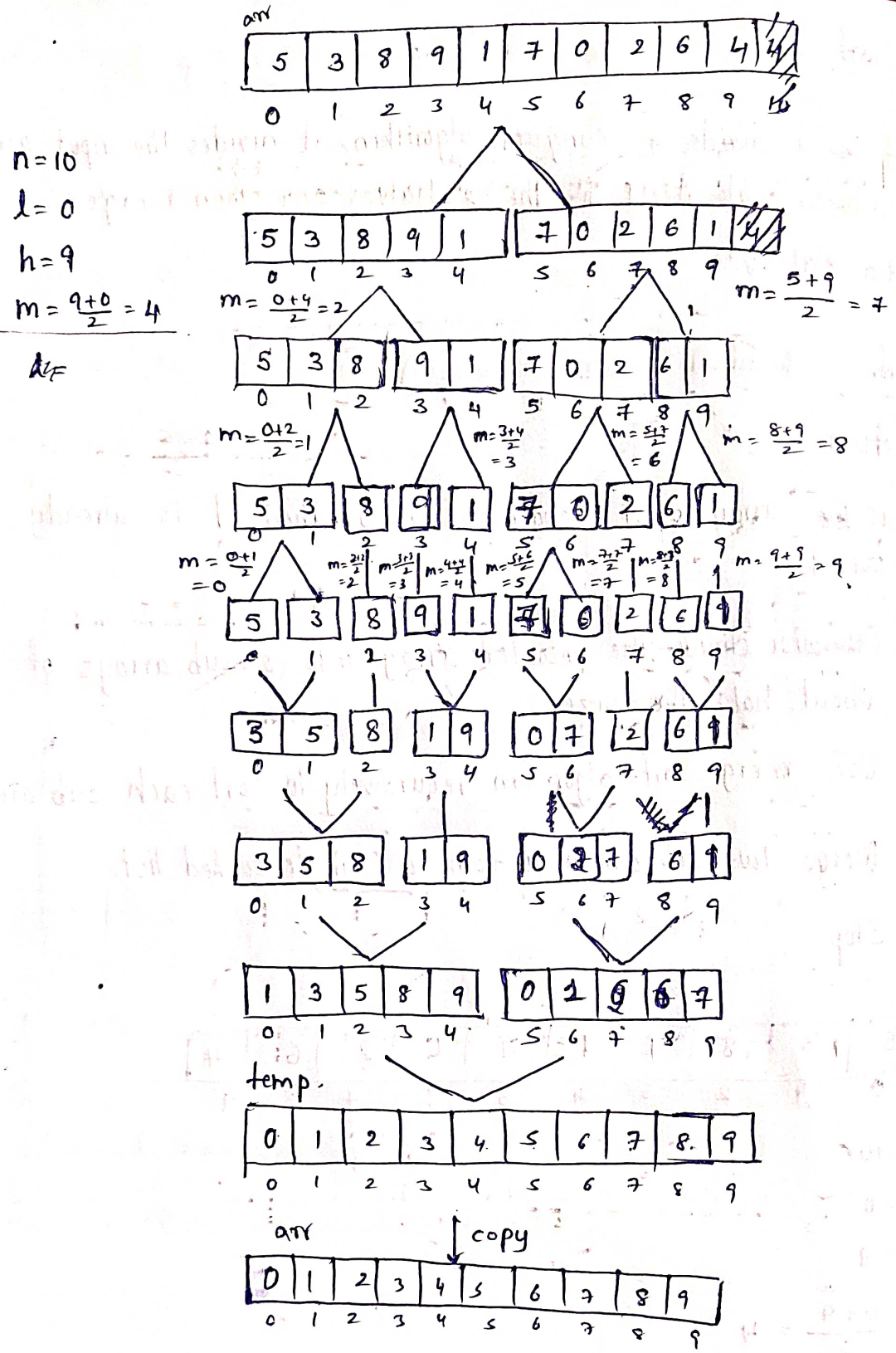
$m = \frac{0+9}{2} = 4$

5	3	8	9	1	7	0	2	6	4
0	1	2	3	4	5	6	7	8	9

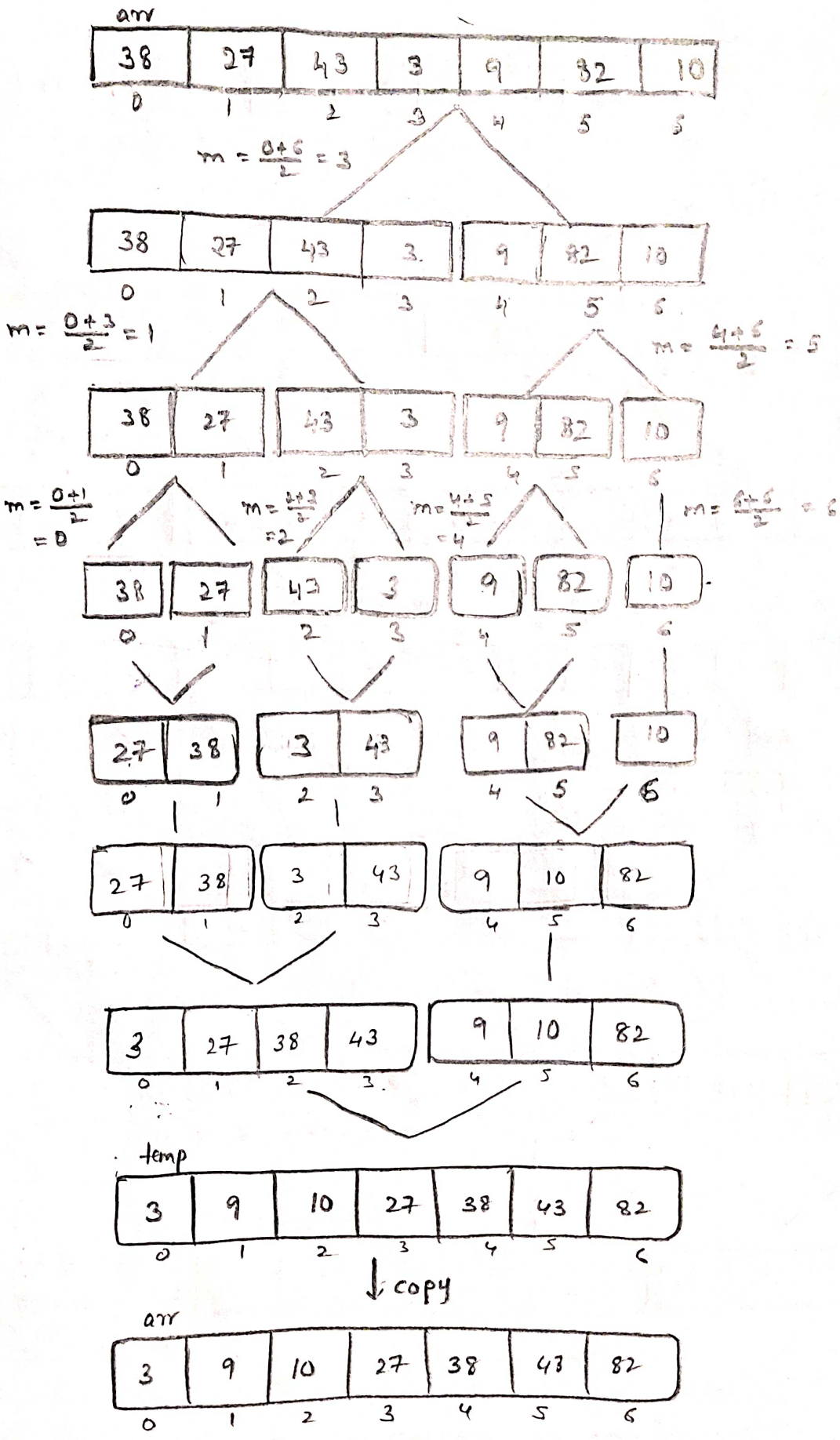
5	3	8	9	1
0	1	2	3	4

7	0	2	6	4
5	6	7	8	9

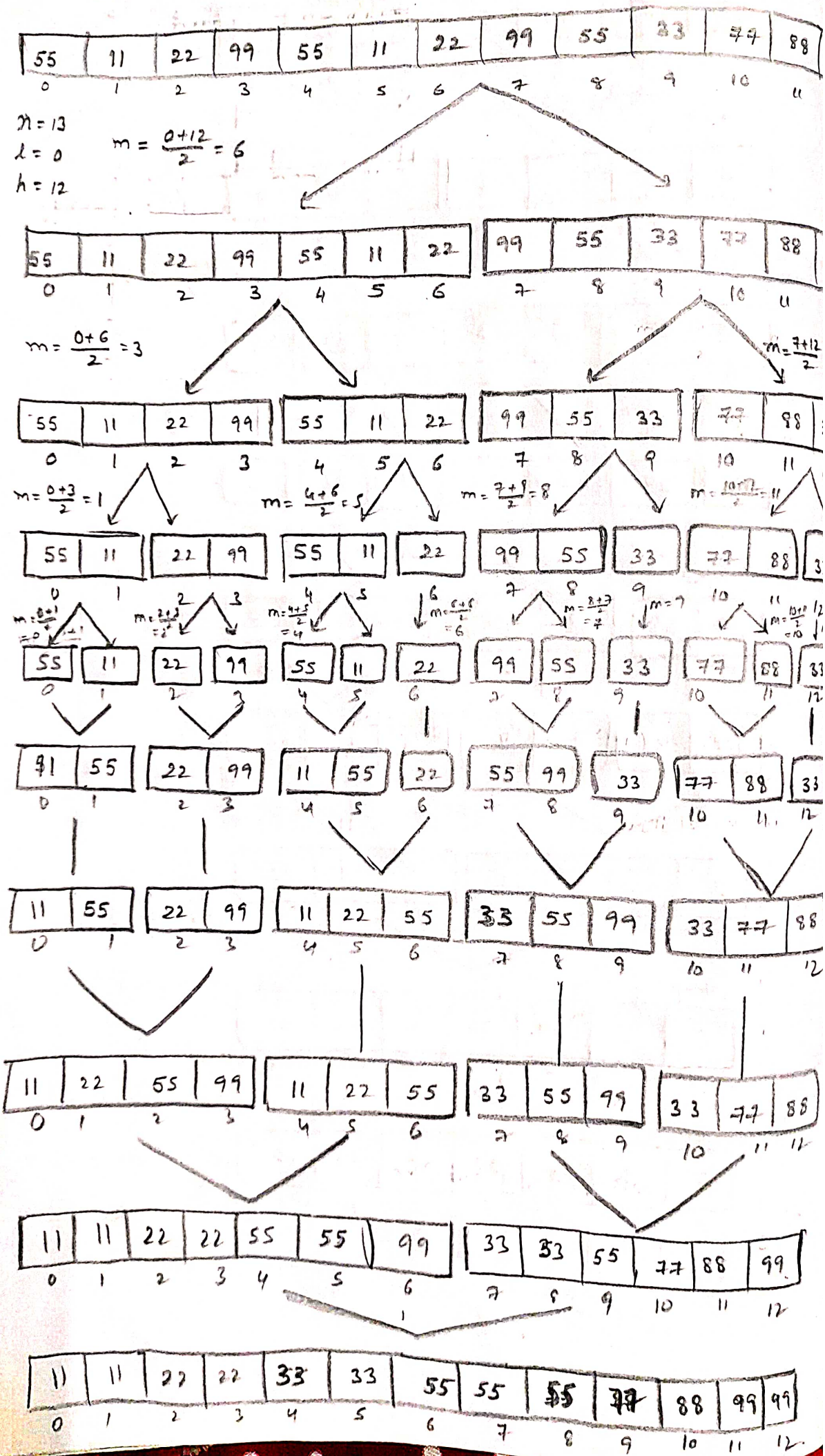
Eg.


arr
5 3 8 9 1 7 0 2 6 4
0 1 2 3 4 5 6 7 8 9
n = 10
l = 0
h = 9
m = 9+0/2 = 4
m = 0+4/2 = 2
m = 5+9/2 = 7
m = 0+2/2 = 1
m = 3+4/2 = 3
m = 5+7/2 = 6
m = 8+9/2 = 8
m = 0+1/2 = 0
m = 2+2/2 = 2
m = 3+3/2 = 3
m = 4+4/2 = 4
m = 5+6/2 = 5
m = 7+7/2 = 7
m = 8+8/2 = 8
m = 9+9/2 = 9
temp.
arr
copy

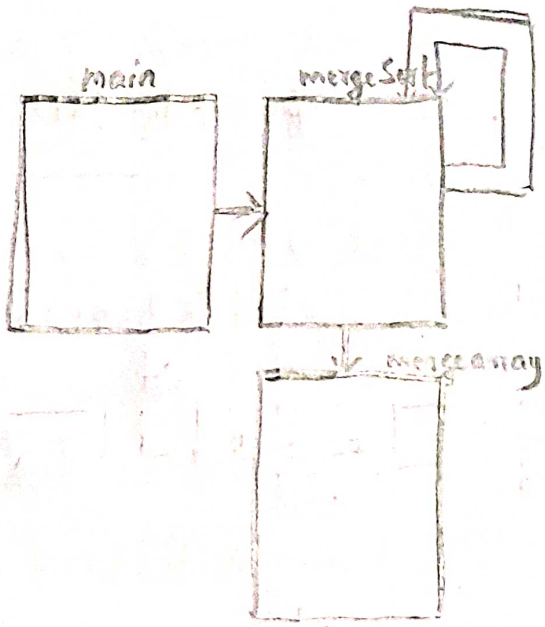
Eg: 38 27 43 3 9 82 10



Eg: 55 11 22 99 55 11 22 99 55 33 77 33



Implementation




```
#include <stdio.h>
void mergeSort (int,int);
void mergeArray (int,int, int, int);
int main()
{
    ____
    ____
    ____
    mergeSort (0,n-1)
    ret
    ____
    ____
}
void mergeSort (int l,h)
{
    int m = (l+h/2);
    mergeSort (l,m);        if (l<h) {
    mergeSort (m+1, h);
    mergeArray (l, m, m+1, h);
}
void mergeArray (int a, int b, int c, int d)
{
    int temp[100];
    int i, j, k=a;
    i = a;
    j = c;
    while (i<=b && j<=d)
    {
        if (a[i]<a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while (i<=b)
    {
        temp[k++] = a[i++];
    }
```

```
while(j<=d)
{
    temp[k++] = a[j++];
}
for(k=0; k<=d; k++)
    a[k] = temp[k];
}
```

→ Drawbacks of Merge Sort

→ Slower comparitive to other sorting algorithm for smaller tasks
→ It requires an additional memory space of $\theta(n)$ for the temporary array.
→ It goes through the whole procesn even if input array is sorted.

→ Bucket Sort

Bucket Sort or Bin sort is a sorting algorithm that works by distributing the elements of an array into a no of buckets.

Each bucket is then sorted individualy either using a different sorting ~~alog~~ algorithm or by recursively applying bucket sort algorithm.

Bucket sort is applied with elements that are uniformly distributed over a range and with the floating point values.

Algorithm

Step 1 : Start

Step 2 : Create n number of buckets or lists.

Step 3 : Do the following for every element, arr[i].

Step 3.1 : Insert the element arr[i] in bucket of [n * arr[i]]

Step 4 : Apply either insertion sort or bucket sort recursively on every individual bucket

Step 5 : Concatinate all the bucket elements to form a sorted list

Step 6 : Stop

→

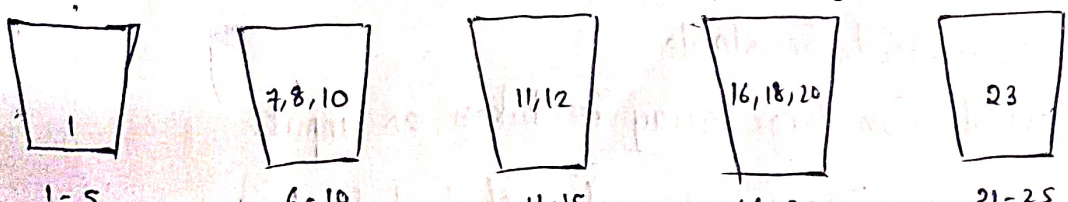
10	8	20	7	16	18	12	1	23	11
0	1	2	3	4	5	6	7	8	9

All elements are in range 1 - 23

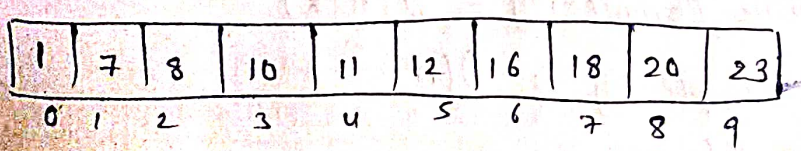
We can take any number of buckets, here we are taking 5 buckets each having range - 5.

(Scattering elements into buckets)

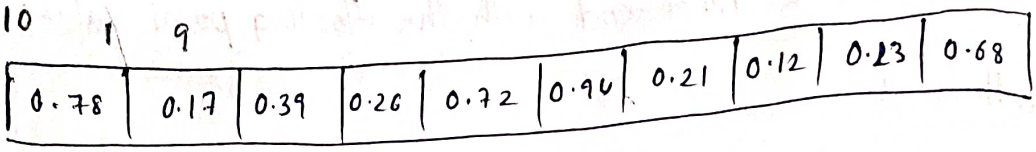
(Sorting in each bucket)

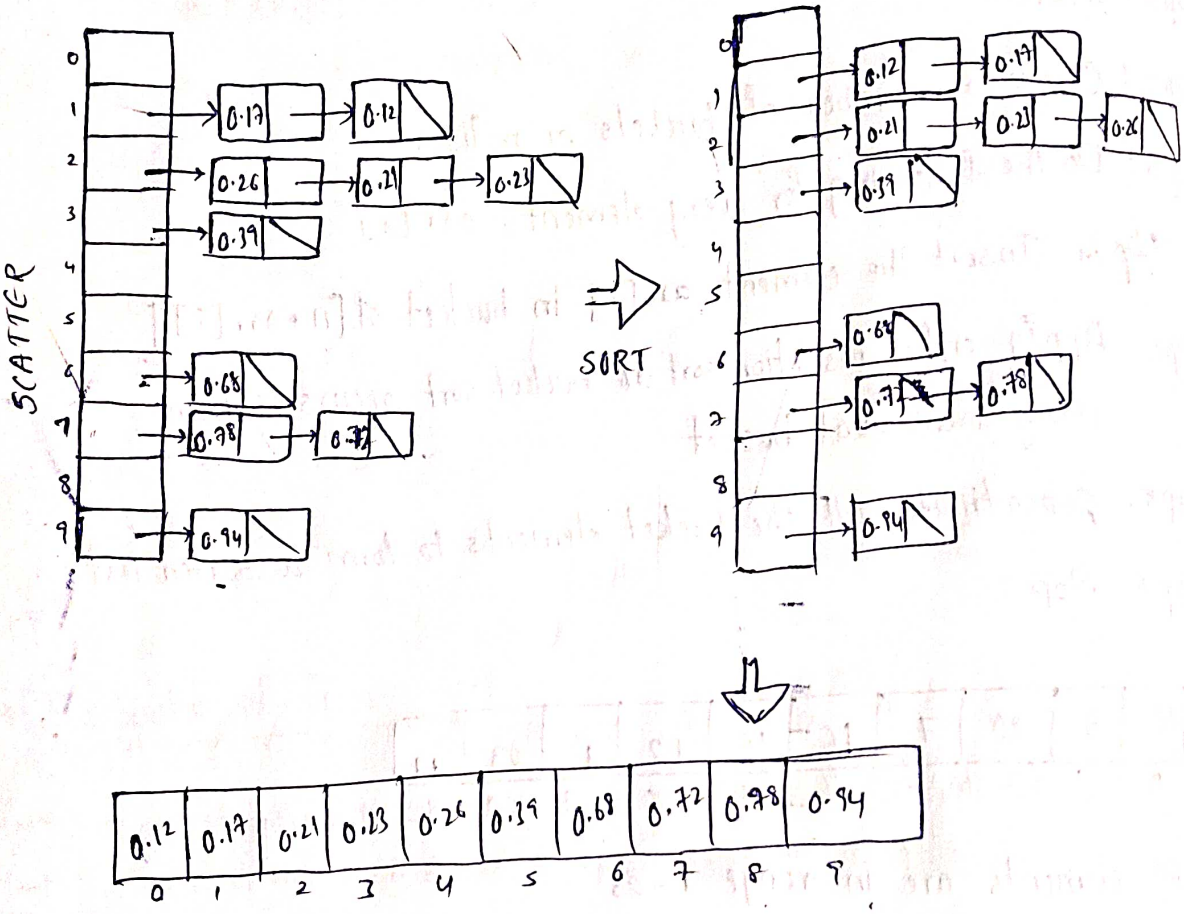


(Concatenation)



→ 0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68





Advantages

→ Reduce no of comparisions.

→ It is asymptotically fast, as the elements are uniformly sorted.

Disadvantages

→ may or maynot be stable

→ not useful when large array is taken as input.

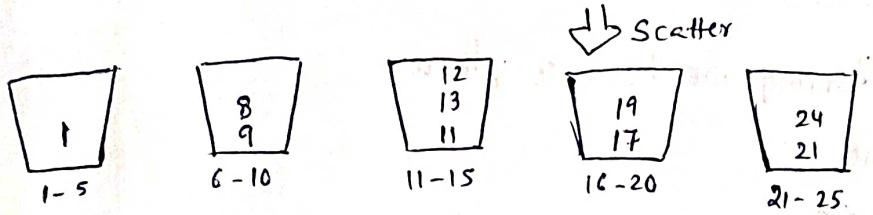
→ We require extra space to sort each bucket

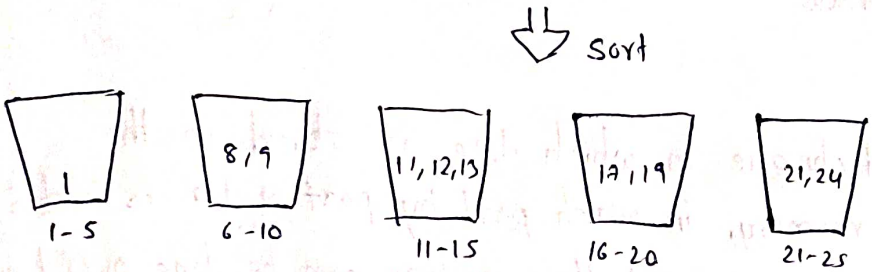
Best & Avg Time Complexity = $O(n+k)$

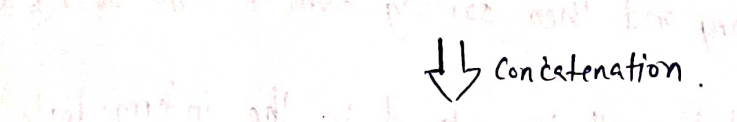
Worst case Time complexity = $O(n^2)$

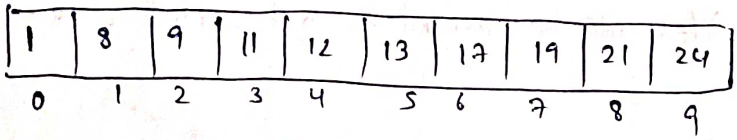
Space Complexity $O(n*k)$

→ 11 9 21 8 17 19 13 1 24 12



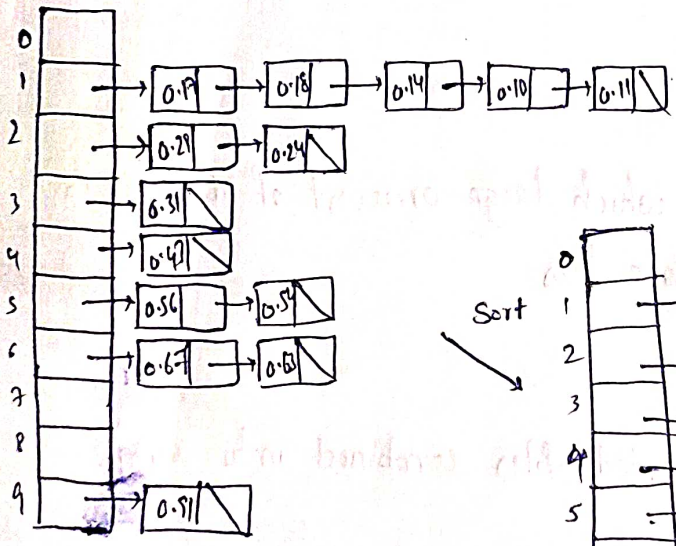


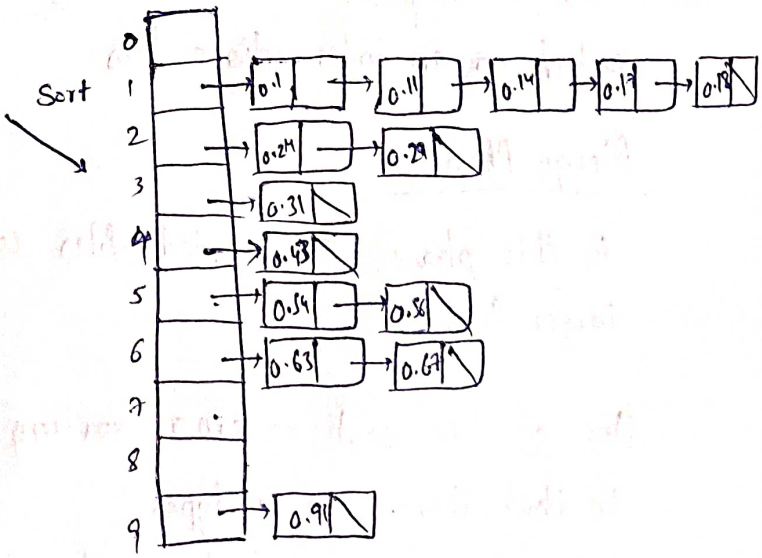




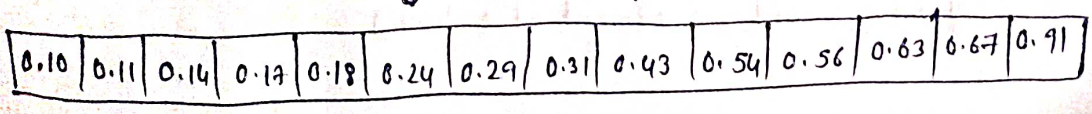
→ 0.17 0.31 0.29 0.56 0.24 0.18 0.14 0.91 0.10 0.67 0.54 0.63.
0.11 0.43

(scatter)





↓ Concatenation.



External Sorting

→ In external sorting, we have 2 phases.

1. Sorting phase
2. Merging phase

It is use

→ It is a technique in which data is stored in the secondary memory, in which part by part data is loaded into the main memory and then sorting can be done over there

Then, this sorted data will be stored in the intermediate files, finally these files will be merged to get sorted a sorted data.

Thus, by using external sorting technique, a huge amount of data can be sorted easily.

In case of external sorting, all the data cannot be accomidated on single memory. In this case, some amount of memory needs to be kept on memory such as hard disk, compact disk & memory chips.

→ ## Sorting Phase

This is the phase in which large amount of data ~~se~~ is sorted in an intermediate files

Merge Phase

In this phase, the sorted files combined into single larger file.

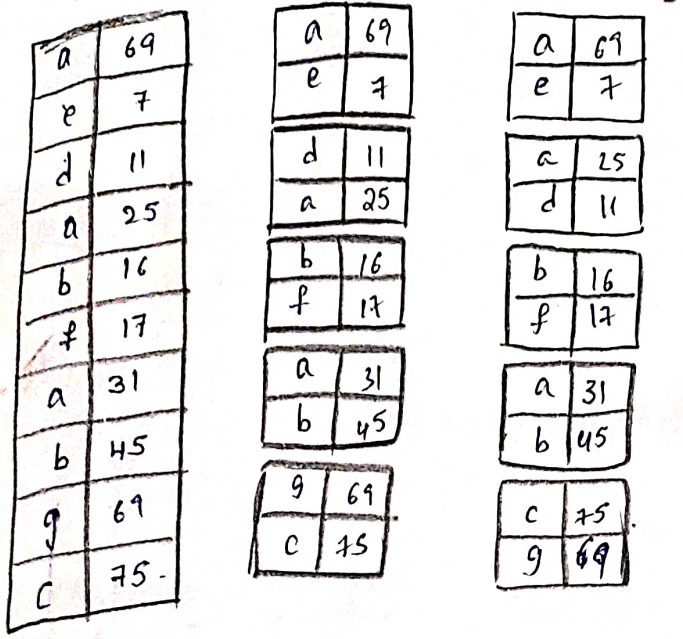
One of the best external sorting is external merge sort.

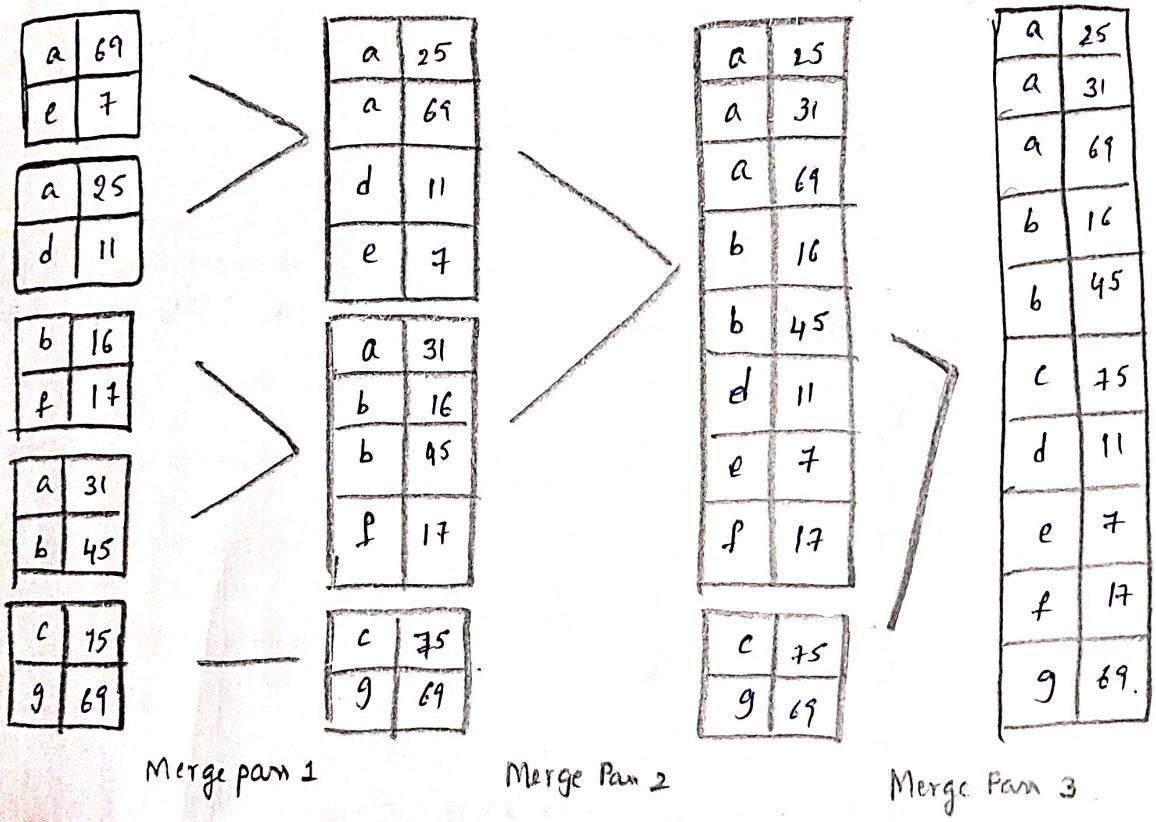
In that, there are 2 ~~types~~

2-way merge sort (taking 2 at a time)

k-way merge sort (taking k at a time).

Sorting





Merge pass 1 Merge Pass 2 Merge Pass 3

→ Algorithm Analysis

1. Mathematical Background.
2. Model - Computer
3. How to analyse.
4. Running time calculation.
5. 1. Time Complexity (by means of Asymptotic notations)
   2. Space Complexity

→ Asymptotic Notations.

1. Big-Oh - O (upper bound or worst case)
2. Big-Theta - Θ (Avg case or avg bound)
3. Big-Omega - Ω (lower bound or best case)

$3n^2 + n$

Algorithm: Sum of n numbers

Step 1 : Start	0
Step 2 : Read n	1
Step 3 : Sum = 0, i = 1	1
Step 4 : sum = sum + i	n
Step 5 : i = i + 1	n
Step 6 : check is i <= n, goto step 4	n+1
Step 7 : display sum	1
Step 8 : Stop	0

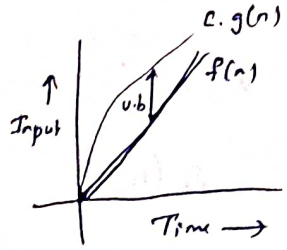
$f(n) = 3n + 4$

1. $f(n) = O(g(n))$ where c, n are constants iff $f(n) <= c\,g(n)$

$f(n) \leq c.g(n)$

$3n + 4 \leq 3n + 4n$

$3n + 4 \leq 7n$



for $n=1$ $7n \leq 7n$

$n=2$ $10 \leq 14$

$c \cdot g(n) = 7 \cdot n$.

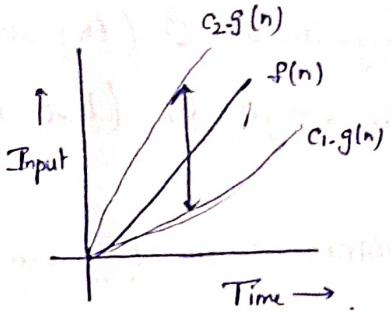
$\therefore$ it is $g(n) = n$

$\therefore$ Time complexity $= O(n)$

2.

$f(n) = \Theta(g(n))$ where c_1, c_2 are constants iff

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$



$2n + 3 \leq 3n + 4 \leq 3n + 4n$.

Checking

$i=1 \rightarrow \quad 5 \leq 7 \leq 7$

$i=2 \rightarrow \quad 7 \leq 10 \leq 14$

$g(n) = n$. Average : $\frac{n}{2}$

$\Rightarrow \quad \Theta(n/2) \rightarrow \Theta(n)$

3.

$f(n) = \Omega(g(n))$ where c, n_0 are constants iff $f(n) >= g(n), c$

$c \cdot g(n) \leq f(n)$

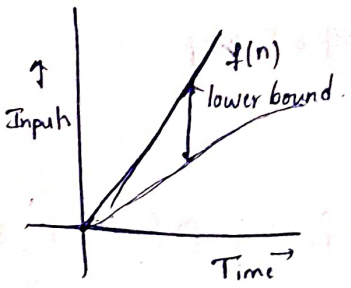
$2n+3 \leq 3n+4$

$n=1 \Rightarrow \quad 5 \leq 7$

$n=2 \Rightarrow \quad 7 \leq 10$.

$g(n) = \cdot\cdot$

$\Rightarrow \quad \Omega(n)$.



→ O(1) – constant rate

→ Time Complexity

→ Insertion Sort

```
for(i=1; i<n; i++){                          ——— 1 + n + (n-1)
    for(j=i; j>0; j--){                      ——— n-1 + (2+3+..+n-1) + (1+2+3..n-1)
        if(a[j] < a[j-1]){                   ——— 2(2+3+..+n-1)
            t = a[j];                        ——— 2(2+3+..+n-1)
            a[j] = a[j-1];                   ——— 2(2+3+..+n-1)
            a[j-1] = t;                      ——— 2(2+3+..+n-1)
        }
    }
}
```

$1 + 3n + 10\{2+3+\dots+n-1\} - 1$

$3n + 10 \times \frac{n(n-1)}{2}$

$= 5n^2 - 2n - 10$

$= O(n^2)$

→ For Insertion sort,

Worst Case : $O(n^2)$

Avg Case : $\theta(n^2)$

Best Case : $O(n)$

Space Complexity : $O(1)$

→ Merge Sort

$T(n) = T(n/2) + T(n/2) + n \longrightarrow$ (Divide & conquer algorithm) $\rightarrow ①$

Substitute $T(n/2)$ in eq ①

$T(n/2) = T(n/4) + T(n/4) + \frac{n}{2} \longrightarrow ②$

Substitute eq ② in ①

$T(n) = 2\left(2T(n/2^2) + \frac{n}{2}\right) + n$

$= 2^2 T(n/2^2) + 2n \longrightarrow ③$

Sub $T(n/4)$ in eq ①

$$T(n/4) = T(n/8) + T(n/8) + n/4.$$

$$T\left(\frac{n}{4}\right) = 2T(n/8) + \frac{n}{4} \longrightarrow ④$$

Substitute in eq ②.

$$T(n) = 2^2\left[2T(n/2^2) + n/4\right] + 2n$$

$$= 2^3 T\left(n/2^3\right) + 3n.$$

For i^{th} form

$$\boxed{T(n) = 2^i T(n/2^i) + in}$$

$$= n.T(1) + \log_2^n + n$$

$$= n\log_2^n + n.T(1)$$

$$= O(n\log_2^n)$$

$\frac{N}{2^i} = 1$

$N = 2^i$.

$\log_2^N = i\log_2^2$

$i = \log_2^N$

Space Complexity = $O(n)$

Best Case Time Complexity : $O(n\log n)$

Avg Case Time Complexity : $O(n\log n)$

Worst Case Time Complexity : $O(n\log n)$

Merge Sort is stable

Quick Sort

Time complexity for entire recursive ~~quick~~ tree is equal to no of levels into time taken for 1 lvl.

~~No of levels $= \log_2^n$ $= T(n/2) + T(n/8)$~~

Best Case : $O(n\log n)$ Worst Case : $O(n^2)$

Avg Case : $O(n\log n)$

No of levels = $T(n/2) + T(n/2)$

$= T(\frac{n}{4}) + T(\frac{n}{4}) + T(\frac{n}{4}) + T(n/4)$

$\vdots$

$= 2^i T(n/2^i)$

Total no of levels = $\log_2^n$

Space Complexity : $O(\log n)$

Best/Co Worst Case Time Complexity

When array elements are in descending order and is to be sorted in ascending order, it takes $O(n^2)$ as time complexity.

Quick Sort is not stable.

T/D.

For stack

→ For shell sort,

Best Case Time complexity: $O(n)$

Avg Case.: $O(n^2)$

Worst.: $O(n^2)$ Space Complexity: $O(1)$.

→ Bucket sort

Best: $O(n+k)$

Avg: $O(n+k)$

Worst: $O(n^2)$

Space Complexity: $O(n)$.

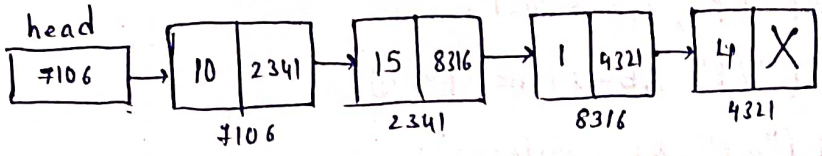
CO-2

Dynamic memory implementation of Linear Data Structures

1. Single Linked list: Creation, Insertion, Deletion, Display
2. Double Linked list: Creation, Insertion, Deletion, Display
3. Circular Linked list: Creation, Insertion, Deletion, Display.
4. Stack Using Single linked list
5. Queue using Single linked list
6. Infix to Postfix Conversion
7. Postfix evalution
8. Balancing Symbols.
9. Types of Queues
   1. Circular Queue
   2. DQueue

1. Single Linked list:

It is a linear data structure. It is collection of nodes. The node contains two parts.: data & link to the next node.



Operations:

Create()	Search()
InsertAtBegin()	length()
InsertAtMiddle()	count()
~~inser~~append()	sort()
Deletion()	max()
Display()	min().

→ Define a structure Bike with members model, make, cost, tank capacity & milege. Read & display bike details using DMA

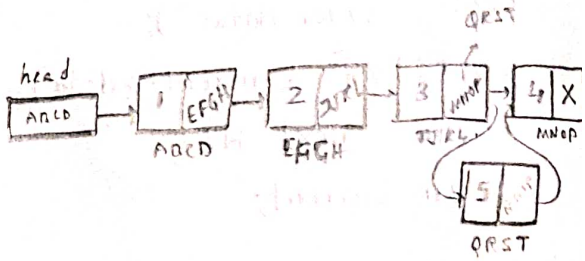
```
#include <stdio.h>
#include <stdlib.h>
struct Bike {
    char model[30];
    char make[20];
    float cost;
    float tankcapacity;
    float mileage;
};
int main()
{
    struct Bike *b;
    printf("Enter bike details: ");
    gets(b->model);
    gets(b->make);
    b = (struct Bike*)malloc(sizeof(struct Bike));
    printf("Enter bike details: ");
    gets(b->model);
    gets(b->make);
    scanf("%f", &b->cost);
    scanf("%f", &b->tankcapacity);
    scanf("%f", &b->mileage);
    printf("Bike details:\n");
    printf("%s\n%s\n%f\n%f\n%f\n", b->model, b->make,
           b->cost, b->tankcapacity, b->mileage);
    return 0;
}
```

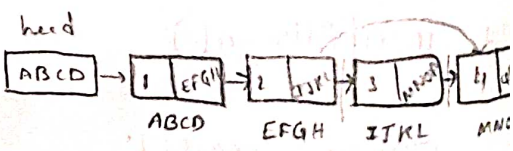
Implementation:

> **Self referencial Structure**
>
> The member of structure pointing to another structure of same type

```
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct node *next;
};
struct node *head, *current, *p, *q;
struct node*create()
{
    current = (struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d", &current->data);
    current->next=NULL;
    return current;
}
void append()
{
    current = create();
    if(head==NULL)
        head = current;
    else {
        p=head;
        while(p->next!=NULL)
            p=p->next;
        p->next=current;
    }
}
void insertAtBegin()
{
    current=create();
    if(head==NULL)
        head=current;
    else{
        current->next=NULL, head;
        head=current;
    }
}
```

```
int length()
{
    int l=0;
    p=head;
    while(p!=NULL){
        p=p->next;
        l++;
    }
    return l;
}
void insertAtMiddle()
{
    int k,i=2;
    current=create();
    printf("enter position:");
    scanf("%d",&k);
    if(k<(length(1))
    {
        p=head;
        while(i<k)
        {
            p=p->next
            i++;
        }
        current->next=p->next;
        p->next=current;
    }
    else
        printf("Insertion not possible");
}
void deletion()
{
    int k,i=2;
    ecarprintf("enter position:");
    scanf("%d",&k);
    if(k>(length(1)))
        printf("Deletion not possible");
    else if(k==1){
        p=head;
        head=p->next;
        p->next=NULL;
        free(p);}
```

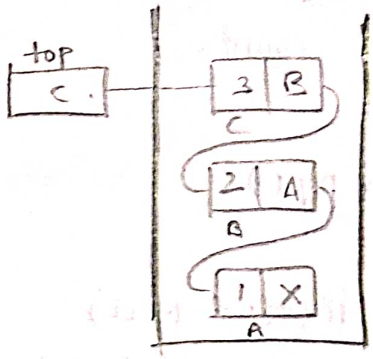




```
else {
    p = head;
    while (i<k){
        p = p -> next;
        i++;
    }
    q = p -> next;
    p -> next = q -> next;
    q -> next = NULL;
    free (q)
}
}.
```

4. Stack Using Linked list

→ Always insertion at begin take place & deletion at begin take place as operations on stack takes place at one end.



→ As we are using linked list for stack implementation, stack overflow is not possible.

→ If top == NULL, then stack is empty.

→ Implementation

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node* next;
};
struct node *current, *top = NULL, *p, *q;
int count;
struct node *create()
```

```
struct node * create()
{
    current = (struct node *)malloc (sizeof (struct node));
    printf("enter data: ");
    scanf("%d",&current->data);
    current->next = NULL;
    return current;
}
void push()   (Insert At Begin)
{
    current= create();
    if(top== NULL)
        top= current;
    else {
        current->next =top;
        top= current;
    }
    count++;
}
void pop()    (deleteAtBegin
{
    if (top== NULL)
        printf ("stack is empty");
    else {
        p =top;
        top= p->next;
        p->next= NULL;
        free (p);
        count--;
    }
}
void peak()
{
    if (top== NULL)
        printf (" stack is empty");
    else {
        printf(" %d", top->data);
}
```

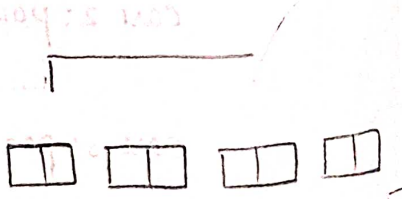
```
void display()
{
    if (top == NULL)
        printf("stack is empty");
    else {
        p = top;
        while (p != NULL) {
            printf("%d ", p->data);
            p = p->next;
        }
    }
}
int main()
{
    int ch;
    printf(" 1. push \n 2. pop \n 3. peak \n 4. display \n 5. exit \n
     count \n 6. exit \n");
    while (1) {
        printf("enter choice :");
        scanf("%d", &ch);
        switch (ch) {
            case 1: push();
                    break;
            case 2: pop();
                    break;
            case 3: peak();
                    break;
            case 4: display();
                    break;
            case 5: printf(" no of elements: %d ", count);
                    break;
            case 6: exit(0);
            default: printf(" invalid !! ");
        }
    }
    return 0;
}
```

5. Linked list ~~list~~ implemention of Queue

→ Queue is a linear data structure which follows first in first out principle (FIFO).

Implementation

```
#include <stdio.h>
struct node {
    int data;
    struct node* next;
};
struct node *current, *p, *f=NULL, *r=NULL;
int count=0;
struct node* create()
{
    current = (struct node *)malloc (sizeof(struct node));
    printf(" enter data:");
    scanf("%d", &current->data);
    current -> next= NULL;
    return current;
}
void enqueue()   (insertAtEnd)
{
    current = create();
    if( f==NULL && r==NULL) {
        rear = current;
        front = rear;
    }
    else {
        current->next = rear;  rear -> next = current;
        rear = current;
    }
    count++;
    printf(" enqueued succesfully!");
}
```



```
void dequeue()   (Delete At Begin)
{
    if(f == NULL)
        printf("Queue is empty!");
    else {if(front->next!=NULL) {
        p = front;
        front = p->next;
        p->next = NULL;
        free(p);
    }  count--; }
    }
    printf("dequeued succesfully!");
}
void display()
{
    if(f==NULL)
        printf("Queue is empty!");
    else {
        p = front;
        while(p!=NULL){
            printf("%d", p->data);
            p = p->next;
        }
    }
}
void frontele()
{
    if(f==NULL)
        printf("Queue is empty!");
    else
        printf("front element: %d", front->data);
}
```

```
else {
    free(f);
    f = NULL;
    r = NULL;
    count--;
}
```

```
if(f==NULL)
    r = NULL;
```

```
void rearele()
{
    if(r==NULL)
        printf("Queue is empty!");
    else
        printf("rear element: %d", rear->data);
}
int main()
{
    int ch;
    printf("1. enqueue\n 2. dequeue\n 3. display\n 4. frontele\n
            5. rearele\n 6. count\n 7. exit\n");
    while(1) {
        printf("\nenter choice:");
        scanf("%d",&ch);
        switch(ch) {
            if(ch<=7).
            case 1: enqueue();
                    break;
            case 2: dequeue();
                    break;
            case 3: display();
                    break;
            case 4: frontele();
                    break;
            case 5: rearele();
                    break;
            case 6: printf("no of elements in queue: %d", count);
                    break;
            case 7: exit(0);
            default: printf("invalid!");
    }
    return 0;
}
```

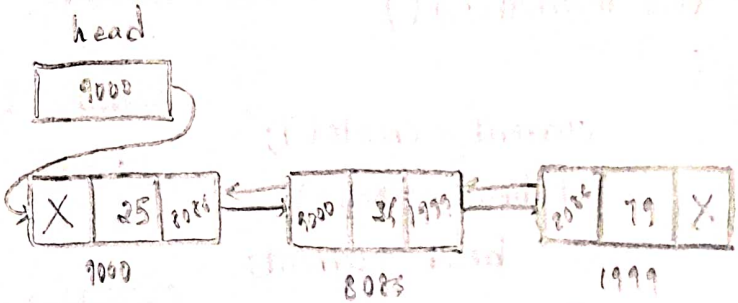
→ Doubly Linked List

→ It's a linear data structure in which it is collection of nodes.

→ Each node has 3 fields

1. Data part
2. Link to previous node
3. Link to next node.

→ Operations

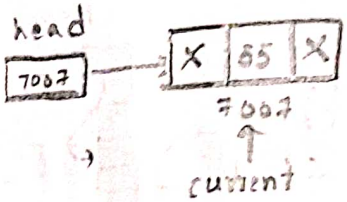


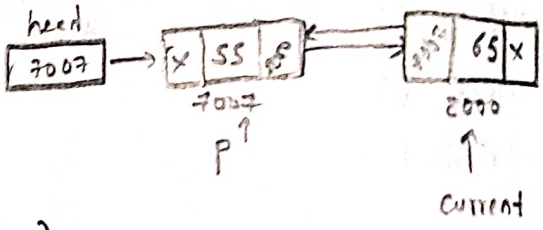
1. Create(), ~~insert()~~
2. Insert
   - insertAtBegin()
   - insertAtMiddle()
   - insertAtEnd()
3. Delete
   - deleteAtBegin()
   - deleteAtMiddle()
   - deleteAtEnd()
4. Display
5. Search
6. Sort
7. Reverse
8. Update

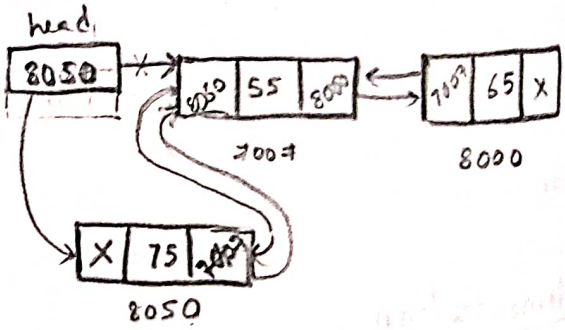
→ Implementation

```
#include<stdio.h>
 struct node {
      int data;
      struct node* previous;
      struct node*next;
 }*head=NULL,*current,*p,*q;
 int count=0;
```

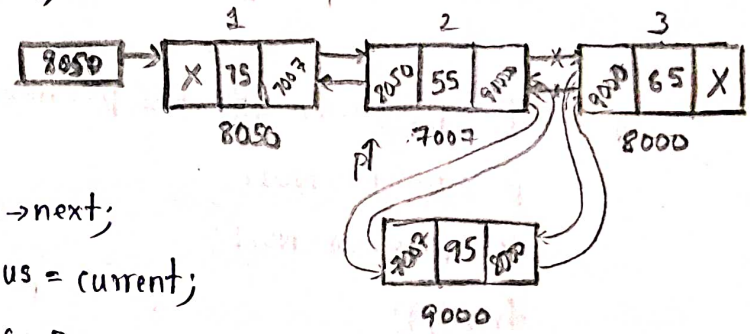
```
struct node * create()
{
        current = (struct node*) malloc (sizeof (struct node));
        printf ("enter data:");
        scanf ("%d", &current -> data);
        current -> previous = NULL;
        current -> next = NULL;
        return current;
}
void insertAtEnd()
{
        current = create();
        if(head == NULL){
            head = current;
        }
        else {
            p=head;
            while(p->next != NULL)
                p = p -> next;
            p ->next = current;
            current->previous = p;
        }
        count++;
}
void insertAtBegin()
{
        current = create();
        if(head == NULL)
            head = current;
        else {
            current->next = head;
            head -> previous = current;
            head = current;
        }
        count++;
}
```







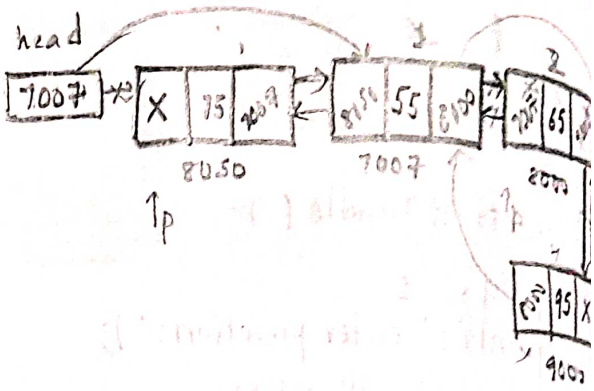
```
void insert()
int length() void display()
{
    if(head == NULL)
        printf("Doubly Linked list is empty!");
    else{
        p = head;
        while(p != NULL){
            printf("%d ", p->data);
            p = p->next;
        }
    }
}
void insertAtMiddle()
{   int k,i=1;
    printf("enter position:");
    scanf("%d", &k);
    if(k > count)
        printf("insertion not possible!");
    else{
        current = create();
        p = head;
        while(i<k)
            p = p->next;
        current->next = p->next;
        (p->next)->previous = current;
        current->previous = p;
        count++;
        p->next = current;
        count++;
    }
}
```



```
{
    int k;
    printf("enter position:");
    scanf("%d", &k);
    if (k > count)
        printf("deletion not possible!");
    else if (k == 1) {
        p = head;
        head = p->next;
        (p->next)->previous = NULL;
        p->next = NULL;
        free(p);
        count--;
    }
    else {
        p = head;
        int i = 2;
        while(i <= k){
            p = p->next;
            i++;
        }
        (p->next)->previous = p->previous;

        (p->previous)->next = p->next;
        p->next = NULL;
        p->previous = NULL;
        free(p);
        count--;
    }
} */

void deleteAtBegin()
{
    int k;
    if (head == NULL)
        printf("deletion not possible");
    else if (head->next == NULL) {
        free(head);
        head = NULL;
        count--;
```



```
    else {
        p = head;
        head = p->next;
        p->next->previous = NULL;
        p->next = NULL;
        free(p);
        count--;
    }
}
void deleteAtMiddle()
{
    if(head == NULL)
        printf("List is empty!");
    else if(head->next == NULL){
        free(head);
        head = NULL
        count--;
    }
    else {
        printf("enter position:");
        scanf("%d", &loc);
    else{ p = head;
        int i = 2;
        while(i <= loc) {
            p = p->next;
            i++;
        }
        (p->next)->previous = p->previous;
        (p->previous)->next = p->next;
        p->next = NULL;
        p->previous = NULL;
        free(p);
        count--;
    }
}
```

```
if(loc > count)
    printf("deletion not possible");
```

```
Void deleteAtEnd()
{
    if(head==NULL)
        printf("List is empty");
    else if(head->next==NULL){
        free(head);
        head=NULL;
        count--;
    }
    else{
        p=head;
        while(p->next!=NULL)
            p=p->next;
        (p->previous)->next=NULL;
        p->previous=NULL;
        free(p);
        count--;
    }
}
void search()
{
    if(head==NULL)
        printf("List is empty!");
    else{
        printf("enter element to search:");
        int k,c=1;
        scanf("%d",&k);
        p=head;
        while(p!=NULL)
        {
            if(p->data==key){
                printf("element found at %d node",c);
                break;
            }
            c++;
            p=p->next;
        }
```

```
if ( c == count+1 )
    printf ("element not found ");
}
```

→ Circular Linked List

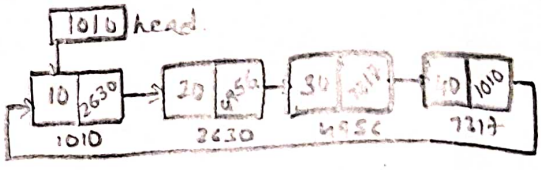
→ Circular Linked list is a linear data structure in which the last node is connected to first node.

It is of two types

- Circular Singly Linked list
- Circular Doubly Linked list

→ Circular Singly Linked list

- In which last node's next is connected to first node.



→ Differences with SLL

→ Any node can be the first node

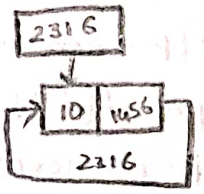
→ Last node is connected to first node.

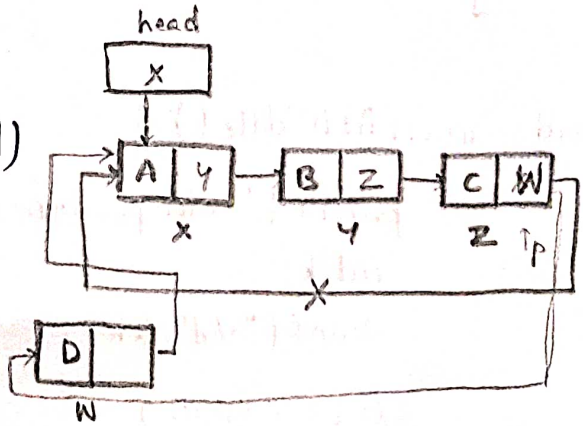
→ Operations

create()
insert()
delete()
display()
search()
sort()
length()
update()

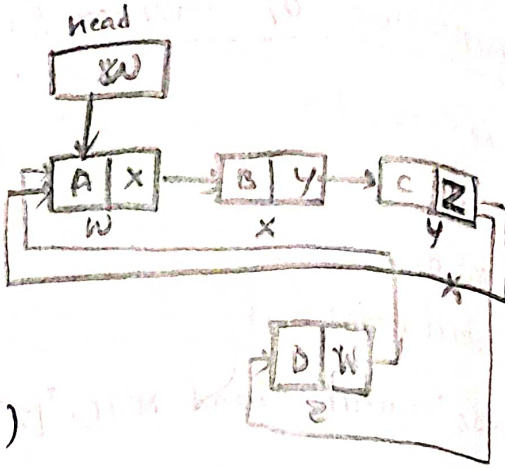
Implementation of Circular Single Linked List

```
#include <stdio.h>
struct node {
        int data;
        struct node * next;
} *current , *head = NULL , *p, *q;
int count = 0;
struct node * create ( ) {
        current = (struct node *) malloc (sizeof (struct node));
        printf ("Enter data : ");
        scanf ("%d", &current -> data);
        current -> next = NULL;
        return current;
}
void insertAtBegin( ) {
        current = create();
        if(head == current) {
                head = current;
                current -> next = head;
        }
        else {
                p = head;
                while (p -> next != head)
                    p = p -> next;
                p -> next = current;
                current -> next = head;
                head = current;
        }
        count++;
}
```





```
void insertAtHead() {
        current = create();
        if (head == NULL)
                head = current;
        else {
                p = head;
                while (p->next != head)
                        p = p -> next;
                p -> next = current;
                current -> next = head;
        }
        count++;
}
```

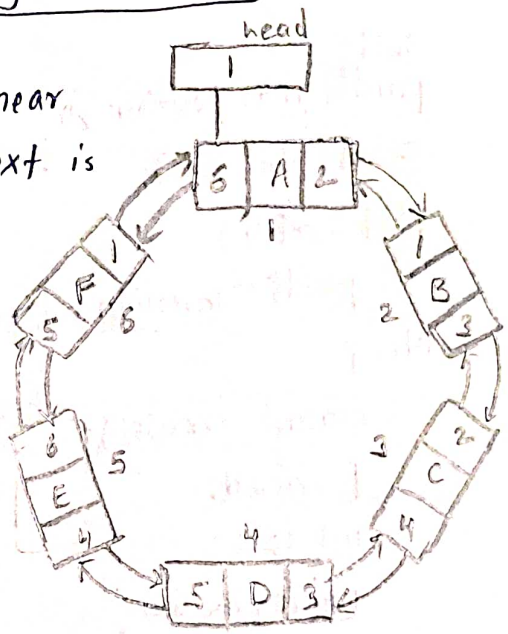



```
void display() {
        p = head;
        if (head == NULL)
                printf("List is empty!");
        else {
                while (p -> next != head){
                        printf("%d ", p -> data);
                        p = p -> next;
                }
                printf("%d", p -> data);
        }
}
void insertAtMiddle() {
        printf("enter position: ");
        int k;
        scanf("%d", &k);
        if (k > count)
                printf("insertion not possible!");
        else {
                p = head;
                int i=1;
                while (i < k-2) {
                        p = p -> next;
                        i++;
                }
```

```
            current->next=p->next;
            p->next = current;
            count++;
        }
    }
void deleteAtBegin() {
    if(head==NULL)
        printf("List is empty");
    else {
        p = head; q = head.
        while(p->next != head) {
            p = p->next;
        p->next = head->next;
        head->next = NULL          ⟶ head = hedd->next;
        free(q);
        head = count--;
    }
}
void deleteAtEnd() {
    if(head == NULL)
        printf("List is empty");
    else {
        p = head; q=head;
        while(p->next->next != head)
            p = p->next;
        q = p->next;
        p->next = head;
        q->next = NULL;
        free(q);
        count--;
    }
}
```

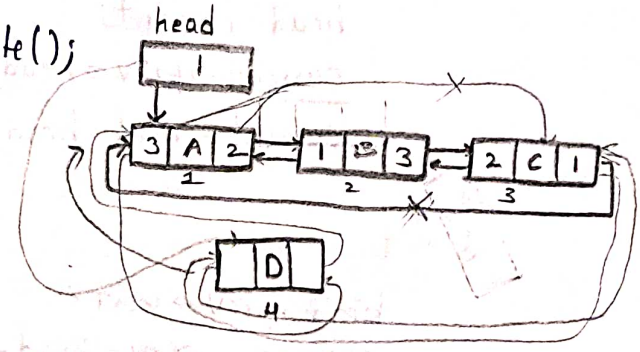
```
void deleteAtMiddle() {
    int k;
    printf("enter position: ");
    scanf("%d", &k);
    if(k > count)
        printf("List is empty");
    else {
        int i=2;
        p=head;
        while(i<k) {
            p=p->next;
            i++;
        }
        q=p->next;
        p->next=q->next;
        q->next=NULL;
        free(q);
        count--;
    }
}
int main() {
{
```

Implementation of Circular Doubly Linked List

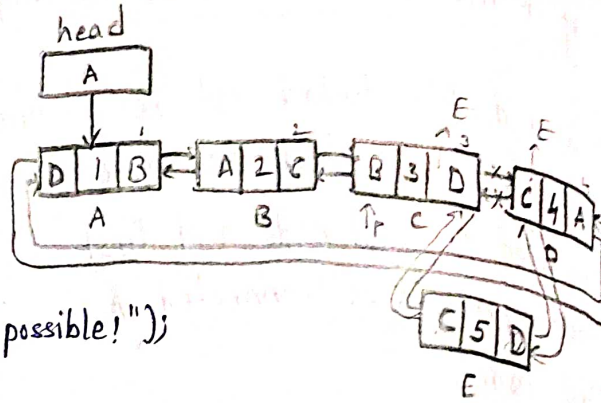


Circular doubly linked list is a linear data structure in which last node's next is connected to first node and first node's previous is connected to last node.

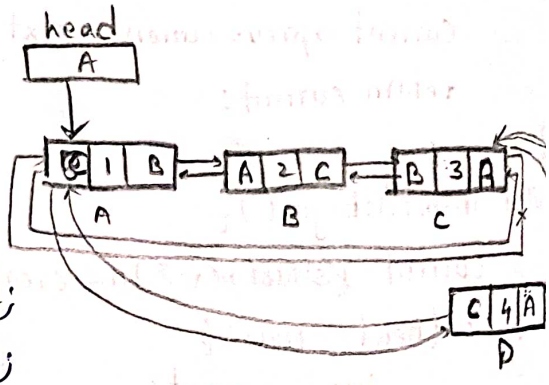
```
#include<stdio.h>
struct node {
        int data;
        struct node *prev;
        struct node *next;
} *head = NULL, *current, *p, *q;
int count = 0;
struct node** create() {
        current = (struct node*) malloc (sizeof(struct node));
        printf("enter data: ");
        scanf("%d", &current->data);
        current->prev = current->next = NULL;
        return current;
}
void insertAtBegin(){
        current = (struct node*) create();
        if(head==NULL){
                head = current;
                current = head;
                current->prev = head;
                current->next = head;
        }
        else {
                current->next = head;
                current->prev = head->prev;
                head->prev->next = current;
                head->prev = current;
                head = current;
        } count++; }
```



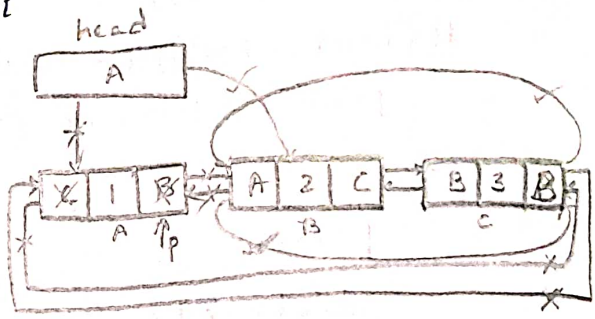
```
void insertAtMiddle()
{
    int k;
    printf("enter position: ");
    scanf("%d", &k);
    if(k> count)
        printf("insertion not possible!");
    else {
        current= create();
        p=head;
        int i=2;
        while(i < k) {
            p=p->next;
            i++;
        }
        current->next=p->next;
        p->next->prev = current;
        current->prev = p;
        p->next = current;
        count++;
    }
}
```



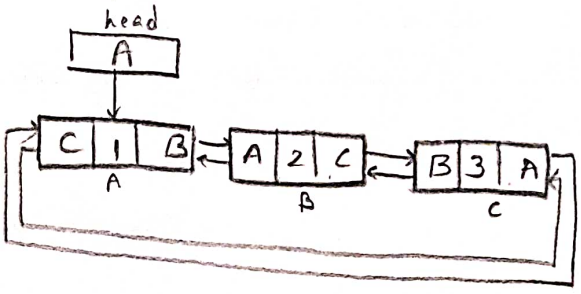

```
void insertAtEnd()
{
    if(head ==NULL){
        head = current;
        current->prev = head;
        current->next = head;
    }
    else {
        head->prev->next =
        current->prev = head->prev
        head->prev->next = current
        current->next = head;
        head->prev = current;
    }
    count++;
}
```



```
void deleteAtBegin() {
{
    if (head == NULL)
        printf("list is empty");
    else if (head -> next == head) {
        head -> prev = NULL;
        head -> next = NULL;
        free(head);
    }
    else {
        int i = 1;
        p = head;
        p -> prev -> next = p -> next;
        p -> next -> prev = p -> prev;
        head = p -> next;
        p -> prev = p -> next = NULL;
        free(p);
    }
}
```




```
void deleteAtEnd()
{
    if(head == NULL)
        printf("list is empty!");
    else if (head -> next == head) {
        head -> prev = head -> next = NULL;
        free(head);
    }
    else {
        p = head -> prev;
        p -> prev -> next = head;
        head -> prev = p -> prev;
        p -> prev = p -> next = NULL;
        free(p);
    }
}
```



```
void deleteAtMiddle()
{
    int k, i=1;
    printf("enter position:");
    scanf("%d", &k);
    if (head==NULL || k>count)
        printf("deletion not possible");
    else {
        p=head;
        while(i<k) {
            p=p->next;
            i++;
        }
        (p->next)->prev = p->prev;
        p->prev->next = p->next;
        p->next = NULL;
        p->prev = NULL;
        free(p);
        count--;
    }
}
```

Application of Stacks Data Structure

1. Infix to Postfix Expression Conversion.
2. Postfix expression Evalution
3. Balancing brackets.
4. string reversal
5. Memory managment
6. Recursive function calls.
7. Lexical analysis.
8. Syntax Parsing.
9. Backtracking Algorithms.
10. Undo and Redo operations

1. Infix to Postfix Expression Conversion

Expression: It is combination of operands and operators

Infix expression:

<operand> <operator> <operand>

→ If operator is in b/w the operands then that expression is called Infix expression.

Eg: a+b + b+c

Postfix expression:

→ If the operator is succeeded by the operands then it is called Postfix expression.

<operand> <operand> <operator>

Eg: a+b ⇒ ab+

Algorithm to convert Infix to Postfix expression

Step 1 : Start

Step 2 : Scan the given input expression (Infix) from left to right

Step 3 : If the symbol is operand write it into postfix notation, if it is operator push it on to stack.

Step 4 : If the symbol is '(' push it on to the stack.

Step 5 : Always ensure that the operator which is at the top of the stack with highest priority.

Step 6 : If the incoming operator is having highest priority than the operator at the top of the stack then push the incoming operator on to the top of the stack.

Step 7 : If the incoming operator is having less (or equal) priority than the operator at the top of the stack then pop that operator and push it on to the stack.

Step 8 : If incoming operator is having equal priority with operator at the top of the stack, then pop the top of the stack & push the incoming operator on to the stack.

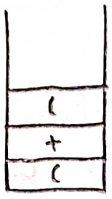
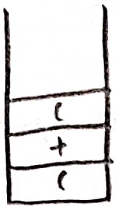
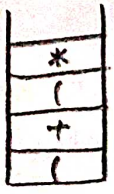
Step 9 : If the scanned operator is closed paranthesis ')', then pop all the operators in the stack until the open paranthesis and add these operators into postfix notation.

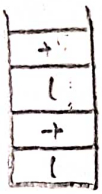
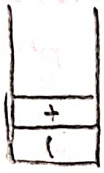
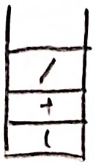
Step 10 : Repeat Step 2 to Step 9 until stack becomes empty

Step 11 : Display the postfix notation

Step 12 : Stop.

→ A + (B*C+D)/E

S.NO	symbol	Stack	Postfix Notation	Description
1	(		-	'(' is pushed into stack
2.	A		A	A is written in postfix notation
3.	+		A	'(' is considered as neglicible and don't affect incoming operator. + is pushed into stack.
4.	(		A	(is pushed into stack.
5.	B		AB	B is written into postfix notation.
6.	*		AB	* is pushed into stack.

S.no	symbol	stack	Postfix notation	Description
7	C		ABC	C is written into postfix notation.
8	+		ABC*	* is popped out from stack and added to postfix notation and + is ~~added~~ pushed to stack.
9	D		ABC*D	D is written into postfix notation.
10	)		ABC*D+	popped+ and written in postfix notation
11	/		ABC*D+	push / on to the stack
12	E		ABC*D+E	E is written in postfix notation.
13	)		ABC*D+E/+	/ and + are popped out and written in postfix notati

The postfix notation obtained ABC*D+E/+

→ A+B*C/D

S.NO	symbol	stack	Postfix Notation	Description
1	(	(	-	(is pushed onto the stack
2.	A	(	A	A is written into postfix notation
3.	+	(+	A	+ is pushed onto stack
4	B	(+	AB	B is written into postfix notation
5.	*	(+*	AB	* is pushed onto stack
6.	C	(+*	ABC	C is written into postfix notation.
7.	/	(+/	ABC*	* is popped out & / is pushed onto stack
8.	D	(+/	ABC*D	D is written into postfix
9.	)	-	~~ABCD/*+~~ ABC*D/+	+ & / are popped out

Postfix notation: ABC*D/+

→ A+B*C/D - K+L^U

S.no	symbol	stack	Postfix Notation	Description
1	(	(	-	(is written onto ~~postfix notation~~ stack.
2.	A	(A	A	A is written onto stack.
3.	+	(+	A	+ is pushed onto stack
4.	B	(+	AB	B is written into postfix notation

Sno	symbol	Stack	Postfix notation	Description
5.	*	(+*	AB	* is pushed onto
6.	C	(+*	ABC	C is written into postfix notation.
7.	/	(+/	ABC*	* is popped out and / is pushed onto st
8.	D	(+/	ABC*D	D is written into pos notation.
9.	−	(−	ABC*D/+	+ & / are popped ou and − is pushed on stack.
10.	K	(−	ABC*D/+K	K is written into postfix notation.
11.	+	(+	ABC*D/+K−	− is popped out & + is pushed into stack.
12.	L	(+	ABC*D/+K-L	L is written into postfix notation
13.	^	(+^	ABC*D/+K-L	+ & are popped and ^ is pushed o stack.
14	U	(+^	ABC*D/+K-LU	^ is popped out U is written onto postfix notation

Postfix Expression : ABC*D/+K-LU^

15	)	−	ABC*D/+K-LU^+	^, +, − are popped out.

Final Postfix Notation : ABC*D/+K-LU^+

→ $(300 * 23) * (43 - 21) / (84 + 7)$

→ $(a * (b + c)) + (b / d) * a$

S.no	symbol	stack	Postfix Notation	Description
1.	(	(	-	(is pushed onto stack
2.	(	((	-	(is pushed onto stack
3.	300	((	300 -	300 is written into postfix notation
4.	*	((*	300 300	* is pushed onto stack
5.	23	((*	300 23	23 is written into postfix notation
6.	)	(	300 23*	* is popped out
7	*	(*	300 23*	* is pushed onto stack
8.	(	(*(	300 23*	(is pushed
9.	43	(*(	300 23* 43	43 is written
10	-	(*(-	300 23* 43	- is pushed
11	21	(*(-	300 23* 43 21	21 is written
12.	)	(*	300 23* 43 21-	- is popped out
13.	/	(/	300 23* 43 21 - *	* is popped & / is pushed
14.	(	(/(	300 23* 43 21 - *	(is pushed.
15.	84	(/(	300 23* 43 21 - * 84	84 is written
16	+	(/(+	300 23* 43 21 - * 84	+ is pushed
17	7	(/(+	300 23* 43 21 - * 84 7	7 is pushed.

18	)	(	300 23 * 43 21 - * 84 7 +	+ is popped out
19	)	-	300 23 * 43 21 - * 84 7 + /	/ is popped out

Postfix Notation - 300 23 * 43 21 - * 84 7 + /

→ (a*(b+c)) + (b/d)*a

S.no	symbol	stack	Postfix notation	Description
0.		(		
1.	(	(	-	(is pushed
2.	a	(	a	a is written
3.	*	(*	a	* is pushed
4.	(	(*(	a	(is pushed
5.	b	(*(	ab	b is written
6.	+	(*(+	ab	+ is pushed
7	c	(*(+	abc	c is written
8	)	(*	abc+	+ is popped out.
9.	)	(	abc+*	* is popped out
10.	(	(	abc+*	(is pushed
11.	b	(	abc+*b	b is written
12.	/	(/	abc+*b	/ is pushed.
13.	d	(/	abc+*bd	d is written
14.	)	(	abc+*bd/	/ is pushed.
15.	*	(*	abc+*bd/	* is pushed

16 a (* abc+*bd/a a is written

17) - abc+*bd/a* * is popped out

Postfix Notation - abc+*bd/a*

→ Implementation

```
#include <stdio.h>
#include <ctype.h>
char stack[100];
int top = -1;
void push(char x){
    stack[++top] = x;
}
char pop(){
   if(top == -1)
      return -1;
   else
      return stack[top--];
}
int priority(char c){
    if(c == '^')
        return 3;
    else if(c == '/' || c == '*' || c == '%')
        return 2;
    else if(c == '+' || c = '-')
        return 1;
    else
        return 0;
}
int main(){
    char exp[100];
    char *e, x;
    printf("Enter the infix expression: ");
    scanf("%s", exp);
    printf("\n");
```

used to differentiate characters
isdigit - to check digit
isalpha - to check alphabet
isalnum - to check alphabet & numbers

```
    printf("\n Final Postfix Expression: ");
    while(*e != '\0'){
        if(isalnum(*e))
            printf("%c", *e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')'){
            while((x=pop()) != '(')
                printf("%c", x);
        }
        else{
            while(priority(stack[top]) >= priority(*e))
                printf("%c", pop());
            push(*e);
        }
        e++;
    }
    while(top != -1){
        printf("%c", pop());
    }
    return 0;
}
```

Input

Enter the infix expression: A+B*C/D-K+L^U

Output

Final Postfix Expression: A B C * D / + K - L U ^ +

→ Pos

Postfix Expression Evalution

Algorithm

Step 1 : Start

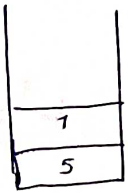
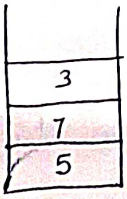
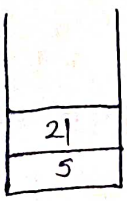
Step 2 : Scan the given postfix expression from left to right.

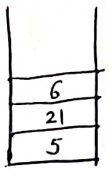
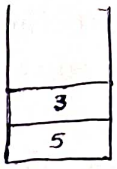
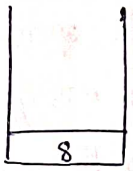
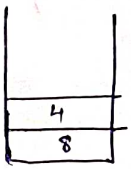
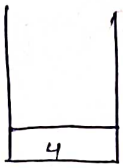
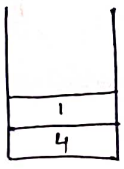
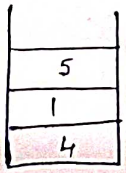
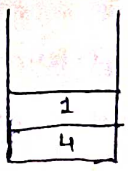
Step 3 : If the reading symbol is operand then push it onto the stack.

Step 4 : If the reading symbol is operator then perform two pop operations and store the two popped operands in 2 different variables (operand 1, operand 2). Then perform reading symbol operation using operand 1 and operand 2. and push result back to the stack.

Step 5 : Finally perform a pop operation and display the popped value as final result.

Eg: 5 7 3 * 6 / + 4 - 1 5 ^ +

S.no	Symbol	Stack	Evalution	Description
1.	5		-	5 is pushed onto stack
2.	7		-	7 is pushed onto stack
3.	3		-	3 is pushed onto stack
4.	*		op1 = 3 op2 = 7 op2*op1 = 7*3 = 21	3 & 7 is popped out, performed 7*3 and. 21 is pushed onto stack

S.no	symbol	stack	Evalution	Description
5.	6		–	6 is pushed onto
6.	/		op1 = 6 op2 = 21 op2 / op1 = 21/6 = 3	6 & 21 are popped o performed 21/6 & 3 is pushed onto sta
7	+		op1 = 3 op2 = 5 op2 + op1 = 5 + 3 = 8	3 & 5 are popped o performed 5+3 8 is pushed onto st
8	4		–	4 is pushed onto stac
9.	–		op1 = 4 op2 = 8 op2 - op1 = 8 - 4 = 4	4 & 8 are popped ou performed 8 - 4 4 is pushed onto st
10	1		–	1 is pushed onto sta
11.	5		–	5 is pushed onto stack
12	^		op1 = 5 op2 = 1 op2 ^ op1 = 1 ^ 5 = 1	5 & 1 are popped out performed 1^5 1 is pushed onto stack

S.no	symbol	stack	Evalution	Description
+3			op1=1 op2=4 op2+op1 = 4+1 =5	1 & 4 are popped out performed 4+1 5 is pushed onto stack

pop() = 5

Final Result = 5.

① A+B*(C+D)/F +D*E

S.no	symbol	stack	Evalution	Description
1	A	-	A	A is printed in postfix evalution
2	+	+	A	+ is pushed onto stack.
3.	B	+	AB	B is printed in postfix notation
4	*	+*	AB	* is pushed onto stack
5	(	+*(	AB	(is pushed onto stack
6	C	+*(	ABC	C is printed in postfix notation
7	+	+*(+	ABC	+ is ~~printed~~ pushed onto stack.
8	D	+*(+	ABCD	D is printed in postfix notation.
9.	)	+*	ABCD+	+ is popped out.
10	/	+/	ABCD+*	* is popped out.
11	F	+/	ABCD+*F	F is written into postfix notation

S.NO	symbol	stack	Evalution	Description
12	+	+	ABCD+*F/+	+ is pushed onto sta after / & + are popp out.
13	D	+*	ABCD+*F/+D	D is written onto st
14	*	+*	ABCD+*F/+D*	* is pushed onto stack after * is popped out.
15	E	+*	ABCD+*F/+D*E	E is written into postfix notation.
16	-	-	ABCD+*F/+D*E*+	+ & * are popped out.

Final Postfix Notation: ABCD+*F/+D*E*+

Given A = 5, B = 2, C = 3, D = 4, E = 6, F = 7

5 2 3 4 +* 7 / +4 * 6 *+

S.no	symbol	stack	Evalution	Description
1	5	5	-	5 is pushed onto postfix notation
2	2	5 2	-	2 is pushed onto postfix notation
3	3	5 2 3	-	3 is pushed onto postfix notation
4	4	5 2 3 4	-	4 is pushed onto postfix notation
5	+	5 2 7	op1 = 4 op2 = 3 op2+op1 = 4+3 = 7	4 & 3 are popped out and 4+3 is performed and 7 is pushed onto stack.

s.no	symbol	stack	Evalution	Description
6	*	5 14	op1 = 7 op2 = 2 op2*op1 = 14	7 & 2 are popped out and 7+2 is performed and 9 is pushed onto stack
7	7	5 14 7	op1 = 7 op2 = 14 —	7 is pushed onto stack
8	/	5 14 2	op1 = 7 op2 = 14 op2/op1 = 14/7 = 2	2 is pushed onto stack after performing 14/7 when 14 & 7 are popped out
9.	+	7	op1 = 5 op2 = 2 op2+op1 = 7	5 & 2 are popped out and 5+2 is performed and 7 is ~~popped out~~ pushed onto stack
10.	4	7 4	—	4 is pushed onto stack
11	*	28	op1 = 4 op2 = 7 op2*op1 = 28	4 & 7 are popped out and 4*7 is performed and 28 is ~~popped out~~ pushed onto stack
12	6	7 4 28 6	—	6 is pushed onto stack
13	*	7 24	op1 = 6 op2 = 24 op2*op1 = 6*4 = 24	6 & 24 are popped out & 4*6 is performed and 24 is pushed onto stack.
14	+	31	op1 = 24 op2 = 7	24 & 7 are popped out & 7+24 is performed and 31 is pushed onto stack.

Final Value Obtained: 31

→ ABC + * DE / -

A = 5 B = 6 C = 2 D = 12 E = 4

5 6 2 + * 12 4 / -

S.no	symbol	stack	Evalution	Description
1.	5	5	-	5 is pushed onto stack
2.	6	5 6	-	6 is pushed onto stack
3.	2	5 6 2	-	2 is pushed onto stack
4	+	5 8	op1 = 2 op2 = 6 op2+op1 = 6+2 = 8	2 & 6 are popped out and 6+2 is performed and 8 is pushed onto stack.
5	*	40	op1 = 8 op2 = 5 op2 * op1 = 5*8 = 40	8 & 5 are popped out & 8+5 is performed and 40 is pushed onto stack
6	12	40 12	-	12 is pushed onto stack
7	4	40 12 4	-	4 is pushed onto stack
8	/	40 3	op1 = 4 op2 = 12 op2 / op1 = 12/4 = 3	4 & 12 are popped out and 12/4 is performed and 3 is pushed onto stack.
9.	-	37	op2 = 40 op1 = 3 op2 - op1 = 37	40 & 3 are popped out & 40-3 is performed and 37 is pushed onto stack

Final Answer: 37

Evalute 7 4 -3 * 1 5 + / *.

s.no	symbol	stack	Evalution	Description
1	7	7	—	7 is pushed onto stack
2	4	7 4	—	4 is pushed onto stack
3	-3	7 4 -3	—	-3 is pushed onto stack
4	*	7 -12	op1 = -3 op2 = 4 op2 * op1 = -12	-3 & 4 are popped out -3 * 4 is performed -12 is pushed onto stack
5	1	7 -12 1	op —	1 is pushed onto stack
6	5	7 -12 1 5	—	5 is pushed onto stack
7	+	7 -12 6	op1 = 5 op2 = 1 op2 + op1 = 5+1 = 6	5 & 1 are popped out 1+5 is performed & 6 is pushed onto stack
8	/	7 -2	op1 = 6 op2 = -12 op2/op1 = -2	6 & -12 are popped out ~~6-12~~ -12/6 is performed & -2 is pushed onto stack
9	*	-14	op1 = -2 op2 = 7 op2 * op1 = 7 * (-2) = -14	-2 & 7 are popped out -2 * 7 is performed -14 is pushed onto stack

Final Value: -14

Implementation

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<math.h>
int stack[20];
int top=-1;
void push(int val)
{
    stack[++top]=val;
}
void pop()
{
    return(stack[top--]);
}
int main()
{
    int op1, op2, len, result, i;
    char postfix[30], op;
    printf("\n Enter postfix expression: ");
    scanf("%s", postfix);
    len=strlen(postfix);
    for(i=0; i<len; i++){
        if(isdigit(postfix[i]))
            push(postfix[i]-'0');
        else {
            op=postfix[i];
            op1=pop();
            op2=pop();
            switch(op){
                case '+': push(op2+op1);
                          break;
                case '-': push(op2-op1);
                          break;
                case '*': push(op2*op1);
                          break;
```

```
                case '/' : push(op2/op1);
                           break;
                case '%' : push(op2%op1);
                           break;
                case '^' : push(pow(op2, op1));
                           break;
            }
        }
    }
    printf("Final value obtained: %d", pop());
    return 0;
}
```

→ Balancing Brackets

Algorithm

Step 1 : Start

Step 2 : Declare a ~~stack~~ character stack S, traverse the expression string from left to right.

Step 3 : If the current character is:

step 4 : If the current character is starting bracket ('(' or '[' or '{') then push it onto the stack.

Step 5 : If the current character is a closing bracket. (')' or ']' or '}') then pop from stack and if the popped character is matching starting bracket then balanced otherwise brackets are ~~not~~ balanced.

Step 6 : After complete traversal, if there is some starting bracket left in the stack. then the expression is not balanced.

Step 7 : Stop.

Eg: [{( )( )( )( )}] - Balanced.

[( )( )( )( )] - Not Balanced

[[[{ { ( ) } }]]] - Balanced

Implementation

```
#include<stdio.h>
#include <string.h>
#define MAX 100:
 char stack[MAX];
 int top=-1;
 void push(char symbol){
     if (top == MAX-1)
        printf("stack is full \n");
     else
        stack [++top] = symbol;
}
 void pop (){
    if (top==-1)
       printf("Stack is empty/n");
    else
       top--;
}
 int checkPair (char val1, char val2){
      return ((val1 == '(' && val2==')') || (val1=='[' && val
         || (val1=='{' && val2=='}'))
}
 int checkBalanced (char expr[], int len){
    int i;
    for(i=0; i<len; i++){
       if(expr[i]=='(' || expr[i]=='[' || expr[i]=='{'){
          push(expr[i]);
       }
```

```
            else {
                if (top == -1)
                    return 0;
                else if (checkPair (stack[top], expr[i])) {
                    pop();
                    continue;
                }
                return 0;
            }
        }
        return 1;
}
int main(){
    char exp[MAX];
    printf("Enter expression string: ");
    scanf("%s", exp);
    int len = strlen(exp);
    checkBalanced(exp, len) ? printf("Balanced") : printf("Not Balanced");
    return 0;
}
```

Input

{([])}

Output

Balanced

→ Test-1

CO1

→ Derivation of Time complexities of Merge & Quick in 3 cases.

→ Arranging time complexities.

→ 6 Sorting Techniques. (4 programs)

- Algorithm
- Work Out
- Program
- Time Complexity. (Merge, Quick, Insertion, Shell)

CO2

→ SLL, DLL, CSLL, CDLL, SSLL, QSLL, Infix to postfix, postfix, balancing brackets, CQ, PQ., Double Ended Queue.

- Search
- reverse
- delete a ~~no~~ key node.
- min, max, duplicate, sum.

→ Types Of Queues

- Circular Queue
- Priority Queue
- DeQueue

→ Circular Queue

It is a linear data structure in which front & rear are both are adjacent to each other. (which means last is connected to first)

Applications:

- Memory Management
- CPU Scheduling algorithms.
- Traffic systems.

Operations

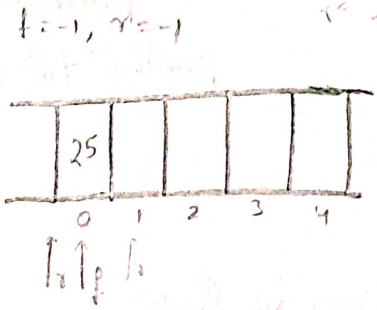
Enqueue()

Dequeue()

Display()

Implementation

```
#include<stdio.h>
int cq[100], rear=-1, front=-1;
void enqueue(int ele)
{
    if(front == (rear+1)%size)
        printf("Circular Queue is full");
    else if(front==-1 && rear==-1){
        front=rear=0;
        cq[rear]=ele;
    }
    else{
        rear=(rear+1)%size;
        cq[rear]=ele;
    }
}
void dequeue()
{
    if(front==-1)
        printf("Circular Queue is empty");
    else if(front==rear){
        printf("%d element is deleted!", cq[front]);
        front=rear=-1;
    }
    else{
        printf("%d element is deleted!", cq[front]);
        front=(front+1)%size;
    }
}
void display(){
{
    if(front==-1 && rear==-1)
        printf("Circular Queue is Empty");
```



```
    else {
        for(int i = front; i!=rear; i = (i+1)%.size)
            printf("%d", cq[i]);
        printf("%d", cq[rear]);
    }
}
```


→ Priority Queue

It is a special type of queue in which each node will have predefined priority to service.

```
#include<stdio.h>
struct node {
    int data;
    int priority;
    struct node *next;
} *head = = NULL, *current, *p, *q;
void pEnqueue(int data, int priority)
{
    current = (struct node *) malloc(sizeof(struct node));
    current -> data = data;
    current -> priority = priority;
    current -> next = NULL;
    if(head == = NULL)
        head = current;
    else if ( current -> priority < head -> priority) {
        p = head;
        head p =
        current -> next = head;
        head = current;
    }
    else {
        p = head;
        while q = head;
```

```
            while (p != NULL && (p->priority < current->priority)) {
                p=p->next;
                q=p;
                p=p->next
            }
            current->next=p;
            q->next = current;
        }
}
void pDequeue()
{
    if (head == NULL)
        printf("Queue is empty!");
    else {
        p=head;
        head=p->next;
        p->next=NULL;
        free(p);
    }
}
void display()
{
    if (head == NULL)
        printf("Queue is empty");
    else {
        p=head;
        while (p!=NULL) {
            printf("%d ", p->data);
            p=p->next;
        }
    }
}
```

Double-Ended Queue

① WCP to convert an expression an expression from infix to postfix and convert the following expression A*B - (C+D)*(

② Evaluate the following postfix expression using stack.

12 7 3 - 1 2 1 5 + * +

③ Write the algorithm for insertion sort and sort the following numbers 20, 35, 40, 100, 3, 10, 15, 1

④ Write an algorithm to sort given list of names in alphabetical order using bucket sort.

⑤ WCP to imptiment stack operations using Singly Linked List.

⑥ Write mergeSort Algorithm & sort following elements 23, 11, 37, 28, 15, 19, 55, 9.

⑦ Write a functional code for deleting a desired node in a DoublyLinked list.

⑧ WCP to perform enqueue, dequeue and display operations in a priority queue.

⑤

```
#include <stdio.h>
struct node {
    int data;
    struct node* next;
}*top = NULL, *current, *p, *q;
void push()
{
    current = (struct node*) malloc (sizeof(struct node));
    scanf("%d", &current->data);
    current->next = NULL;
    if (top == NULL)
        head = current;
```

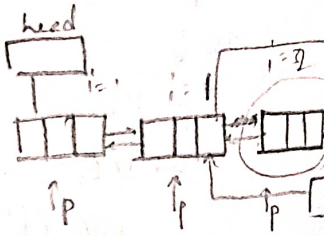
```
        else {
            p=head;
            while(p!=
            current->next=head=top;
            head = current;
        }

}
void pop()
{
    if (head==NULL)
        printf("Stack is empty");
    else {
        p = top;
        top = p->next;
        p->next = NULL;
        free(p);
    }
}
    display
void peak()
{
    if (top==NULL)
        printf("Stack is empty");
    else {
        p= top;
        while(p!=NULL){
            printf("%d ", p->data);
            p=p->next;
        }
    }
}

void peak()
{
    if (top==NULL)
        printf("Stack is empty");
    else {
        printf("%d", top->data);
}
```

```
int main()
{
    int ch;
    printf("1. push\n2. pop\n3. peak\n4. display\n5. exit\n");
    while(1){
        scanf("%d", &ch);
        switch(ch) {
            case 1: push();
                    break;
            case 2: pop();
                    break;
            case 3: peak()
                    break;
            case 4: display()
                    break;
            case 5: exit(0);
            default: printf("invalid");
    } }
    }
}
```

(7)




```
void deleteAtMiddle()
{
    int k, i;
    scanf("%d", &k);
    if(k>c)
        printf("deletion not possible");
    else if (head==NULL)
        printf("Not possible");
    else {
        p=head;
        i=1;
        while(i<k){
            p=p->next;
        p->prev->next = p->next;
        p->next->prev = p->prev
        p->prev = p->next = NULL; free(p); } }
```

```
#include <stdio.h>
struct node {
    int data, priority;
    struct node *next;
};
struct node *head = NULL, *current, *p, *q;
void enqueue()
{
    current = (struct node *) malloc (struct node);
    scanf("%d", &current->data);
    current->next = NULL;
    scanf("%d", &current->priority);
    if (head == NULL)
        head = current;
    else if (current->priority < head->priority)
        current->next = head;
        head = current;
    }
    else {
        p = head; q = head;
        while (p != NULL &&
            if (current->priority > p->priority))
                q = p;
                p = p->next;
        }
        current->next = q->next;
        q->next = current;
    }
}
void dequeue()
{
    if (head == NULL)
        printf("PQ is empty");
    else {
        p = head;
        head = p->next;
        p->next = NULL;
        free(p);
    }
}
```

```
void display()
{
    p = head;
    while(p != NULL){
        printf("%d -> %d -> ", p->data, p->priority);
        p = p->next;
    }
}
int main()
{
    int ch;
    printf("1. enqueue\n2. dequeue\n3. display\n");
    while(1){
        scanf("%d", &ch);
        switch(ch){
            case 1: enqueue();
                    break;
            case 2: dequeue();
                    break;
            case 3: display();
                    break;
            default: exit(0);
        }
    }
}
```

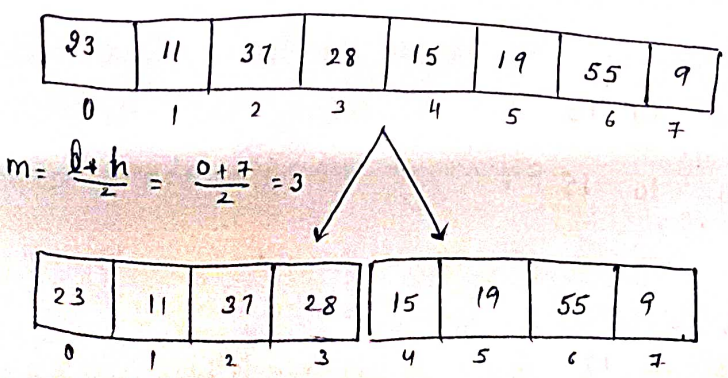
(3)

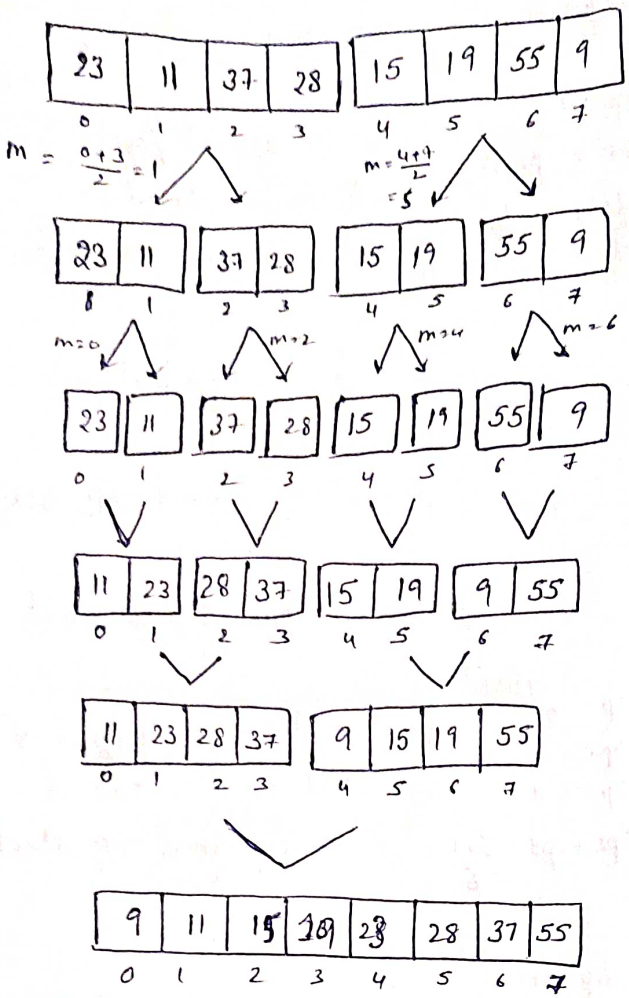
12 7 3 - / 2 1 5 + * +

Symbol	Stack	Evalution	Description
12	12	-	12 is pushed onto stack
7	12 7	-	7 is pushed onto stack
3	12 7 3	-	3 is pushed onto stack
-	12 4	op = - op1 = 3 op2 = 7 op2 - op1 = 7 - 3 = 4	3 & 7 are popped & 7 - 3 is evaluated and

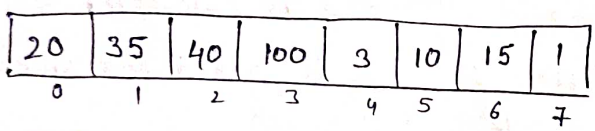
Symbol	Stack	Evalution	Description
/	3	op = / op1 = 4 op2 = 12 op2/op1 = 12/4 = 3.	12 & 4 are popped & 12/4 is calculated & 3 is pushed onto stack.
2	3 2	–	2 is pushed onto stack
1	3 2 1	–	1 is pushed onto stack
5	3 2 1 5	–	5 is pushed onto stack
+	3 2 6	op = + op1 = 5 op2 = 1 op2 + op1 = 5+1 = 6	5 & 1 are popped & 5+1 is calculated & 6 is pushed onto stack
*	3 12	op = * op1 = 6 op2 = 2 op2 * op1 = 2 * 6 = 12	6 & 2 are popped & 6*2 is calculated & 12 is pushed onto stack
+	15	op = + op1 = 12 op2 = 3 op2 + op1 = 3+12 = 15	12 & 3 are popped & 3+12 is calculated & 15 is pushed onto stack
–	–	–	15 is popped out.

6.





3.



P-1

20 | 35 40 100 3 10 15 1

P-20 35 40 100 3 10 15 1

P-2

20 35 | 40 100 3 10 15 1

20 35 40 100 3 10 15 1

P-3

20 35 40 | 100 3 10 15 1

20 35 40 100 3 10 15 1

P-4

20 35 40 100 | 3 10 15 1

20 35 40 3 100 10 15 1

20 35 3 40 100 10 15 1

20 3 35 40 100 10 15 1

3 20 35 40 100 10 15 1.

P-5

3 20 35 40 100 10 15 1

3 20 35 40 10 100 15 1

3 20 35 10 40 100 15 1

3 20 10 35 40 100 15 1

3 10 20 35 40 100 15 1

P-6

3 10 20 35 40 100 15 1

3 10 20 35 40 15 100 1

3 10 25 35 15 40 100 1

3 10 25 15 35 40 100 1

3 10 15 25 35 40 100 1

P-7

3 10 15 25 35 40 100 1

3 10 15 25 35 40 1 100

3 10 15 25 35 1 40 100

3 10 15 25 1 35 40 100

3 10 15 1 25 35 40 100

3 10 1 15 25 35 40 100

3 1 10 15 25 35 40 100

1 3 10 15 25 35 40 100

∴ Final Sorted Array: 1 3 10 15 25 35 40 100

1.

A * B − (C+D)* (P/Q).

Symbol	Stack	Postfix Notation	Description
A	−	A	A is written ~~on~~ in P.F.N
*	*	A	* is pushed onto stack
B	*	AB	B is written on P.F.N
−	* −	AB*	* is popped & − is pushed.
(	− (	AB*	(is pushed.
C	− (	AB*C	C is written in P.F.N
+	− (+	AB*C	+, (is pushed.
D	− (+	AB*CD	D is written in P.F.N
)	−	AB*CD+	+ is popped
*	− *	AB*CD+	* is pushed.
(	− *(	AB*CD+	(is pushed.
P	− *(	AB*CD+P	P is written on P.F.N
/	− *(/	AB*CD+P	/ is pushed
Q	− *(/	AB*CD+PQ	Q is written on P.F.N

) -* AB+CD+PQ/ / is popped and (is neglected.

- - AB*CD+PQ/*- all are popped.

Final Postfix expression : AB*CD+PQ/*.

Implementation

```
#include <stdio.h>
#include <ctype.h>
int stack[100], top=-1;
void push (char x){
   stack[++top]=x;
}
int
~~void~~ ~~char~~ pop (){
    if(top==-1)
       return -1;
    else
       return stack[top--];
}
int priority (char x){
    if(x=='^')
       return 3;
    else if (x=='/'||x=='*'||x=='%')
       return 2;
    else if (x=='+'||x=='-')
       return 1;
    else
       return 0;
}
int main()
{
    char exp[100],*e;
    scanf("%s",exp);
    e=exp;
```

```
while ( e != NULL) {
    if (isalnum (*e))
        printf ("%c", *e);
    else if (*e == '(')
        push(*e);
    else if (*e == ')') {
        while ((x = pop()) != '(')
            printf ("%c", x);
        push(*e);
    }
    else {
        while ( priority (*e) < priority (stack [top]))
            printf ("%c", pop());
        push (*e);
    }
}
while ( top != -1)
    printf (pop());
return 0;
}
```

③ Algorithm for Insertion Sort

Step 1 : Start;

Step 2 : If the array has 0 or 1 then it is already sorted.

Step 3 : Otherwise, first element is already sorted. start from next element.

Step 4 : Check the element with the elements in sorted array

Step 5 : If the element is less than elements in sorted array keep it in its place.

Step 6 : Continue th step 4 and 5 for all elements in array until we get sorted array

Step 7 : Stop.

⑧ Algorithm for Bucket Sort

Step 1: Start

Step 2: Create n no of buckets.

Step 3: In an array, drop each element a[i] in bucket of range [n*a[i]].

Step 4: Apply Insertion or bucket sort on each bucket. to get sort elements.

Step 5: Combine all to get sorted array.

Step 6: Stop.

⑨ Algorithm for merge sort

Step 1: Start

Step 2: If array has 0 or 1 elements array is already sorted.

Step 3: Otherwise divide the array into 2 parts.

Step 4: Apply merge sort algorithm recursively on each ~~part~~ parts. ~~merged~~ pa until we get single element in each part

Step 5: Merge All to get sorted array. using swapping.

Step 6: Stop.

→ Time Complexity

```
for(i=1; i<n;i++){
    for(j=i; j
```