19/01/22

Data Structures

Data Structures:

Data structure is a specific way to store and organize (arrange) data in a computer's memory

Categories of Data structure:

→ The data structure can be sub divided into two types

i, Linear Data structure

ii, Non-linear Data structure.

Linear Data structure:

Data is arranged in sequential order or linearly and each member element is connected to its previous and next element.

Examples:

- Arrays
- Queues
- Stacks
- Linked lists
  - → Singly linked list
  - → Doubly linked list
  - → circular linked list

Non Linear Data structure:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

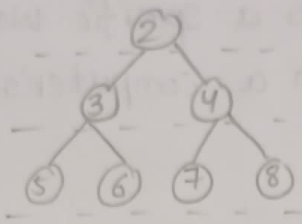
Examples:

- Graphs
- Trees

Tree:

The data structure that reflects hierarchical relationship among various elements is termed as a tree

ex:

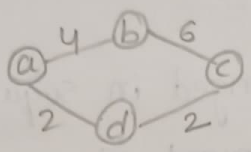


follows hirority order

Graph:

The data structure that holds a relationship between the pair of elements which is not necessarily following hierarchical structure.

ex:



follows Shortest path

Sorting technique:

Sorting refers to the operation (or) technique of arranging (or) rearranging the elements in some specific (or) particular order

categories of sorting:

The techniques of sorting can be divided into two categories.

i, Internal sorting
ii, External sorting

Internal sorting:

If all the data that is to be sorted can be adjusted at a time in the "main memory" (RAM)

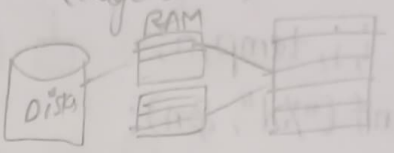
External sorting:

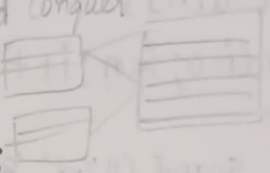
When the data that is to be sorted cannot be accomodated in the memory at the same time and some has to be kept in auxiliary memory

such as "hard disks".

Sorting technique

- Internal sorting
  - i) Bubble sort
  - ii) Insertion sort
  - iii) Shell sort
  - iv) Quick sort } follows divide and conquer
  - v) Merge sort
- External sorting (large data)





20/01/22

Insertion sorting [In place sorting]:

Insertion sorting is a simple sorting Algorithm that builts the final sorted array one item at a time.

Example:

0	1	2	3	4	5
5	1	6	2	4	3

5 1 6 2 4 3

temp = 1

1 5 6 2 4 3

temp = 2

1 5 6 2 4 3

1 5 2 6 4 3

temp = 4

1 2 5 6 4 3

1 2 5 4 6 3

temp = 3

1 2 4 5 6 3

1 2 4 5 3 6

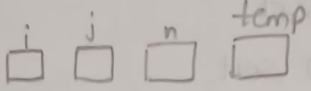
1 2 4 3 5 6

0	1	2	3	4	5
1	2	3	4	5	6

(Final sorted array)

Programm for Insertion sort:

```
#include<stdio.h>
int main( )
{
  int i, j, temp, n;
  scanf ("%d", &n);
  int a[n];
  for (i=0; i<n; i++)
   {
     scanf ("%d", &a[i]);
   }
 for (i=1; i<n; i++)
  {
    for (j=i ; j>0; j= j-1)
  {
    if (a[j]<a[j-1])
    {
      temp= a[j];
       a[j] = a[j-1];
      a[j-1] = temp;
    }
 for (i=0; i<n; i++)
 {
   Printf("%d\t", a[i]);
 }
}
```



Example 2:

0	1	2	3	4	5	6	7
10	7	35	9	25	23	86	65

10 7 35 9 25 23 86 65

7 10 35 9 25 23 86 65

7 10 35 9 25 23 86 65

7 10 9 35 25 23 86 65

7 9 10 35 25 23 86 65

7 9 10 25 35 23 86 65

7 9 10 25 23 35 86 65

7 9 10 23 25 35 86 65

7	9	10	23	25	35	65	86

Example 3:

0	1	2	3	4	5	6	7	8	9
30	-14	29	26	37	11	-9	10	90	44

30 -14 29 26 37 11 -9 10 90 44

-14 30 29 26 37 11 -9 10 90 44

-14 29 30 26 37 11 -9 10 90 44

-14 29 26 30 37 11 -9 10 90 44

-14 26 29 30 37 11 -9 10 90 44

-14 26 29 30 37 11 -9 10 90 44

-14 26 29 30 11 37 -9 10 90 44

-14 26 29 11 30 37 -9 10 90 44

-14 26 11 29 30 37 -9 10 90 44

-14 11 26 29 30 37 -9 10 90 44

-14 11 26 29 30 -9 37 10 90 44

-14 11 26 29 -9 30 37 10 90 44

-14 11 26 -9 29 30 37 10 90 44

-14 11 -9 26 29 30 37 10 90 44

-14 -9 11 26 29 30 37 10 90 44

-14 -9 11 26 29 30 10 37 90 44

-14 -9 11 26 29 10 30 37 90 44

-14 -9 11 26 10 29 30 37 90 44

-14 -9 11 10 26 29 30 37 90 44

-14 -9 10 11 26 29 30 37 90 44

-14 -9 10 11 26 29 30 37 90 44

-14 -9 10 11 26 29 30 37 44 90

-14	-9	10	11	26	29	30	37	44	90

Example 4:

consider there are 10 members in a cinema hall to buy tickets from the Counter. The height of these 10 members is given in centimetres as follows

0	1	2	3	4	5	6	7	8	9
165	173	195	150	179	184	139	145	169	151

165 173 195 150 179 184 139 145 169 151

165 173 195 150 179 184 139 145 169 151

165 173 195 150 179 184 139 145 169 151

165 173 150 195 179 184 139 145 169 151

165 150 173 195 179 184 139 145 169 151

150 165 173 195 179 184 139 145 169 151

150 165 173 179 195 184 139 145 169 151

150 165 173 179 184 195 139 145 169 151

150 165 173 179 184 139 195 145 169 151

150 165 173 179 139 184 195 145 169 151

150 165 173 139 179 184 195 145 169 151

150 165 139 173 179 184 195 145 169 151

150 139 165 173 179 184 195 145 169 151

139 150 165 173 179 184 195 145 169 151

139 150 165 173 179 184 145 195 169 151

139 150 165 173 179 145 184 195 169 151

139 150 165 173 145 179 184 195 169 151

139 150 165 145 173 179 184 195 169 151

139 150 145 165 173 179 184 195 169 151

139 145 150 165 173 179 184 195 169 151

139 145 150 165 173 179 184 169 195 151

139 145 150 165 173 179 169 184 195 151

139 145 150 165 173 169 179 184 195 151

139 145 150 165 169 173 179 184 195 151

139 145 150 165 169 173 179 184 151 195

139 145 150 165 169 173 179 161 184 195

139 145 150 165 169 173 151 179 184 195

139 145 150 165 169 151 173 179 184 195

139 145 150 165 151 169 173 179 184 195

139 145 150 151 165 169 173 179 184 195

0	1	2	3	4	5	6	7	8	9
139	145	150	151	165	169	173	179	184	195

21/01/22

Shell Sorting:

* Shell sort takes the help of Insertion Sort.

Example:

9	8	2	7	5	6	4	1

$n = 8$

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

phase-1:

9	8	2	7	5	6	4	1

9 swap 5

5	8	2	7	9	6	4	1

8 swap 6

5	6	2	7	9	8	4	1

2 No swap 4

5	6	2	7	9	8	4	1

7 swap 1

5	6	2	1	9	8	4	7

phase 2:

5	6	2	1	9	8	4	7

$gap = gap/2 = 4/2 = 2$

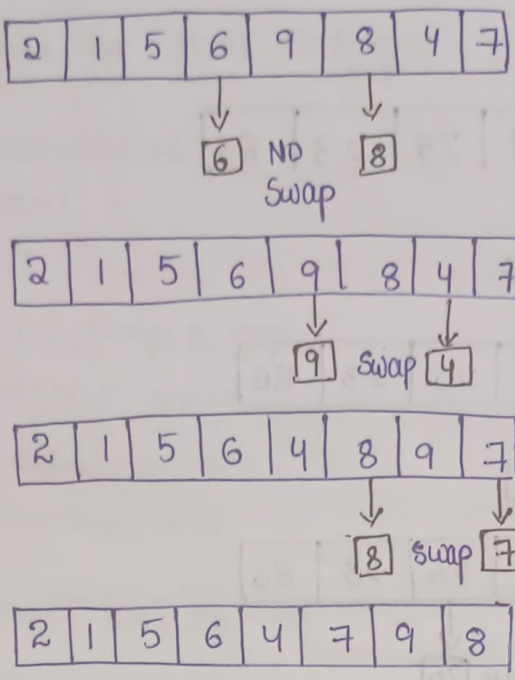
5 swap 2

2	6	5	1	9	8	4	7

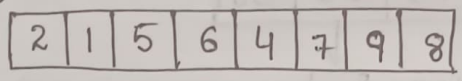
6 swap 1

2	1	5	6	9	8	4	7

5 NO swap 9



Phase-3:



gap = gap/2 = 2/2 = 1

⟶ If gap = 1, perform Insertion sorting.

2 1 5 6 4 7 9 8

2 1 5 6 4 7 9 8

1 2 5 6 4 7 9 8

1 2 5 6 4 7 9 8

1 2 5 6 4 7 9 8

1 2 5 4 6 7 9 8

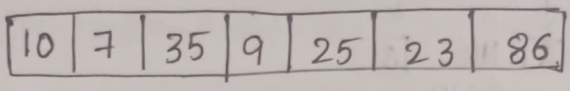
1 2 4 5 6 7 9 8

1 2 4 5 6 7 9 8

1 2 4 5 6 7 9 8

1 2 4 5 6 7 8 9

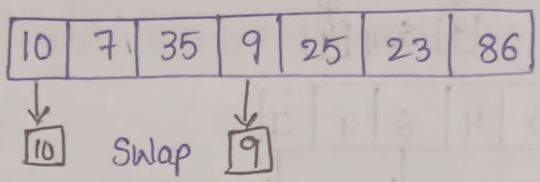
Example 2:

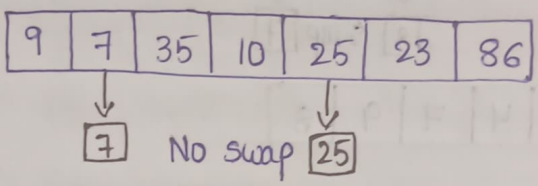


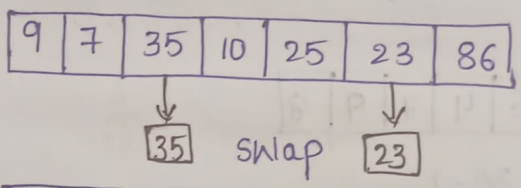
phase-1:

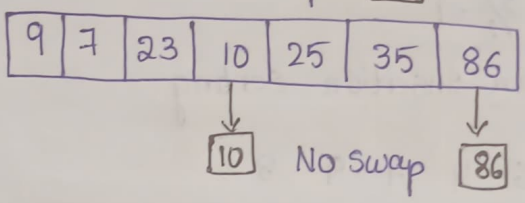
$n = 7$

$gap = n/2 = 3$.

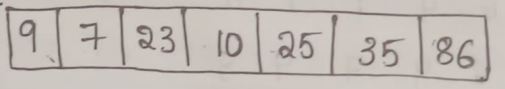








phase 2:



$gap = gap/2 = 3/2 = 1$

→ If gap=1, perform Insertion Sorting

9 7 23 10 25 35 86

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

Programm for shell sorting:

```
#include<stdio.h>
int main( )
{
  int i,j, temp,n, gap;
  scanf("%d", &n);
  int a[n];
  for(i=0; i<n ;i++)
  {
    scanf("%d", &a[i]);
  }
 for(gap= n/2 ; gap>=1; gap = gap/2)  [To calculate gap]
 {
   for(i=gap ; i<n ; i++)  [To check all elements]
  {
     for (j=i; j>=gap; j= j-gap)
     {
        if (a[j]<a[j-gap])
       {
          temp=a[j];
          a[j] = a[j-gap];
          a[j-gap]= temp;
       }
     }
  }
   for(i=0; i<n ; i++)
   {
     printf("%d", a[i]);
   }
}
```

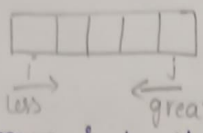
Quick Sorting: [partition exchange sort]

→ set the pivot element

→ Initialize i and j

* i = 0,
* j = n-1

→ move i and j



* move i to the right and j to the left

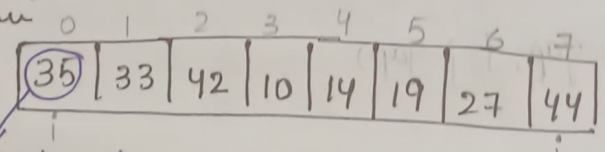
→ if i and j do not cross each other

swap a[i] and a[j]

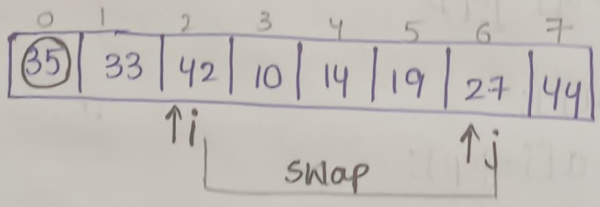
else

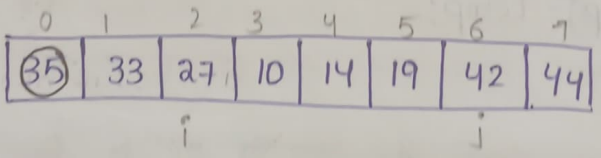
swap a[j] and a[pivot]

Example:

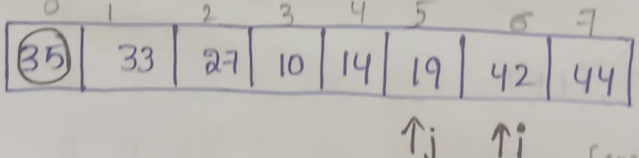


pivot element

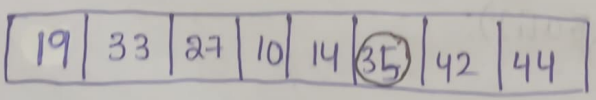


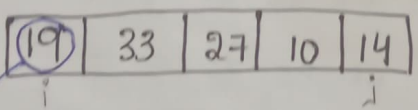


→

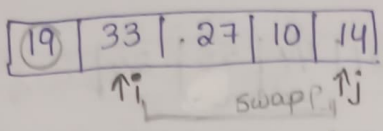


[crossed each other]

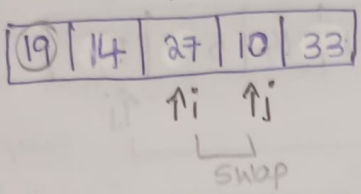




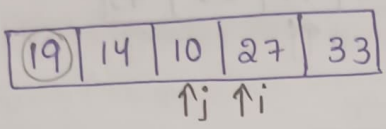
pivot element



Swap a[i] and a[j]

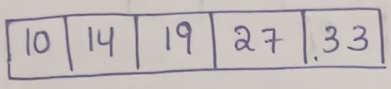


swap a[i] and a[j]

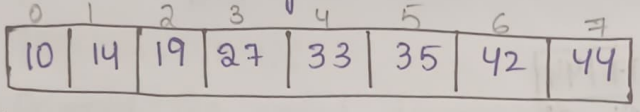


[crossed each other]

Swap a[j] and pivot

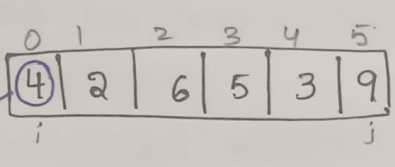


final sorted array is

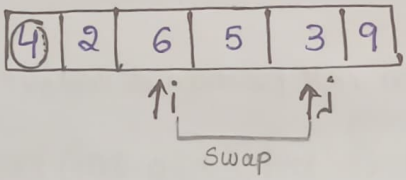


25/01/22

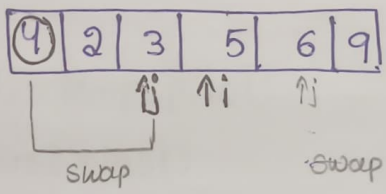
Example:



pivot element

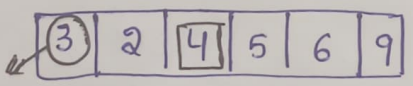


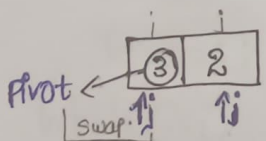
Swap a[i] and a[j]



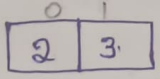
i and j crossed each other (i<j)

Swap a[j] and a[pivot]

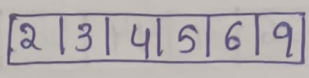




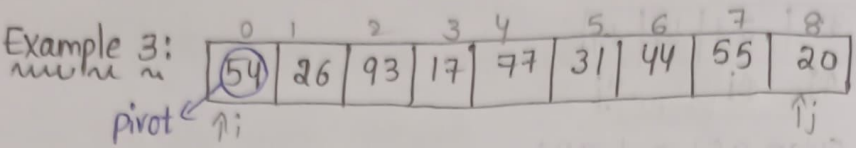
Swap a[j] and a[pivot]

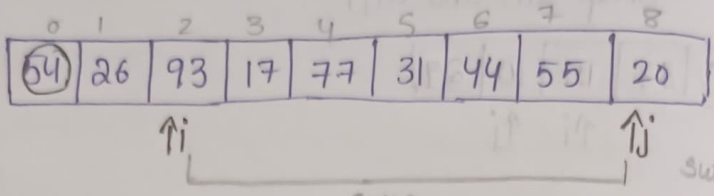


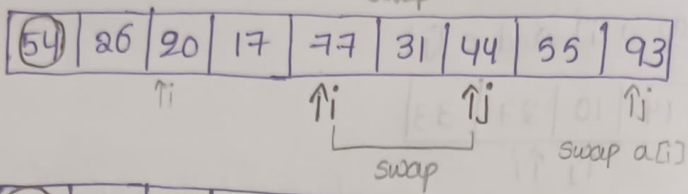
final sorted array is

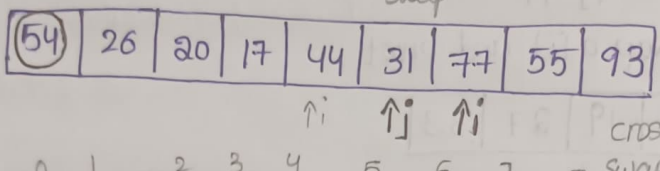


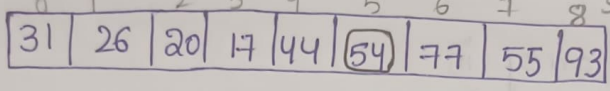
Example 3:

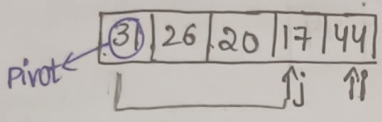


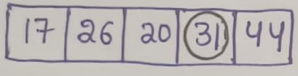


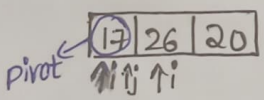




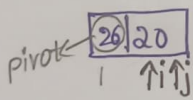




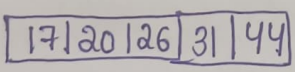


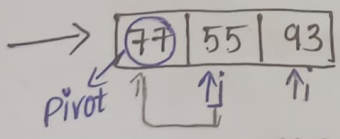


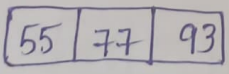
Pivot is in right position it will not move



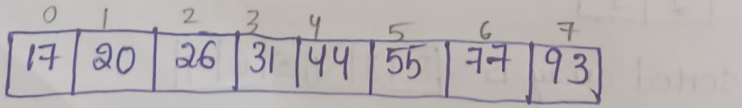
Swap a[j] and a[pivot]







∴ The final sorted array is



26/01/22

Programm for Quick Sort:

```
#include<stdio.h>
void qsort(int[ ], int, int);
int main( )
{
  int n, i, a[15];
  Printf("Enter the number of elements");
  Scanf("%d",&n);
  for(i=0; i<n; i++)
  {
    Scanf("%d", &a[i]);
  }
  qsort(a, 0, n-1);
  Printf("The sorted elements");
  for(i=0; i<n; i++)
  {
    Printf("%d\t", a[i]);
  }
}
Void qsort(int a[15], int first, int last)
{
  int i, j, temp, pivot;
  if(first<last)
  {
    Pivot=first;
      i=first;
      j=last;
  While(i<j)
  {
  While(a[i]<=pivot && i<last)
   i++;
```

actual parameters

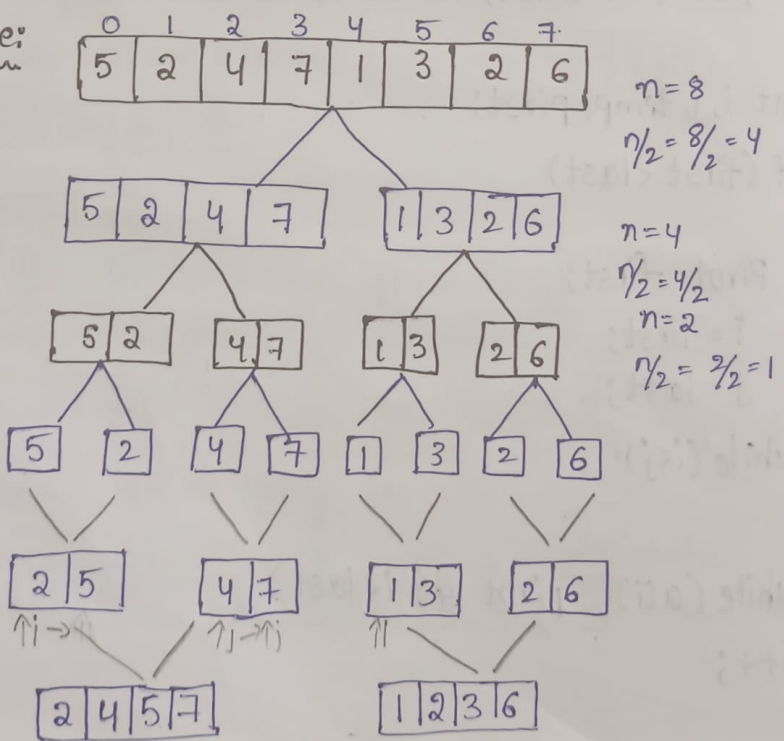
formal parameters

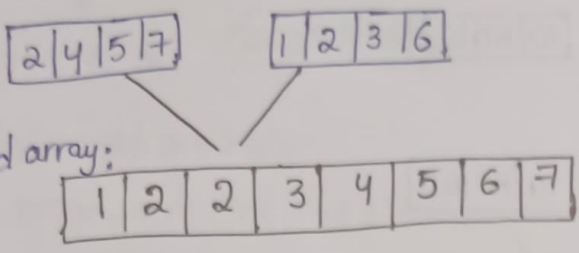
```
While (a[i]>a[pivot])
j--;
if (i<j)
 {
  temp=a[i];
   a[j]=a[j];
    a[j]=temp;
 }
}
 temp=a[pivot];
 a[pivot]=a[j];
   a[j] =temp;
 qsort (a, first, j-1);
 qsort (a, j+1, last);
}
}
```

Merge sorting:

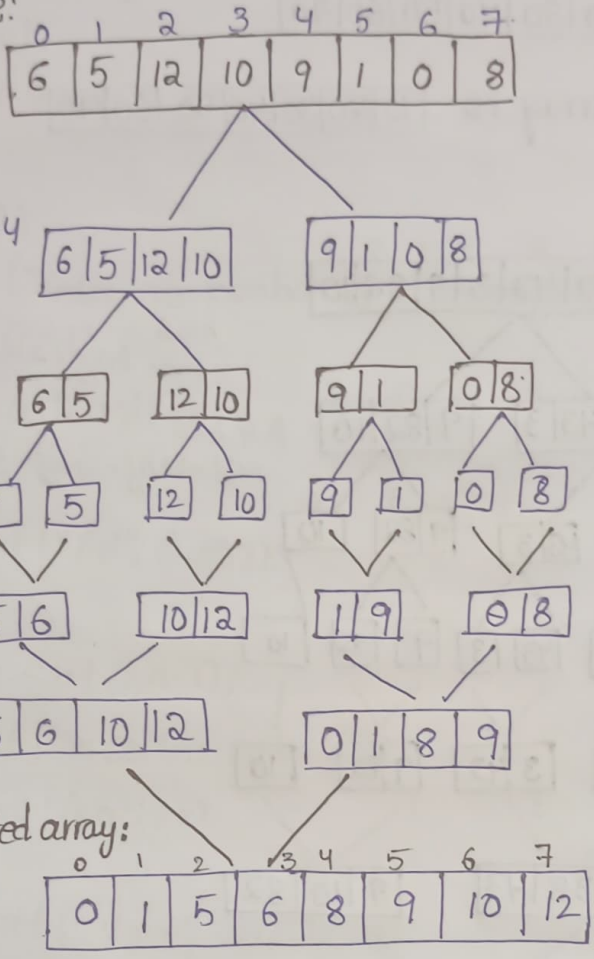
process: Split

Example:





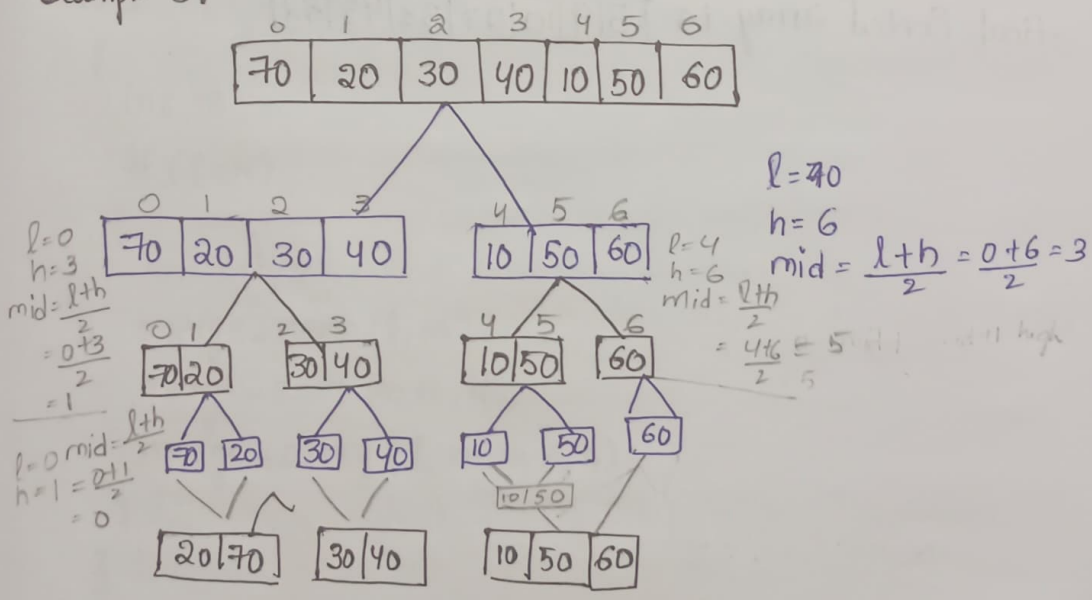
Example 2:

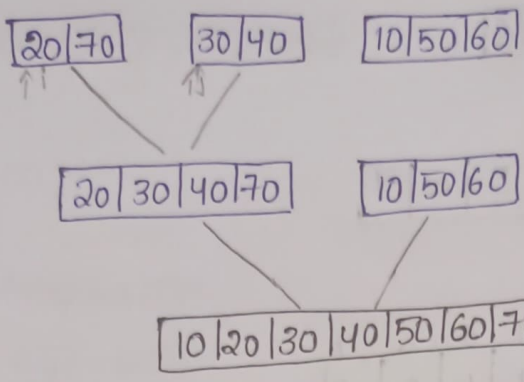


final sorted array:

28/11/22

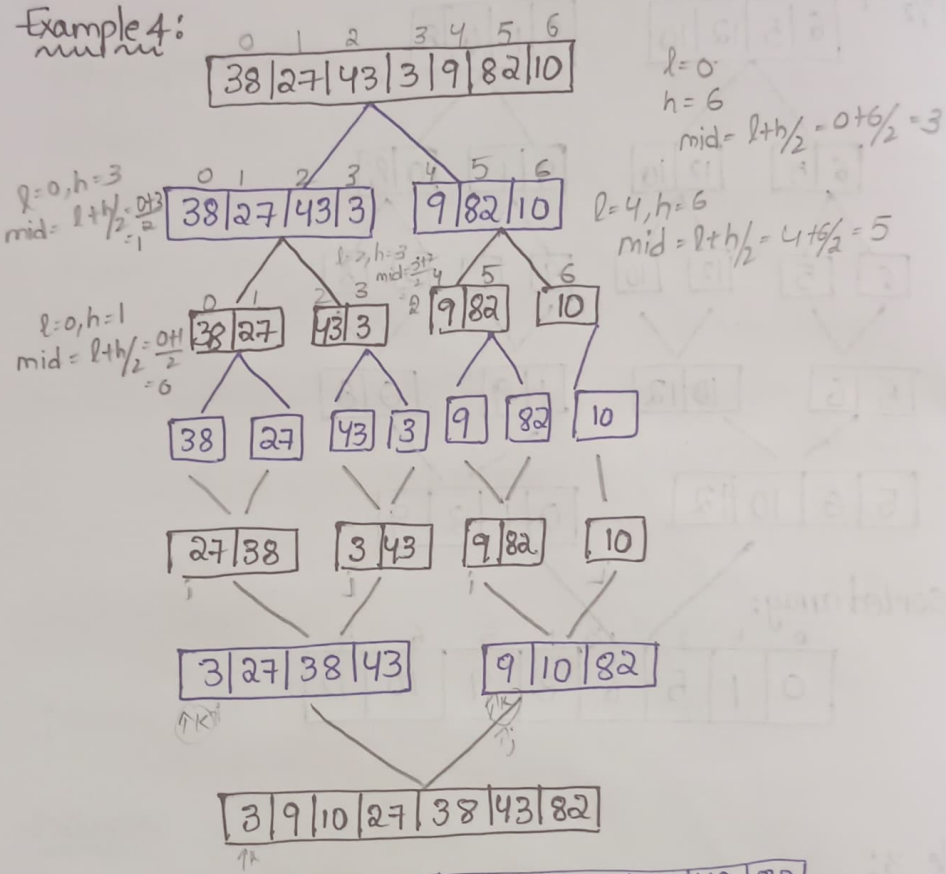
Example 3:





final sorted array is 10 20 30 40 50 60 70

Example 4:



final Sorted array is 3 9 10 27 38 43 82

Programme for Merge Sort:

```
#include <stdio.h>
void merge_sort (int, int);
void merge - array (int, int, int, int);
int arr[100];
int main( )
{
  int i, n;
  Printf ("Enter the number of Elements":);
  Scanf ("%d", &n);
  for (i=0; i<n; i++)
  {
   Scanf ("%d", &arr[i]);
  }
  merge_sort (0, n-1);
  Printf ("\n Sorted Data:");
  for (i=0; i<n; i++)
  {
   Printf (" %3d", arr[i]);
  }
}
void merge_sort (int l, int h)
{
  int m;
  if (l<h)
  {
   m= l+h/2;
   merge_sort (l, m);
   merge - sort (m+1, h);
   merge - array (l, m, m+1, h);
  }
}
```

```
void merge_array(int a, int b, int c, int d)
{
  int t[100];
  int i=a, j=c, k=0;
  while(i<=b && j<=d)
  {
    if (arr[i]<arr[j])
    t[k++] = arr[i++];
   else
      t[k++]= arr[j++];
  }
// collect remaining elements
  while(i<=b)
   t[k++] = arr[i++];
// copy elements to the original array
  while(j<=d)
   t[k++] = arr[j++];
  for(i=a; j=0; i<=d; i++, j++)
     arr[i] = t[j];
}
```

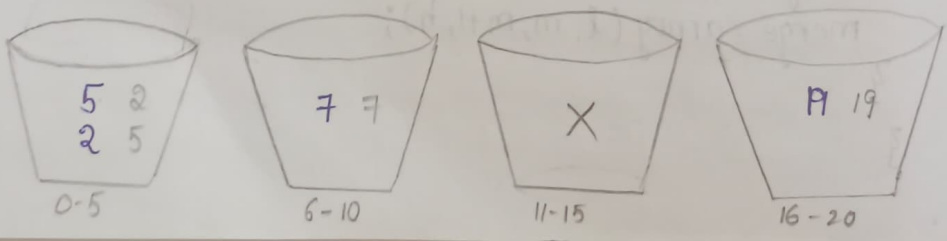
Bucket sorting:

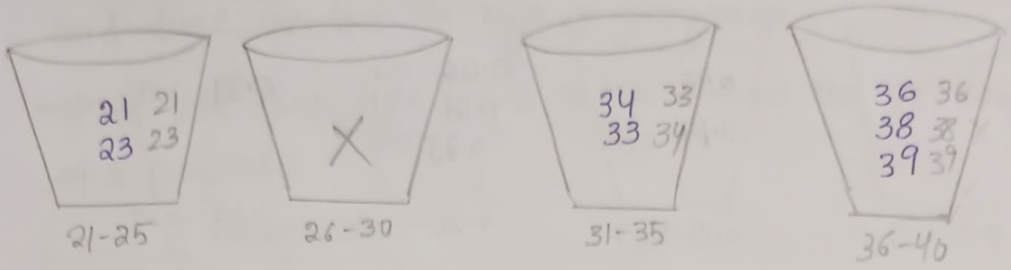
Bucket sort is used when input data is uniformly distributed over a range

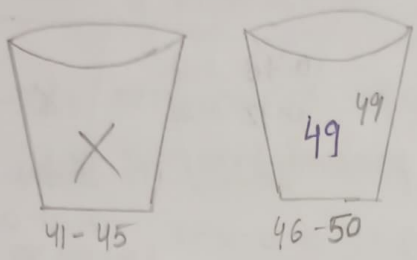
Example:

0	1	2	3	4	5	6	7	8	9	10	11
5	2	7	19	21	23	34	33	36	39	38	49

Number of buckets = 10 ; Range = 5







final Sorted array is

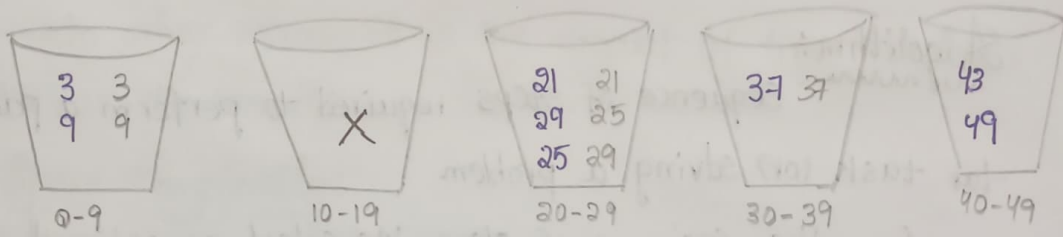
2	5	7	19	21	23	39	34	36	38	39	49

Example 2:

0	1	2	3	4	5	6	7
29	25	49	3	9	37	21	43

and range is 10.

Number of buckets = 10, Range is



final sorted array is

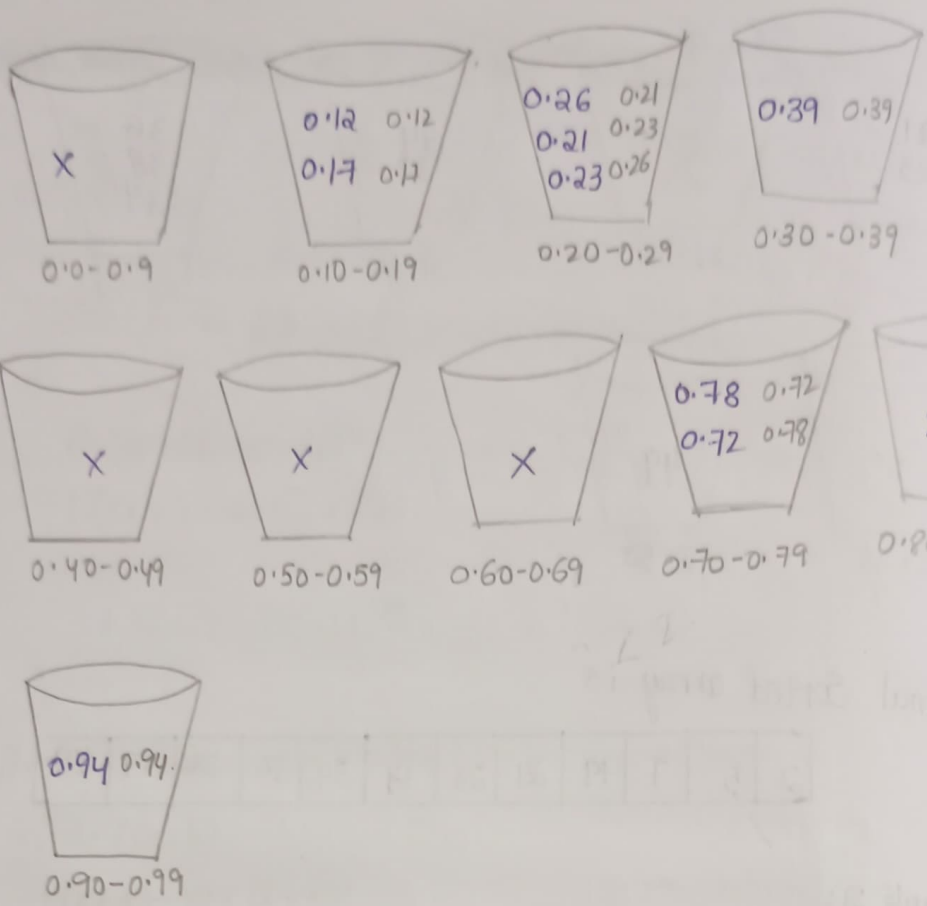
3	9	21	25	29	37	43	49

Example 3:

0	1	2	3	4	5	6	7	8
0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23

Number of buckets = 10

Range = 10



final Sorted array is

0·12	0·17	0·21	0·23	0·26	0·39	0·72	0·78	0·94

Algorithm:

Sequence of steps required to perform a particular task (or) solving a problem

⟶ from the sequence of steps, we select an optimal solution

Algorithm Analysis:

To analyze a particular program algorithm, we need to understand for which input the algorithm takes less time and for which input it takes more time.

* The performances of a algorithms can be measured on the scales of "time" and "space". The performance of a program is the amount of computer memory

and time needed to run a program.

→We use two approaches to determine the performance of a program

i, Analytical

ii, Experimental

→In performance analysis we use analytical methods, while in performance measurement we conduct experiments

*Based

* Algorithm Analysis is based on

i, Time complexity

ii, Space complexity

Time Complexity:

The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program.

→In other words, it is the amount of time it needs to run to compile

Space Complexity:

The space complexity of an algorithm or a program is a function of the space needed by the algorithm or program to run

→A program that saves space over a competing progr-am is considerable desirable

Note:

Based on time complexity, we divide the inputs in three cases

1) Best case
2) Worst case
3) Avergge case.

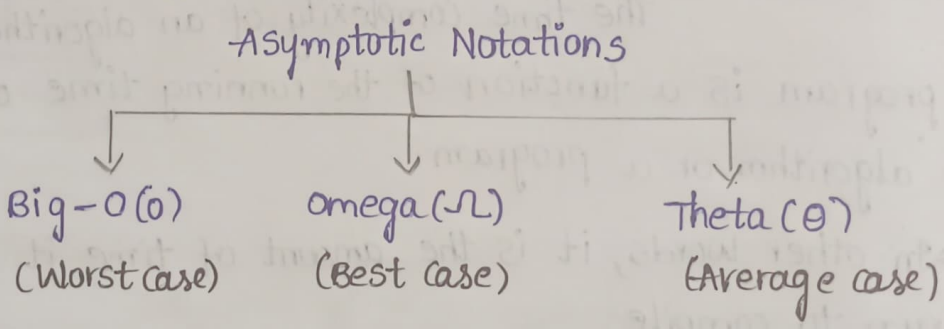
1) Best case: Where we assume the input, for which algorithm takes less time.

2) Worst case: Where we assume the input, for which algorithm takes long time.

3) Average case: Where the input lies in between best case and Worst case.

Asymptotic Notations:

Asymptotic Notations to calculate the Asymptotic Notations are used to calculate the running time complexity of an algorithm



Big O Notation (O):

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the Worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

Omega Notation (Ω):

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or

the best amount of time an algorithm can possibly take to complete.

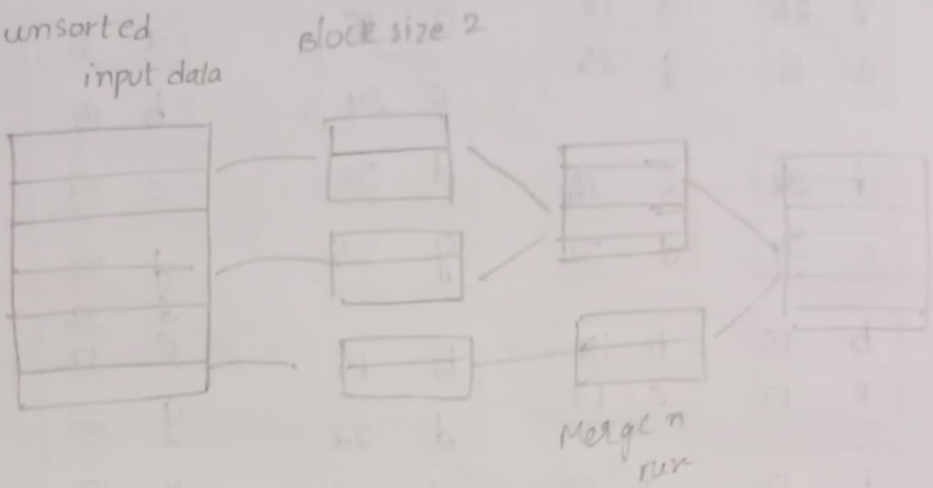
Theta Notation (Θ):

The notation $\Theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

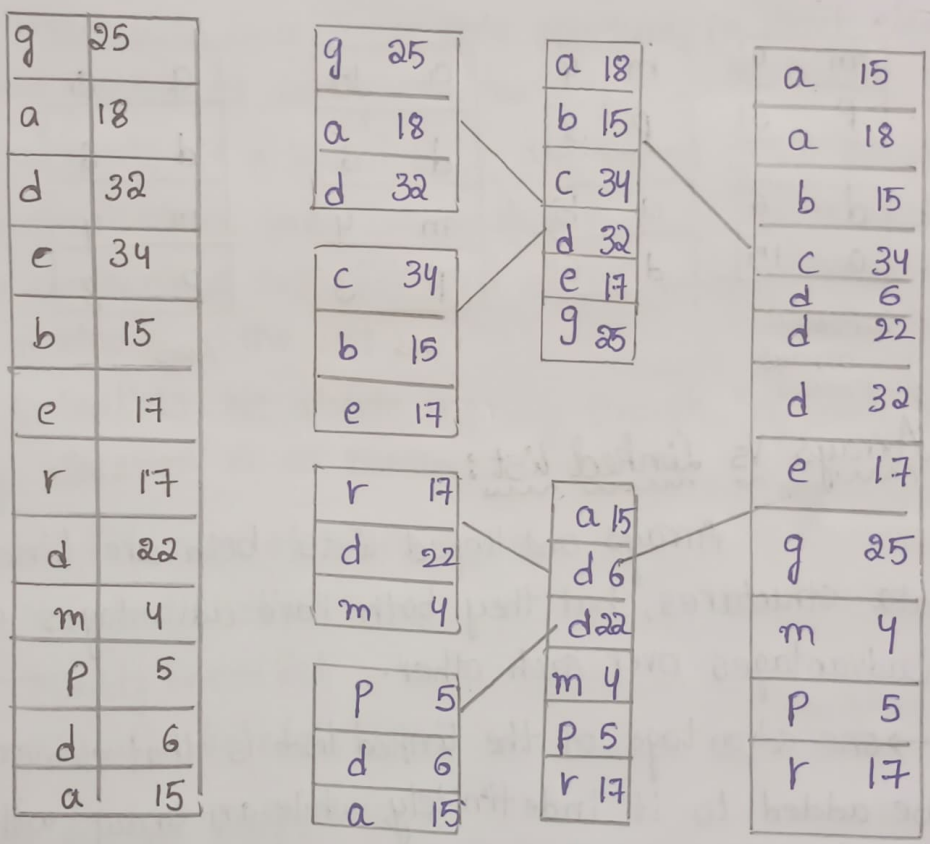
Common Asymptotic Notations:

constant	$O(1)$
linear	$O(n)$
quadratic	$O(n*n)$
cubic	$O(n^3)$
logarithmic	$O(\log n)$
n log n	$O(n \log n)$
factorial	$O(N!)$

External Sorting:



Example: Block size is 3

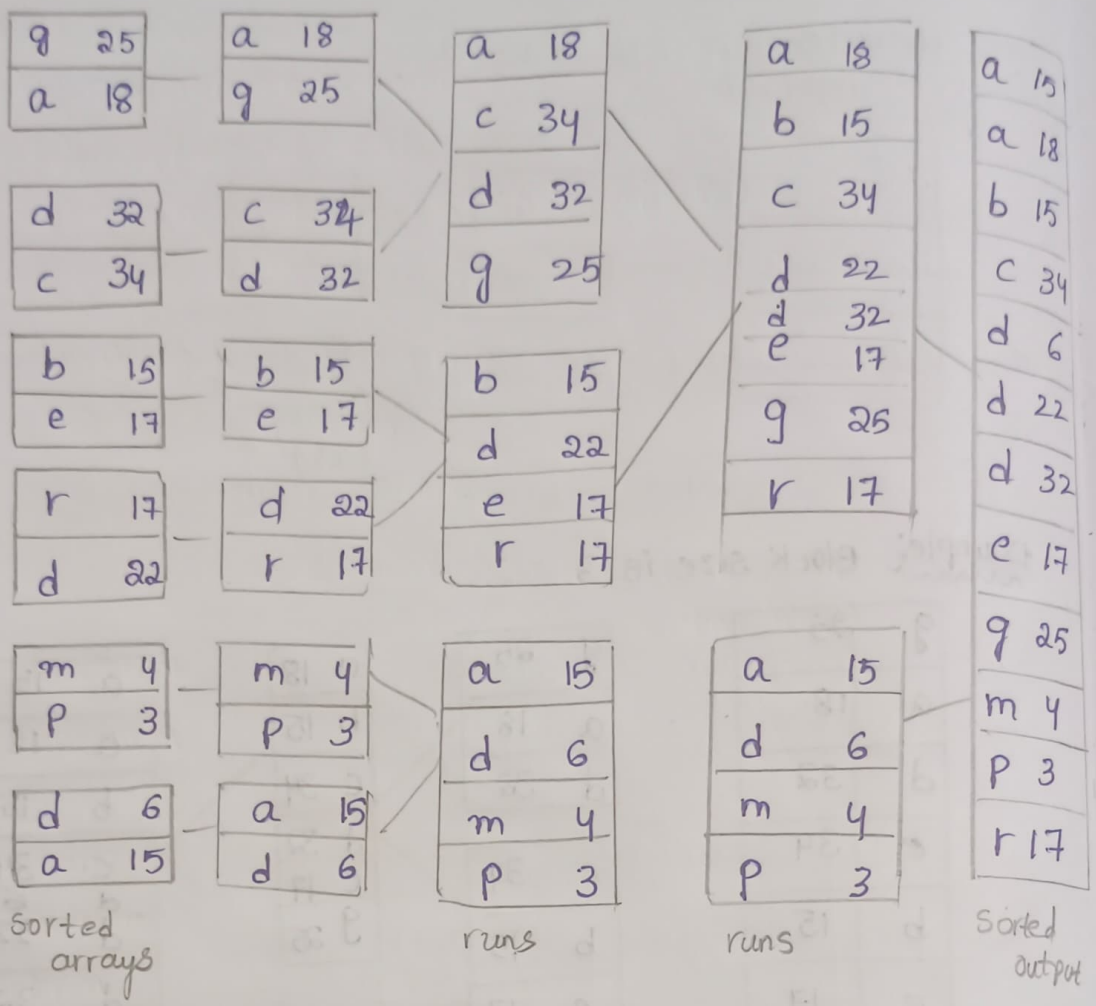


External Sorting:

External Sorting is a class of sorting algori-thms that can handle massive amounts of data

→ External Sorting is required when the data being stored do not fit into the main memory of a computer device (RAM) and instead it requires external memory (hard disk)

Block size 2.



Arrays Vs Linked list:

Arrays and linked lists both are linear data structures, but they both have advantages and disadvantages over each other.

→ one advantage of the linked list is that elements can be added to it indefinitely, while an array will eventually get filled or have to be resized.

→ Elements are also easily removed from a linked list whereas removing elements from an array leaves empty spaces that are a waste of a computer memory.

Major differences between Arrays and Single linked list :

* Size
* memory allocation
* memory efficiency
* execution time
* Dynamic allocation
* ease of insertion/ deletion

Linked Lists:

* A linked lists is a linear data structure, in which elements are not sorted at contiguous memory locations.
* The elements in a linked lists are linked using pointers.
* In simple words, linkedlist consists of nodes where each node contains a data field and a self reference (link) to the nextnode in the list.
* In linkedlists, We allocate memory through "Dynamic memory allocation" or at runtime

  linked lists are of three types
  - —>Single linkedlist
  - —>Doubly linked list
  - —>Circular linked list

Single Linked Lists:

—>Single linked list takes place on forward navigation only

—>A linked list is represented by a pointer to the first node of the linkedlists.

—>The first node is called "head". If the linkedlist is empty then the value of headpoints to "NULL".

Operations on single linked list:

1. Insertion
   a) Insertion at beginning
   b) Insertion at end
   c) Insertion at given position (middle)
2. Deletion.
   a) Deletion at beginning
   b) Deletion at end
   c) Deletion at middle.
3. Display/traverse (visiting each and everyone at a single linked list)
4. Maximum
5. Minimum
6. Sum
7. Count
8. Search
9. Even

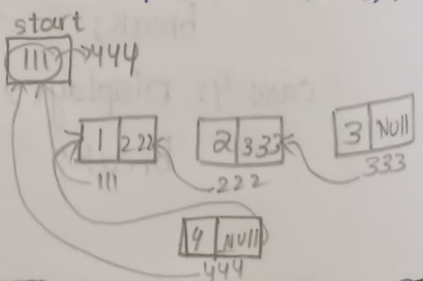
Programm:

```
#include<stdio.h>
struct node
{
  int id;
 Struct node *next;
}
 *start= NULL ;*temp ,*ptr;
   void insert at beginc );
   void insert at end( );
   void insert at middle( );
   void display( );
   void maximum ( );
   void minimum ( );
   void Sum ( );
```

```
Void count();
void Search();
Void even();
Void create();
Void delete at begin();
Void delete at middle();
Void delete at end();
Void main()
{
  int data, i, choice;
  Printf(**Menu**);
  While(1)
{
Printf("1. Insert at begin\n 2. Insert at end\n 3. Insert at
middle\n 4. Display\n 5. Maximum\n 6. minimum\n 7.
Sum\n 8. Count\n 9. search\n 10. even\n 11. create\n
12. delete at begin\n 13. delete at middle\n 14. delete at
end\n);
 Printf("....\n");
 printf("Enter the choice");
 Scanf("%d", &choice);
 Switch choice()
 {
  Case 1: Insert at begin()
        break;
  Case 2: Insert at end()
        break;
  Case 3: Insert at middle()
        break;
  Case 4: Display()
        break;
```

```
Case 5: maximum()
        break;
case 6: minimum()
        break;
case 7: Sum()
        break;
case 8: Count()
        break;
case 9: Search()
        break;
Case 10: even()
        break;
case 11: create()
        break;
case 12: delete at begin()
        break;
Case 13: delete at middle()
        break;
case 14: delete at end()
        break;
case 15: pri
default: printf("Invalid selection");
        break;
  }
}
```

```
→ void insert at begin (int value)
  {
    struct node *temp;
    temp= (struct node *) malloc (size of (struct node));
    temp→data= value;
    temp→next= NULL;
    if (start == NULL)
      { start = temp;
```



```
else
 {temp->next = start;
     start = temp;
  printf("one node is inserted");
 }

Void insert at end ( )
 {
  struct node *temp;
  temp = (struct node *)malloc (size of (struct node));
   temp -> data = value;
   temp -> next = NULL;
   if (start == NULL)
    {
     start = temp;
    }
  else
   { struct node *ptr = start;
    While (ptr -> next != NULL)
    {
      ptr = ptr -> next;
    }
    ptr -> next = temp;
   }
   printf(" one node is inserted");
  }

Void insert at middle ( )
 {
  struct node * temp;
 temp = (struct Node*)malloc (size of (struct node));
  temp -> data = value;
  temp -> next = NULL;
  if (start == NULL)
  {
   start = temp;
  }
 else
```

```
Struct Node *ptr= start;
While (Ptr -> data! = key)
{
  ptr = ptr -> next;
}
 Ptr -> next = temp -> next;
  Ptr -> next = temp;
  Printf ("one node is inserted");
}
Void display( )
{
  if (start == NULL)
  printf(" linked list is empty");
  else
    Struct Node *ptr = start;
    Printf(" elements are . . . \n");
    While (ptr -> next ! = NULL)
    {
      Printf("%d . . . ", ptr -> data);
      ptr = ptr -> next;
    }
    printf ("%d . . . . >", ptr -> data);
  }
Void maximum( )
  {
    Struct node *start;
    int max = 0;
     Ptr = start;
    While (ptr ! = NULL)
    {
      if (ptr -> id > max)
         max = ptr -> id;
         Ptr = ptr -> next;
    }
```

```
printf("%d", max);
}
void minimum( )
{
 struct node *start;
  int min=0;
   ptr= start;
  While (ptr != NULL)
 {
   if (ptr->id<min)
    min = ptr->id;
     ptr = ptr->next;
  }
printf("%d", min);
  }
void Sum( )
 {
  Struct node *start;
  int sum = 0;
  While (ptr != NULL)
  {
    sum = sum+ptr->data;
  }
  printf("%d", Sum);
  }
void count( )
 {
   if (start == NULL)
   printf("linked list is empty");
  else
    struct node *ptr= start;
```

```
While (ptr→next != NULL)
{
 printf ("%d ....", ptr→data);
      c++;
    ptr = ptr→next;
 }
 printf ("%d ...", c);
 }
→void search()
  {
   Struct node * start;
   int key, count=0;
   printf ("Enter the elements to be searched");
   scanf ("%d", key);
    ptr = start;
   While (ptr!=NULL)
   { if (ptr→data = key)
     {
       count++;
       break;
     }
    ptr = ptr→next;
    }
    if (count==0)
    printf ("element is found");
    else
       Printf ("element is not found");
    }
→void evenc()
  {
   if (start == NULL)
   printf ("linked list is empty");
   else
```

```
struct node *ptr = start;
printf("elements are ....\n");
if (ptr -> data % 2 == 0)
  ptr = ptr ->
  printf("%d", elements are even);
else
   printf(" elements are odd");
   ptr = ptr -> next;
}
```

```
Void create ()
{
 struct node *start;
 int n;
 scanf("%d ", &n);
 temp = (struct node*) malloc (size of (struct node));
  temp -> data = n;
 if (start == NULL)
   {
    start = temp;
    temp -> next = NULL;
   }
else
    {
      ptr = start;
     while (ptr -> next != NULL)
     {
       ptr = ptr -> next;
     }
    ptr -> next = temp;
    temp -> next = NULL;
   }
}
```

```
Void delete at begin ( )
{
 if (start == NULL)
 printf("L.L is empty");
```

```
else
  Struct node *ptr = start;
  if (start -> next = NULL)
    {
      start = NULL;
      free (ptr);
    }
  else
      Start = start ->next;
      free (ptr);
    Printf ("one node is deleted");
    }

-> Void delete at end ( )
    {
      if (start = = NULL)
      Printf (" L.L is empty ");
      else
          Struct node *ptr1 = start;
           Struct node *ptr2;
        if (start -> next = = NULL)
          {
            Start = NULL;
          } free (ptr1);
        else
        While (ptr1 -> next ! = NULL)
        {
          ptr2 = ptr1;
          ptr1 = ptr1 -> next;
        }
         ptr2 -> next = Null;
           free (ptr1);
        Printf ("one node is deleted");
      }
```

```
Void delete at middle ( )
{
  if (start -> next == NULL)
  {
    start = NULL;
    free (ptr1);
  }
  struct node *ptr1 = start;
  struct node *ptr2;
  While (ptr -> data != key)
  {
    ptr2 = ptr1;
    Ptr1 = ptr1 -> next;
  }
  ptr2 -> next = ptr1 -> next;
    free (ptr2);
  printf ("one node is inserted");
  }
```

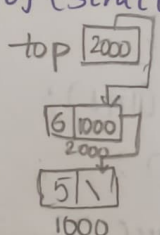
Stacks:

Stack is the last in first out linear data structure where the insertions and deletions takes place from a single end known as top (Peak)

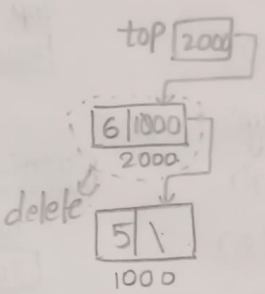
Stacks using Linked Lists:

```
#include<stdio.h>
Struct node
{
  int data;
  struct node *next;
}
  *top = NULL;
  Void push (int);
  Void pop ( );
  Void display ( );
```

```
Void main()
{
 int choice, value;
 printf("\n: Stack using Linked List :\n");
  While(1)
  {
   Printf("\n*** MENU ***\n");
   Printf("1.push\n 2.pop\n 3.Display\n 4.Exit\n");
   Printf("Enter your choice:");
   Scanf("%d", &choice);
   Switch(choice)
   {
   case1: printf("Enter your value to be insert:");
          Scanf("%d", &choice);
          Push(value);
          break;
   Case 2: Pop();
           break;
   Case 3: display();
           break;
   Case 4: exit(0);
   default: Printf("\n Wrong Selection!\n");
  }
 }
}
Void push(int value)
{
  struct node *temp;
  temp=(Struct node*)malloc(size of(struct node));
  temp → data = value;
  temp → next = NULL;
  if (top==NULL)
```



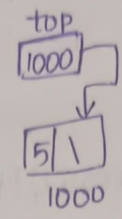
```
    temp -> next = NULL;
else
    temp -> next = top;
    top = temp;
    Printf("\n Insertion is Success !!!\n");
}
void pop()
{
  if (top==NULL)
  printf("\n Stack is Empty !!!\n");
 else
  {
   Struct node * ptr = top;
   Printf("\n Deleted element: %d", temp->data);
   top = ptr->next;
   free(ptr);
  }
}
void display()
{
 if (top== NULL)
 printf("\n Stack is Empty !!!\n");
else
 {
  Struct node *ptr = top;
  While ( ptr->next != NULL)
  {
   Printf("%d ---> ", temp->data);
    ptr=ptr->next;
  }
    Print-f("%d ---> NULL", temp->data);
 }
}
```



* Example:

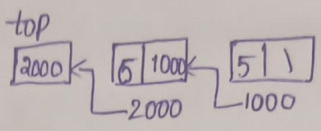
Push(5)
Push(6)
Push(8)
Pop()
display()
pop()
pop()
display()
push(7)
push(8)
Pop()
display()

Push(5)



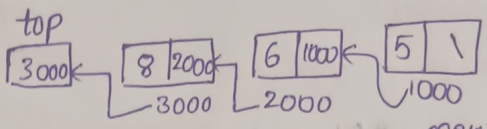
newnode → next = top
top = newnode

Push(6):



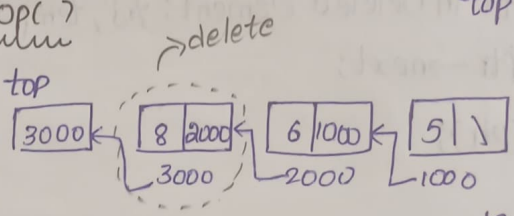
newnode → next = top
top = newnode

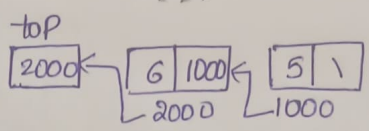
Push(8):



newnode → next = top
top = newnode

Pop()



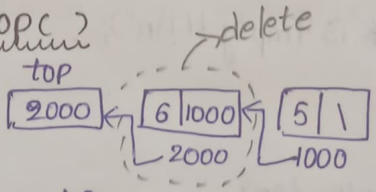


temp = top
top = top → next
free(temp)

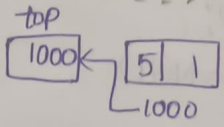
display:

Nodes are 5 and 6.

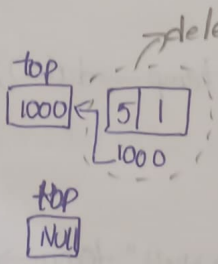
Pop()



temp = top
top = top → next
free(temp)



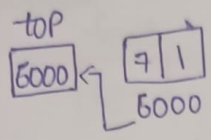
Pop()



display:

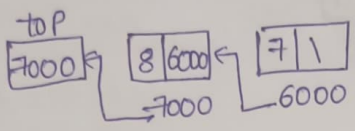
There are no nodes to display

push(7):



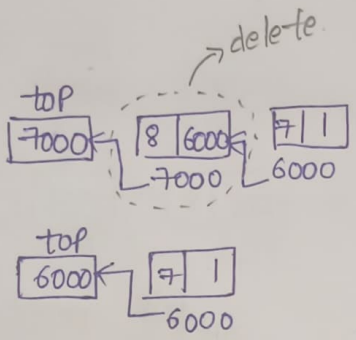
newnode → next = top
top = newnode

push(8):



newnode → next = top
top = newnode

pop():



display():

The node 7 is to be display.

Queues:

→ Queues follows first-in-first out methodology

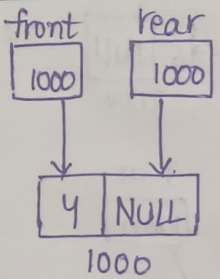
→ In Queues one end is always used to insert data (Enqueue) and other end is used to remove data (Dequeue)

→ The data item stored first will be accessed first.

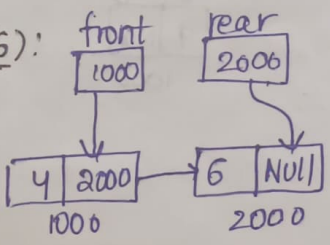
Example:

Enqueue(4)
Enqueue(6)
Enqueue(7)
Display
Dequeue()
Dequeue()
Enqueue(8)
Enqueue(9)
Display()

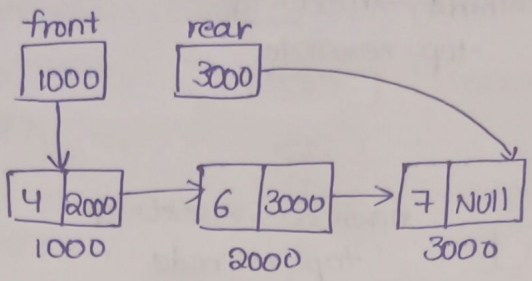
Enqueue(4):



Enqueue(6):



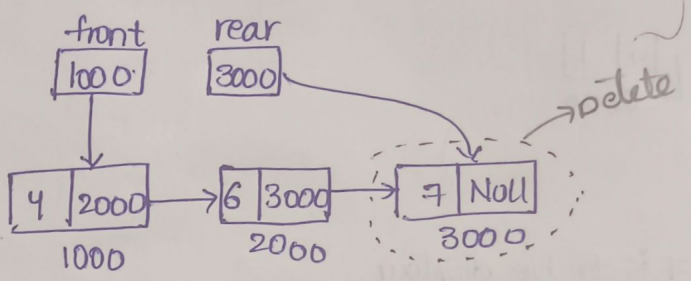
Enqueue(7):

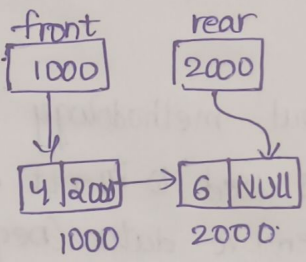


Display():

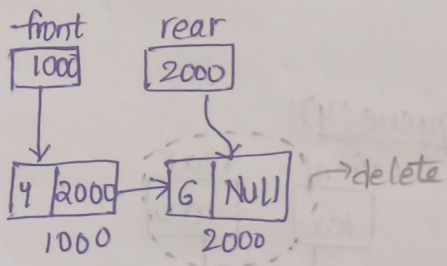
The data are 4, 6, 7

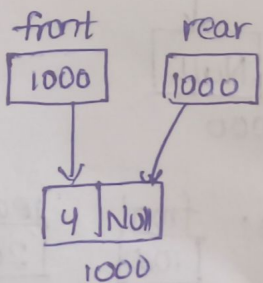
Dequeue():



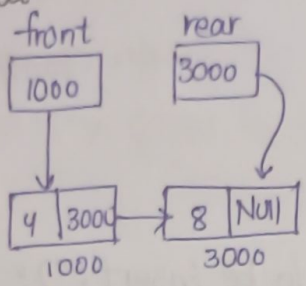


Dequeue():

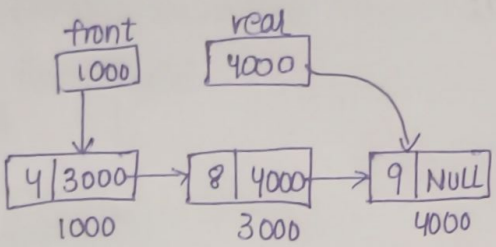




Enqueue(8):



Enqueue(9):



Display():

The data are 4, 8, 9

programm for Queues using linked lists:

```
#include<stdio.h>
Struct Node
{
 int data;
 Struct node *next;
}
 *front = NULL, *rear = NULL;
 Void insert (int);
 Void delete ( );
 Void display();
 Void main( )
 {
  int choice, value;
  Printf("\n:: Queue implementation using Linked list ::\n");
  While(1)
 {
  Printf("\n ***MENU*** \n");
```

```
Printf ("1. Insert \n 2. Delete \n 3. Display \n 4. Exit \n ");
Printf ("Enter your choice: ");
Scanf ("%d", &choice);
Switch (choice)
{
  case 1: printf ("Enter the value to be insert: ");
          scanf ("%d", &value);
          insert (value);
          break;
  case 2: dequeue ();
          break;
  case 3: display ();
          break;
  case 4: Exit (0);
          break;
  default: printf ("\n Wrong selection !\n");
  }
}
}

Void enqueue (int value)
{
  Struct Node *newNode;
  newNode = (Struct node*) malloc (size of (Struct Node));
  newNode → data = value;
  newNode → next = NULL;
  if (front == NULL)
    front = rear = newNode;
  else {
      rear → next = newnode;
      rear = newNode;
    }
  Printf ("In Insertion is success !!!\n");
}
```

```
Void dequeue()
{
  if (front == NULL)
  Printf("\n Queue is Empty!!!\n");
 else
    {
     Struct Node *temp=front;
     front= front ->next;
     Printf("\n Deleted element : %d\n", temp->data);
     free(temp);
    }
  }
void display()
{
  if (front == NULL)
  Printf("\n Queue is Empty !!!\n");
  else
     {
      Struct node *temp= front;
      While (temp->next != NULL)
      {
       printf("%d--->",temp->data);
       temp= temp->next;
       }
      Printf("%d--->NULL\n", temp->data);
     }
 }
```

→ push the divisors of 12 using linkedlists (stacks)

```
#include<stdio.h>
Struct node
{
 int data;
 Struct node *next;
 }
  *top=NULL;
   Void push(int);
   Void display( );
   int main( )
   {
    int n,i;
    Printf("enter the value of n\n":);
    Scanf("%d",&n);
    for(i=2;i<=n;i++)
   {
    if(n%i==0)
    {
      push(i);
    }
   }
 printf("divisors of n are \n");
  display( );
  return 0;
}
Void push(int value)
{
  Struct node *newnode;
 newnode=(struct node*)malloc(size of(struct node));
 newnode→data=value;
 if(top==NULL)
 {
  newnode→next=NULL;
```

```
}
else
{
 newnode->next=top;
}
top=newnode;
}
void display()
{
 if(top==NULL)
 {
  Printf("empty SLL");
 }
 else
 {
  struct node *temp=top;
  while(temp!=NULL)
  {
   Printf("%d",temp->data);
  temp=temp->next;
  }
 }
}.
```

Doubly Linked List:

* Doubly linked list is a complex type of linkedlist in which a node contains a pointer to the previous as well as the next node in the sequence.

* In doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the prerious node (previous pointer)

Creation of Node in doubly linked list:

```
struct node
{
 struct node *prev;
 int data;
 struct node *next;
};
struct node *head;
```