Introduction To Datastructure

Datastructure- It is not only the collection of data but also it includes the following things.

→ organisation of data
→ Relationship among the data
→ Operations performed on the data
→ Access methods to access the data

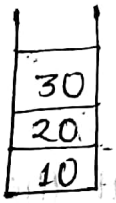
Types Of Datastructure -

→ Linear → Static
→ Non Linear → ~~Non-~~ Static Dynamic

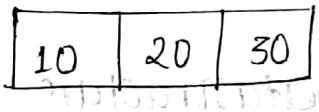
Linear Datastructure

Elements are organised in a linear fashion i.e sequential manner.

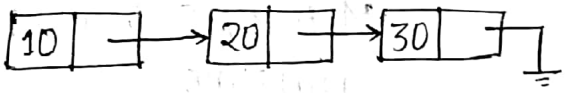
Exp- Stack



Queue



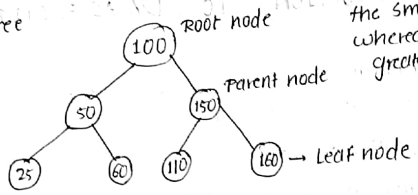
Linked List -



Non-Linear Datastructure

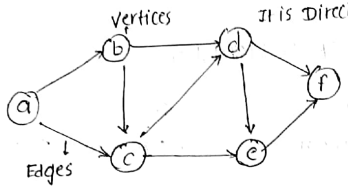
Elements are organised in non-linear fashion i.e they are not organised in sequential manner.

Exp- Tree



Always left side the smaller no. whereas right side greater no.

Graph



It is Directional graph

Static Datastructure -

Memory allocation at the time of compilation.
And once the memory is allocated it will not change.

Exp- Array, stack using array, Queue using array

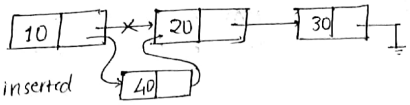
Dynamic Datastructure -

Memory allocation can be done at the time of runtime.
And once the memory is allocated it can be change.

Exp- Linked list, stack using linked list, Queue using linked list

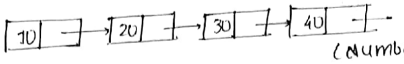
- Linked list: New element can be inserted in between
- stack using linked list: New element is inserted before i.e Elements get added at left side (before)
- Queue using linked list: New element is inserted after i.e element get added at right

Linked List



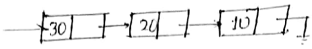
Element can be inserted in between

Queue



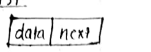
(Number/Elements are arranged in ascending)

Stack

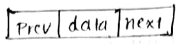


Elements are added these side (i.e arranged in descending here)

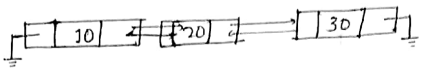
Linked List node



single LL (directional)



double LL (bidirectional)



Stack -

It is a linear datastructure in which elements are push and pop through one end called as top of stack (tos)

It follows LIFO policy i.e Last In First out.

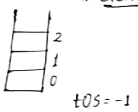
If tos = -1, stack is empty
If tos = Max-1, stack is full

define Max 50
↓
Macro

Stack operation -

#define MAX 50

MACRO or Macro defination



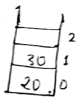

```
→ struct stack
  {
    int tos;
    int elems[MAX];
  } s;
```

```
→ int isEmpty()
  {
    if (s.tos == -1)
        return 1;
    else
        return 0;
  }
```

```
→ int isfull()
  {
    if (s.tos == MAX-1)
        return 1;
    else
        return 0;
  }
```

Push(20) (Adding new elems)

Push(30)




```
→ int Push()
  {
    enter data i.e x
    check stack is full
    i.e if (isfull())
      display stack is full;
    else
      s.elems[++s.tos] = x;
  }
```

POP() (Deleting elems)

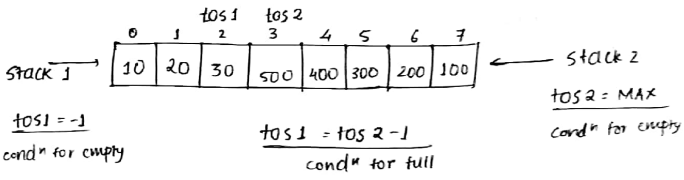
POP()

tos- indicates higher no. of elements present in tos

```
→ int POP()
  {
    check stack empty
    i.e
      if (isEmpty())
          display stack is empty
      else
          return -1;
    return s.elems[s.tos--];
  }
```

```
→ int show()
  {
    check stack empty
      i.e
        if (isEmpty())
            display stack is empty;
        else
        for (i=0; i<=s.tos; i++)
            display elems of
                s.elems[i];
  }
```

Two stacks in a single array -



tos1 = -1
cond^n for empty

tos1 = tos2 - 1
cond^n for full

tos2 = MAX
cond^n for empty

→ Struct stack

```
{
  int tos1, tos2;
  int elems[MAX];
} s;
```

→ int isEmpty1()

```
{
  if (s.tos1 == -1)
    return 1;
  else
    return 0;
}
```

→ int isEmpty2()

```
{
  if (s.tos2 == MAX)
    return 1;
  else
    return 0;
}
```

→ int isfull()

```
{
  if (s.tos1 == s.tos2 - 1)
    return 1;
  else
    return 0;
}
```

→ int push1()

```
{
  Enter data i.e x;
  Check stack is Full
  i.e if (isFull())
    display stack is full;
    else
    s.elems[++s.tos1] = x;
}
```

→ int POP1()

```
{
  check stack1 empty
  i.e
    if (isEmpty1())
    display stack1 is empty;
    else
    s.elems[s.tos1--];
}
```

→ int show1()

```
{
  check stack 1 empty
  i.e
    if (isEmpty1())
    display stack 1 is Empty;
  else
    for (i=0; i<=s.tos1; i++)
      display elems at
          s.elems[i];
}
```

→ int push2()

```
{
  Enter data i.e x;
  check stack is full
  i.e
    if (isfull())
    display stack is full;
    else
    s.elems[--s.tos2]=x;
```

→ int POP2()

```
{
  check stack2 empty
  i.e
    if (isEmpty2())
    display stack 2 is Empty;
    else
    s.elems[s.tos2++];
}
```

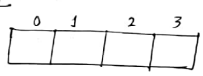
→ int show2()

```
{
check stack 2 is empty
i.e if (isEmpty2())
   display stack 2 is empty;
else
   for (i=MAX-1; i>=s.tos2;
                    i--)
    display elems at
          s.elems[i];
}
```

Queue -

It is a linear datastructure in which elements are inserted through one end called as rear and removed through one end called as front.

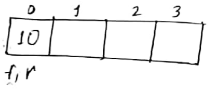
Initially



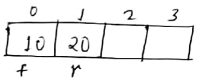
(front) f = 0

(rear) r = -1

insert (10)



insert (20)




```
→ Struct Queue
  {
   int r,f;
   int elems[MAX];
  } q;
```

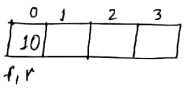
define MAX 50 ← Macro expansion
↑ Macro template
Macro

```
→ int isEmpty()
  {
   if ( q.r < q.f)
     return 1;
   else
     return 0;
  }
```

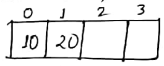
```
→ int isfull()
  {
   if (q.r == MAX-1)
       return 1;
    else
     return 0;
  }
```

```
→ int insert()
  {
    Enter data x;
   check queue is full;
  i.e if (isfull())
       display queue is full;
     else
       q.elems[++q.r] = x;
  }
```

insert (10)

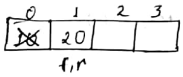


insert (20)

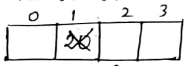



```
→ removes()
  {
  Check queue is empty
  i.e if (isEmpty())
      display queue is empty;
    else
     q.elems[q.f++];
  }
```

removes()



removes()




```
→ show()
  {
   Check queue is Empty
   i.e if (isEmpty())
          display queue is empty;
     else
       for (i = q.f; i <= q.r; i++)
         display elems at
              q.elems[i];
  }
```

Program for stack - (Menu driven program)

```
void main ( )
{
 int ch, x;
 s.tos = -1;
 do
 {
  printf (" Enter 1. Push 2. POP 3. show 4. Exit");
  printf (" Enter the choice");
  scanf (" %d", &ch);
  switch (ch)
  {
   case 1 : push (x);
            break;
   case 2 : x = pop ( );
            if (x == -1)
              stack is empty
            else
              printf (" %d is removed", x);
              break;
   case 3 ; show ( );
              break;
   case 4 ; Exit (0);
              break;
  }
 } while (ch != 4);
 getch( );
```

Applications of Stack -

Prefix (+ 21)

→ Infix to postfix conversion
↓ (operator comes between two operants) (2+1)
↓ (21+)

→ Postfix expression evaluation

→ Removal of recursion

→ Decimal to binary conversion

→ Reverse of a string

1) Infix to Postfix conversion-

↦ In infix expression, operator lies in between operands, e.g - A+B

↦ In postfix expression, operator lies after the operands, e.g - AB+

Operator	Priority	Associativity
(	1 (highest)	L to R
(Raise to) ↑	2	R to L
*, /	3	L to R
+, −	4	L to R

Exp - a + b * c

current symbol	stack	postfix exp^n
a	Empty	a
+	+	a
b	+	ab
*	+ *	ab
c	+ *	abc
EOS End of Symbol	empty	abc * +

↦ (A + B) * C - D/E * (F/G)

current symbol	stack	postfix exp^n
(	(	empty
A	(	A
+	(+	A
B	(+	AB
)	empty	AB +
*	*	AB +
C	*	AB + C
-	-	AB + C *
D	-	AB + C * D
/	- /	AB + C * D

current symbol	stack	postfix exp
E	- /	AB + C * DE
*	- *	AB + C * DE /
(	- * (	AB + C * DE /
F	- * (	AB + C * DE / F
/	- * (/	AB + C * DE / F
G	- * (/	AB + C * DE / FG
)	- *	AB + C * DE / FG /
EOS	Empty	AB + C * DE / FG / * -

→ Don't consider priority of brackets for this case.

↦ a ↑ b * c - d + e/f / (g+h)

cs	stack	postfix exp
a	empty	a
↑	↑	a
b	↑	ab
*	*	ab ↑
c	*	ab ↑ c
-	-	ab ↑ c *
d	-	ab ↑ c * d
+	+	ab ↑ c * d -
e	+	ab ↑ c * d - e
/	+ /	ab ↑ c * d - e

CS	Stack	Postfix exp"
f	+ /	ab↑c * d-ef
/	+ /	ab↑c * d- ef/
c	+ / c	ab↑c * d-ef/
g	+ / (	ab↑ c * d-ef/g
+	+ / (+	ab↑c * d-ef/g
h	+ / (+	ab↑ c * d-ef/gh
)	+ /	ab↑c* d-ef/gh+
EOS	empty	ab↑ c * d -ef/gh+/+

Algorithm

Step 1 - Start

Step 2 - Let S be the empty stack i.e tos = -1

Step 3 - Read the infix expression as an array of characters

Step 4 - Analyse each character of infix expression and perform the following steps till end of input.

i) Let cs be the next input character.

ii) If (cs is an operand)

copy cs to postfix expression

else

{

if (stack is empty)

push (s, cs)

else

{

check priority of stack top and cs

if (priority of stack top >= priority of cs)

{

ts = pop(s)

(target symbol)

add ts to postfix expression

}

else

push (s, cs)

}

Step 5 - if (cs = = '(')

push (s, cs)

if (cs = = ')')

pop operators between (&)

and add to the postfix expression

Step 6 - While (stack is not empty) ~~tos = pos~~ // input exp" finish

{

ts = pop (s)

add ts to the postfix expression

}

Step 7 - Display postfix expression

Step 8 - stop

(A + B + C) + C

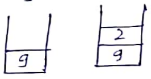
)

2) Evaluation of postfix expression

↦ In postfix expression, operator lies after the operands, e.g- AB+

Exp- ↦ 9, 2, /, 3, −

9 2 /

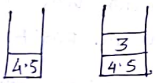


op 2 = pop(s) i.e 2
op 1 = pop(s) i.e 9

value = op1 cs op2
= 9 / 2
= 4.5

push (s, value)

3 −



op2 = 3
op1 = 4.5

value = 4.5 - 3
= 1.5

push (s, value)

∵ End of i/p

value = pop(s)

value of postfix expression
= 1.5

↦ 5, 6, 2, +, *, 12, 7, /, −

5 6 2 +

op 2 = pop(s) i.e 2
op 1 = pop(s) i.e 6

value = op1 cs op2
= 6 + 2
= 8

push (s, value)

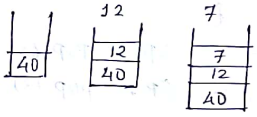
*

op2 = 8
op1 = 5

value = 5 * 8
= 40

push (s, value)

12 7 /

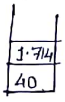


op 2 = 7
op 1 = 12

value = 12 / 7
= 1.714

push (s, value)

−



op 2 = 1.714
op 1 = 40

value = 40 - 1.714
= 38.286

∵ End of i/p

value = pop(s)

value of postfix exp^n = 38.286

= 38.3

Algorithm

Step 1 - start

Step 2 - Let s be the empty stack i.e tos = -1

Step 3 - Read the postfix expression as an array of characters

Step 4 - Analyse each character and perform the following steps till end of input

i) Let cs be the next input character

ii) IF (cs is an operand)

push (s, cs)

else

{

op 2 = pop (s)

op 1 = pop (s)

value = op1 cs op2

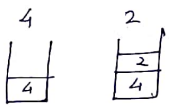
Push (s, value)

}

Step 5 - pop the value from stack and display it.

Step 6 - Stop

↦ 4, 2, ↑, 3, *, 3, -, 8, 4, /, 1, 1, +, /, +



↑

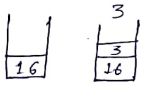
op 2 = 2

op 1 = 4

value = 4^2

= 16

Push (s, value)



*

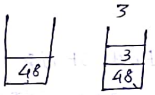
op 2 = 3

op1 = 16

value = 16 * 3

= 48

push (s, value)



-

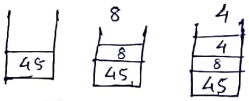
op 2 = 3

op1 = 48

value = 48 - 3

= 45

push(s, value)



/

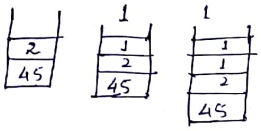
op 2 = 4

op 1 = 8

value = 8 / 4

= 2

push (s, value)



+

op 2 = 1

op 1 = 1

value = 1 + 1

= 2 push (s, value)

/

op 2 = 2
op 1 = 2

value = 2/2
= 1

push (S, value)

+

op 2 = 1
op 1 = 45

value = 45+1
= 46

push (S, value)

∵ End. of i/p
value = pop(S)
Value of postfix exp^n
= 46

3) Removal of Recursion -

↦ A function that calls itself until base condition is satisfied is called as recursion.

↦ Let us consider to find factorial of a number.

```
→ int fact (int x)
  {
    if (x == 0 || x == 1)
        return 1;
    else
      return (x * fact (x-1));
  }
```

x=4

fact(4)
4 * fact(3) 4*6=24
3 * fact(2) 3*2=6
2 * fact(1) 2*1=2

Removal of recursion is possible by using stack -

Algorithm -

Step 1 : start

Step 2 : Let S be the empty stack i.e tos=-1

Step 3 : Enter number n

Step 4 : push the number onto the stack till n>1

```
→ while (n>1)
  {
    push(n)
    n--
  }
```

Step 5: Initialize fact=1

Perform pop operation till stack empty

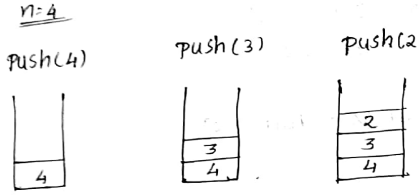
→ while (! stack is empty)

fact = fact * POP()

Step 6: Display factorial of a number.

Step 7: Stop

Exp - n=4



n=n-1 = 3 | n=2 | n=1

∴ stop push opⁿ

fact = 1

pop i.e. 2 | pop i.e 3 | pop i.e 4

∴ fact = 1 * 2 = 2 | ∴ fact = 2 * 3 = 6 | ∴ fact = 6 * 4 = 24

∴ stack is empty

∴ stop pop opⁿ

∴ factorial is 24

4) Decimal to Binary conversion-

Algorithm-

Step 1: start

Step 2: Let s be the empty stack i.e tos = -1

Step 3: Enter number n

Step 4: push r onto the stack till q=0

```
while (n! =0)
{
  r = n % 2
  push(r)
  n = n/2
}
```

Step 5: perform pop operation till stack is empty

```
while (! stack is empty)
  pop()
```

Step 6: Display

Step 7: Stop

Exp - n = 13

r = n % 2

r = 13 % 2 = 1

Push (1)

r = 6 % 2

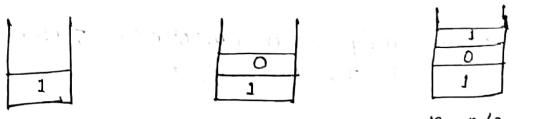
r = 0

Push (0)

r = 3 % 2

r = 1

Push (1)



n = n/2 i.e n = 13/2 = 6 | n = n/2 i.e n = 3 | n = n/2 ⇒ n = 3/2 = 1

r = n%2
= 1%2

r = 1

push(1)

Now q = 0

∴ stop push OP"

POP()

⇒ 1

POP()

⇒ 11

POP()

⇒ 1101

$(13)_{10} = (1101)_2$

5) Reverse of a string

* Algorithm -

Step 1: Start

Step 2: Let s be the empty stack i.e tos = -1

Step 3: Enter the string and read the characters of string

Step 4: Analyse each character and perform the steps till the end of input

i) Let cs be the input character

ii) while (string [i] != 0)

{

cs = string [i]

push (cs)

i++

}

Step 5: pop the value from stack and display it

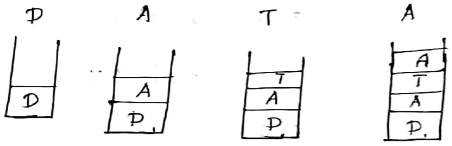
→ while (! stack is empty)

POP ()

Step : stop

Exp - DATA

D A T A



POP()

⇒ ATAD

6) well firmness of paranthesis

Algorithm-

step 1: start

Step 2: Let s be the empty stack i.e tos = -1

Step 3: ~~Enter~~ Read the paranthesis

Step 4: Analyse each paranthesis and perform the following steps -

i) Let cs be the input character

ii) while (~~string~~ cs != /0)

```
{
  if (cs == '(')
      push (cs)
  else
      if (stack is empty)
          display error
      else
          pop()
}

if (EOS && ! stack empty)
      display error
  else
      Accept
```

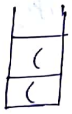
step 5: stop

Exp - (( ) ( ))

'(' push ((

cs = (push

((pop



cs =) pop

pop ( )

cs = (push

(pop ( )

cs =) pop

pop ( )

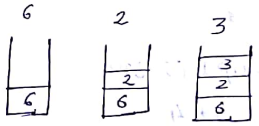
cs =) pop

pop ( )

stack is empty

↦ Evaluate the following expression by using stack -

→ 6, 2, 3, +, −, 3, 8, 2, /, +, *, 2, ↑, 3, +



+

op2 = pop() i.e 3

op1 = pop() i.e 2

value = op1 cs op2

= 2 + 3

= 5

Push (s, value)

-

op2 = 5

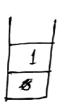
op1 = 6

value = op1 (s op2

= 6 - 5

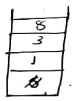
= 1

push (s, value)

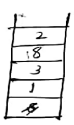


3

8



2



/

op2 = 2

op1 = 8

value = op1 (s op2

= 8 / 2

= 4

push (s, value)

+

op2 = 4

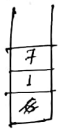
op1 = 3

value = op1 (s op2

= 3 + 4

= 7

push (s, value)



*

op2 = 7

op1 = 1

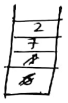
value = op1 (s op2

= 1 * 7

= 7

push (s, value)

2



↑

op2 = 2

op1 = 7

value = op1 (s op2

= 7 ↑ 2

= 49

push (s, value)

3

+

op2 = 3

op1 = 49

value = op1 (s op2

= 49 + 3

= 52

push (s, value)

pop()

value = 52

↳ Transfer the following expression to postfix and then evaluate by assuming a=1; b=2; c=3; d=4; e=6; f=6; g=1; i=3; j=3

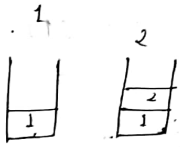
⇒ a + b - c * d / e + f $\uparrow$ g / (i + j)

Current symbol	Stack	postfix exp^n
a	Empty	a
+	+	a
b	+	ab
-	-	ab+
c	-*	ab+c
*	-*	ab+c*
d	-*	ab+c*d
/	-/	ab+c*d*
e	-/	ab+cd*e
+	+*	ab+cd*e/-
f	+	ab+cd*e/-f
↑	+↑	ab+cd*e/-f
g	+↑	ab+cd*e/-fg
/	+↑/	ab+cd*e/-fg↑
(	+/(	ab+cd*e/-fg↑
i	+/(	ab+cd*e/-fg↑i
+	+/(+	ab+cd*e/-fg↑i
j	+/(+	ab+cd*e/-fg↑ij
)	+/	ab+cd*e/-fg↑ij+
EOS	Empty	ab+cd*e/-fg↑ij+/+

postfix expression is - ab+cd*e/-fg↑ij+/+

X, Y, Z, A, B, C,

⇒ 1, 2, +, 3, 4, *, 6, /, -, 6, 1, ↑, 3, 3, +, /, +



+

op2 = 2

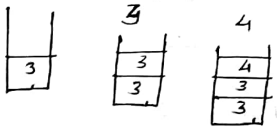
op1 = 1

Value = op1 (s) op2

= 1 + 2

= 3

Push (s, value)



+

op2 = 4

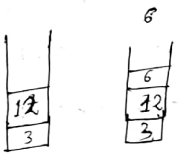
op1 = 3

Value = op1 (s) op2

= 3 * 4

= * 12

Push(s,value)

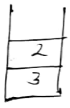


/

value = 6/7

Value = 12/6

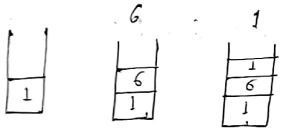
= 2



−

Value = 3-2

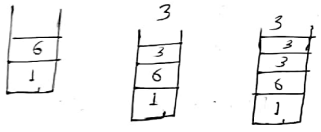
= 1



↑

Value = 6 ↑ 1

= 6



+

Value = 3+3

= 6

/

value = 6/6

= 1

+

Value = 1+1

= 2

POP ()

value = 2

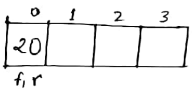
Circular Queue -

Let us consider Linear queue with max = 4

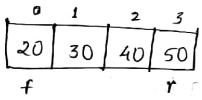
f = 0

r = -1

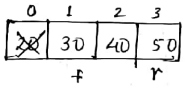
insert (20)



insert (30), insert (40), insert (50)



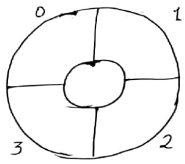
remove ()



we cannot insert new element into the queue even if there is an empty place for a new element because full condition of Queue is satisfied i.e $r = max - 1$

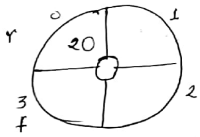
To overcome these circular queue can be used in which two ends of a queue are joined together to form a ring.

→ Circular Queue



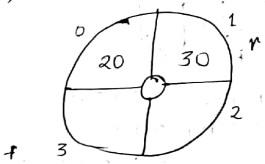
f, r

insert (20)

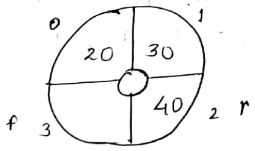


No element should be added where there is cl f(front)

insert (30)

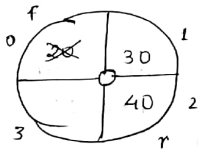


insert (40)

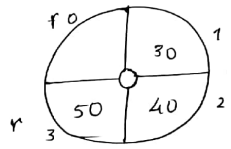


rear = (rear + 1) % Max

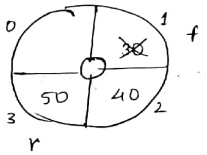
remove ()



insert (50)



remove()



→ only the drawback is just the one place remains empty (i.e gets wasted) otherwise complete Queue is filled.

Algorithm-

→
```
# define max 50
struct cqueue
{
  int r, f;
  int elems[MAX];
} cq;
```

(macro is processed at time of compile)

(Functn is processed at time of run)

→ int isEmpty()

```
{
  if (cq.r == cq.f)
      return 1;
  else
      return 0;
}
```

→ int isfull()

```
{
  if (cq.r == cq.f)
      return 1;
  else
      return 0;
}
```

→ void insert (int x)

```
{
```

① update rear

i.e

cq.r = (cq.r + 1) % MAX

② if (isfull())

display "cqueue is full"; //queue overflow

else

③ cq.elems [cq.r] = x

```
}
```

→ int removes()

```
{
```

① if (isempty())

```
  {
    return -1; //cqueue underflow
  }
  else
```

update front

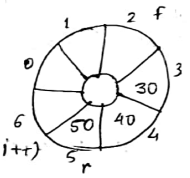
② i.e cq.f = (cq.f+1) % MAX

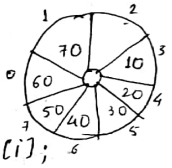
③ return (cq.elems [cq.f]);

```
}
```

→ void display()

```
{
  if (isempty())
  {
    display "cqueue is empty";
  }
  else
  {
    if (cq.front < cq.r)
    {
      for (i = cq.f+1; i <= cq.r; i++)
        display elements at cq.elems [i];
    }
    else
    {
      for (i = cq.f+1; i <= MAX-1; i++)
        display ele. at cq.elems [i];
```





```
    for (i=0; i<=cq.r; i++)
        display elems at cq.elems[i];
    }
  }
}
```

C programming -

→ Sum of digits -

```
# include <stdio.h>
# include <conio.h>
void main()
{
 int n, r, sum=0; clrscr();
 printf("Enter the number");
 scanf("%d", &n);
 while (n>0)
 {
    r = n%10;
   sum = sum+r;
    n = n/10;
  }
  printf("sum of digit is %d", sum);
 getch();
}
```

→ Fibonnacci series -

```
# include <stdio.h>
# include <conio.h>
void main()
{
 int a, n, a,b,c a=0, b=1, c;
 clrscr();
 printf("Enter the number");
 scanf("%d", &n);
 while (n>0)
 {
   a = b;
   b = c;
   c = a+b;
 }
 printf("Fibbonacci series is %d", c);
 getch();
}
```

→ Prime number -

```
# include <stdio.h>
# include <conio.h>
void main()
{
 int n,
 clrscr();
 printf("Enter the number");
 scanf("%d", &n);
```

```
while
while (
{
 for(j=2; j<=n-1;i++)
   {
    if (n%.i!=0)
       flag=1;
       break;
    }
 else
   if(flag==1)
      It is prime no.;
   else
      It is not a prime no.;
 }
```

For 2 marks -

Q.1) Explain stack with its operation.

2) Explain Queue with its operation.

3) Why circular Queue is used.

4) Differentiate between stack and Queue.

5) Types of Datastructure.

6) Explain Recursion.

7) Explain ADT (Abstract Datatype) with example.

8) What is priority Queue and its types.

9) Explain time complexity.

10) What are the applications of stack.

11) What are the characteristics of algorithm.

For 5 marks -

1) stack using array. (Algorithm)

2) Queue using array.

3) Circular Queue using array.

4) Infix to postfix conversion.

5) Postfix expression evaluation.

6) Assymptotic notations.

ADT (Abstract Datatype)

- Abstract means to hide the details from the user.
- In context of datastructures there are various types of data which are stored into datastructure and purpose of each data structure is different.

For exp - Queue can be used for job scheduling.
Graph can be used to find shortest distance between two cities.
Stack can be used to find expression value and so on.

- Not all users are interested about how things are happen rather they expected final result.
- ADT means what operations datastructures provide rather how these operations give final result.

Exp - Stack ADT

```
int isEmpty(s) // check whether stack is empty
int isFull (s) // check whether stack is full
void push(s,x) // insert x into the stack
int POP(s)     // Removes the top element from the stack
void show (s)  // Display elements from the stack
int stacktop(s) // Returns the top element from stack
```

Queue ADT

```
int isEmpty(q) // check whether Queue is empty
int isFull(q) // check whether Queue is full
Void insert (q,x) // insert x into the queue
int removes (q) // Removes the element from the queue
void display (q) // Display elements from the queue
```

Characteristics of Algorithm-

- Algorithm is a step-by-step procedure to solve a given problem.
- It is a blueprint of a program that map inputs to output

Characteristics -

- Input - It represents input data given by user externally.
- output - It represents the result given by program after successful completion.
- Finiteness - An algorithm must be terminated after finite number of steps.
- Definiteness - An algorithm must contain clear, precise, and unambiguous statement.
- Effectiveness - It represents time complexity and space complexity of an algorithm must be as low as possible

Exp - Let us consider addition of two numbers

Step1 : start

Step2 : Accept two numbers a and b

Step3 : c = a+b

Step4 : print c

Step5 : stop

characteristics -

→ There are two inputs a and b

→ Addition of a and b is c

→ Algorithm terminate in 5 steps

→ Above algorithm contains clear, precise and unambiguos statement

→ Time complexity is O (1) and space complexity is 6 bytes
(Big-oh)

Priority Queue -

→ In linear Queue elements are inserted through rear end removed through front end and it follows first in first out (FIFO) policy

→ In priority Queue elements are inserted through rear end but elements are removed based on priority.

There are two types -

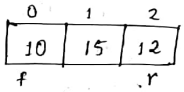
↦ Ascending priority Queue

↦ Descending priority Queue

1) Ascending priority Queue -

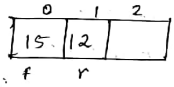
→ Elements are inserted through rear end but removed in ascending order.

For exp -



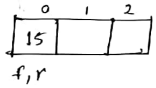
removes()

10 removed



removes()

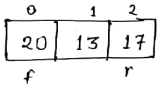
12 removed



2) Descending priority Queue -

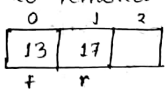
→ Elements are inserted through rear end by removed in descending order.

For exp -



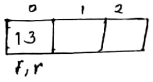
removes()

20 removed



removes()

17 removed



Asymptotic Notations -

1) Big - Oh (O)

Let f(n) and g(n) be the two functions, we can say that f(n) is big-oh (O) of g(n)

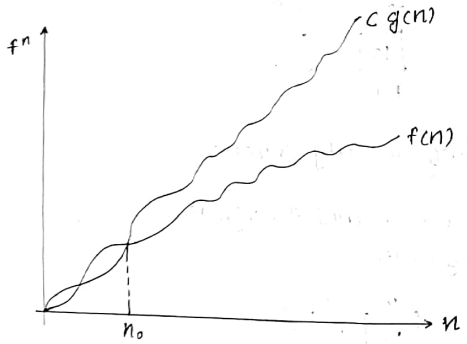
i.e $f(n) = O(g(n))$

If it satisfies the following condition -

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

where c is the constant and
n be number of inputs such that
$n > n_0$

Graphical Representation -



Exp - Let

$f(n) = 2n^2 + n + 100$

$\because 2n^2 + n + 100 < \overset{c\ g(n)}{3n^2}$

$\therefore f(n) = O(n^2)$

2) Big - Omega (Ω)

Let f(n) and g(n) be the two functions, we can say that f(n) is big-omega (Ω) of g(n)

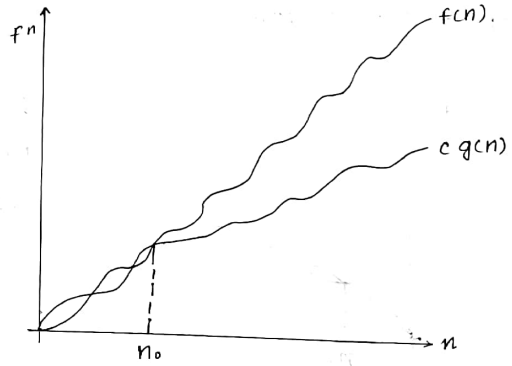
i.e $f(n) = \Omega(g(n))$

If it satisfies the following condition -

$f(n) \geq c\,g(n)$

where c is the constant and
n be the number of inputs such that
$n > n_0$

Graphical Representation -



Exp -

Let $f(n) = 2n^2 + n + 100$

$\because 2n^2 + n + 100 > \overset{c\ g(n)}{2n^2}$

$\therefore f(n) = \Omega(n^2)$

3) Big-Theta (Θ)

→ It is a combination of both big-oh (O) and big-omega (Ω) notations.

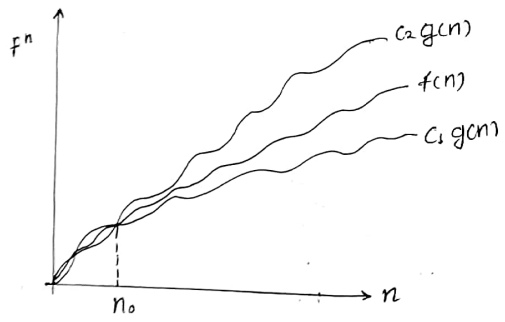
→ Let f(n) and g(n) be two functions, we can say that f(n) big-Theta (Θ) of g(n) $f(n) = \Theta(g(n))$

If is satisfies following condition -

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

where c_1 and c_2 are constants and n be number of inputs such that $n > n_0$

Graphical Representation-



Exp - Let

$f(n) = 2n^2 + n + 100$

$c_1 g(n) \quad c_2 g(n)$

$\therefore 2n^2 < 2n^2 + n + 100 < 3n^2$

$\therefore f(n) = \Theta(n^2)$

Time Complexity

→ It represents the time required by computer machine to execute a program, Accept the inputs from user and display output.

→ There are various notations used to calculate time complexity like -

- Big-oh (O)
- Big-Omega (Ω)
- Big-Theta (Θ)

Types -

1) Worst case TC -

→ It represents the upper bound on the running time for any input.

→ It gives us an assurance that the algorithm will never go beyond these limit.

2) Average TC -

→ It is an estimation of running time for average inputs which are selected randomly.

→ This case assumes that inputs are equally likely.

→ Each input is given to the program and calculate its run time.
- Finally, total time is divided by no. of inpus that gives average time complexity.

3) Best TC -

→ It is used to analyse an algorithm under optimal condition.

For exp - In linear search, if element found at first position then it is the base best case.

Examples -

1) constant -

Algo	no. of times instructions executed
Add ( )	
{	
int a, b, sum;	1
sum = a+b;	1
display sum	1
}	
Total	3

As 3 is constant ∴ TC = O (1)

whether it is 3 or 3lakh it is constant and can be written as 1

2) Linear -

Algo	no. of times instructions executed
Test ( )	
{	
for(i=1; i<=n; i++)	n+1
print i	n
}	
Total	2n+1

∴ TC = O(n)

3) Quadratic -

Algo	no. of times instructions executed
Test ( )	
{	
for (j=1; j<=n; j++)	n+1
for (i=1; i<=n; i++)	n(n+1)
print i	n·n
}	
Total	$2n^2+2n+1$

∴ TC = $O(n^2)$

4) Logarithmic -

for multiplication & division

Algo	no. of times instructions executed
Test ( )	
{	
i = 1	1
while (i<=n)	$\log_2^n + 1$
{	
print i;	$\log_2^n$
i = i * 2	$\log_2^n$
}	
Total	$3\log_2^n + 2$

Let n = 1000

i = 1	
2	
4	
8	
16	10 times
32	$\log_2^{1024}$
64	$\log_2^{2^{10}}$
128	$10\log_2^2$
256	10
512	
1024	

∴ TC = $O(\log_2^n)$

Searching And Sorting -

→ Linear Search / Sequential Search -

→ Elements are search in sequential manner from first position till element found.

→ Maximum number of passes required by these searching are n-1

→ Minimum it required single pass.

Exp-

x[] = { 10 (0) 15 (1) 12 (2) 18 (3) 19 (4) 25 (5) 13 (6) }.

Let no. = 18

Pass 1 : i=0

x[0] = 10 ≠ 18

Pass 2 : i=1

x[1] = 15 ≠ 18

Pass 3 : i=2

x[2] = 12 ≠ 18

Pass 4 : i=3

x[3] = 18

∴ Elements found at location 3

Algorithm -

1) Start
2) Enter the number of elements i.e n
3) Enter the elements and store into array x.
4) Enter the elements to be search i.e no
5) int linearsearch(x, no, n)

```
{
 for (i=0; i<=n-1; i++)
 {
   if (x[i] == no)
        return i; // Element found
 }
 return -1; // Element not found
}
```

6) If function returns -1 then display 'element not found' otherwise Element found at position i.

```
i= linearsearch(x,no,n);
if (i==-1)
{ element not found
}
else
{ element found
}
```

7) stop

Binary search -

→ In linear search, element will be search from 1st position to last position hence it is a time consuming method.

To overcome these, binary search method can be used.

→ To search element by using binary search, elements need to be in a sorted order

Exp - x[] = { 12 (0) 25 (1) 28 (2) 31 (3) 45 (4) 51 (5) 62 (6) 68 (7) 73 (8) 93 (9)

1) Number to be found is 25

∴ Let no = 25

Pass 1 :

low = 0

high = n-1 = 9

mid = (low + high)/2 = 4

x[4] = 45 ≠ no

∵ no < x[4]

∴ no may be present on left side of x[4]

Pass 2 :

low = 0

high = mid - 1 = 3

mid = (0 + 3)/2 = 1

x[1] = 25 = no

∴ no. found at posⁿ 1

2) Let no = 73

Pass 1 :

low = 0

high = n-1 = 9

mid = (low + high)/2 = 4

x[4] = 45 ≠ no

∵ no > x[4]

∴ no may be present on right side of x[4]

Pass 2 :

low = mid + 1 = 4 + 1 = 5

high = n-1 = 9

mid = (5+9)/2 = 7

x[7] = 68 ≠ no

∴ not not found at position 7

Pass 3 :

low = mid + 1 = 7 + 1 = 8

high = 9

mid = (8+9)/2 = 8

x[8] = 73 = no

∴ no. found at posⁿ 8

3) Let no=50

Pass 1 : low = 0

high = n-1 = 9

mid = (0+9)/2 = 4

x[4] = 45 ≠ no

∵ no > x[4]

∴ no. may be present on right side of x[4]

Pass 2 : low = mid+1 = 4+1 = 5

high = 9

mid = (5+9)/2 = 7

x[7] = 68 ≠ no

∵ no < x[7]

∴ no not found at position ~~4~~ 7

Pass 3 :

low = ~~5~~ 5

high = mid-1 = ~~6~~ 7-1 = 6

mid = (5+6)/2 = 5

x[5] = 51 ≠ no

∵ no < x[5]

∴ no not found at position 5

Pass 4 : low = 5

high = mid-1 = 5-~~1~~1 = 4

since low > high

∴ Element not found

Algorithm -

1) Start
2) Enter the number of elements i.e n
3) Enter the elements and store into array x
   If elements are not sorted then sort it first by using any sorting technique -
   1) Radix sort
   2) Binary tree sort
   3) Bubble sort
   4) shell sort
   5) selection sort
4) ~~Please~~ Enter the elements to be search i.e no
5)

```
bin_search (x, no, n)
{
  low = 0
  high = n-1
  while (low <= high)
  {
    mid = (low+high)/2
    if (x[mid] == no)
          return mid // ele. found at mid
    else
```

if (no < x [mid])
high = mid -1

else
low = mid + 1

}
return -1 //elements not found

→ x[] = { 11 5 21 3 29 17 2 43}

⇒ Let no = 29, 1

First sort the given elements

passes required for sorting n-1

Pass 1 : 1) 11, 5

11 > 5

swap

5 11

{ 5 (11) (21) 3 29 17 2 43}

2) 11 5, 21

11 5 < 21

NO swap

{ 5 11 (21) (3) 29 17 2 43}

3) 21, 3

21 > 3

swap

{ 5 11 3 (21) (29) 17 2 43}

4) 21, 29

21 < 29

no swap

{ 5 11 3 21 (29) (17) 2 43}

5) 29, 17

29 > 17

swap

{ 5 11 3 21 17 (29) (2) 43}

6) 29, 2

29 > 2

swap

{ 5 11 3 21 17 2 (29) (43) }

7) 29, 43

29 < 43

no swap

{ 5 11 3 21 17 2 29 43}

Pass 2 : { 5 11 3 21 17 2 29 43}

1) 5, 11

5 < 11

no swap

{ 5 (11) (3) 21 17 2 29 43}

2) 11, 43

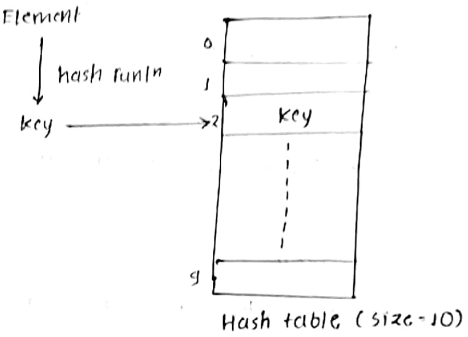
11 < 43 11 > 3

no swap

{ 5 3 11 21 17 2 29 43 }

Hashing -

→ It is one of the searching technique in which elements are stored into hash table by using various hash functions.



Hash table (size - 10)

Hash Functions -

1) Modulo division Method -

→ Address of an element can be calculated by performing mod operation i.e addr = element % table size

Exp -

Addr of 25 = 25 % 10 = 5

2) Direct Method -

→ Element becomes the address.

For exp Addr of 25 is 25

→ It requires large table size (with empty spaces in betn)

3) Substraction Method-

→ A fixed number is substracted from each element to find address.

For exp - Addr of 125 = 125 - 100 (Assuming 100 as fixed no)

= 25

Element will be stored at 25

4) Digit Analysis Method-

→ These method is generally used whenver element contains more number of digits.

For exp - Let no - 6225264

Digit 2 and 6 present multiple times hence address is 26

5) Length Dependant Method -

→ Length of an ~~set~~ element becomes the address.

For exp-

265

Addr is 3

6) Length Independant Method-

→ Address of an element can be calculated by using length and some part of element.

For exp-

Address of 125 = 3 + 12 = 15

7) Folding Method-

a) Fold shifting Method-

Elements are partitioned based on length of address, then these parts are added and ignore carry if generated.

Exp-

no = 12345678

Let address having 2 digits

∴ no. is part" into 12, 34, 56, 78

12 + 34 + 56 + 78

= 180 ← address

↑ Ignore

b) Fold boundary Method-

This method is similar to fold shifting but elements present at the boundary, digits are interchanged.

For exp-

no = 12345678

Let address having 2 digits

∴ no. is partitioned into 12, 34, 56, 78

⇒ 21 + 34 + 56 + 87

= 198 ← Address

↑ Ignor

8) Collision handling Method-

whenever multiple elements hashes to the same address it is called as collision.

For exp -

Let us consider table size as 10 and hashing function as modulo division

addr = element % table size

∴ addr of 12 = 12 % 10 = 2

addr of 32 = 32 % 10 = 2 — collision

Collision handling techniques -

1) open addressing

i) Linear Probing -

→ whenever collision occurs, start searching an empty slot by moving in downward direction till last position of hash table.

→ If no empty slot found then start searching from first position of hash table.

→ If there is no empty slot in entire hash table then it is advisable to increase the size of hash table.

→ Following eqn is used to find address of an element -

$$h(k,i) = (h(k) + i) \bmod m$$

where k is key element, i is prob number = 0,1,2 ---

h(k) is k mod m

m is table size

Exp - 72, 27, 36, 24, 63, 81, 92, 101

~~Table size = 10 = m~~

Table size = 10 = m

Initially, all slots occupy -1 (By array)

0 (If by linked list)

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

h(72,0) = (h(72) + 0) mod m = 10

= (72 mod 10 + 0) mod 10

= 2

h(27,0) = (27 mod 10 + 0) mod 10 = 7

h(36,0) = (36 mod 10 + 0) mod 10 = 6

h(24,0) = (24 mod 10 + 0) mod 10 = 4

h(63,0) = (64 mod 10 + 0) mod 10 = 3

h(81,0) = (81 mod 10 + 0) mod 10 = 1

h(92,10) = (92 mod 10 + 0) mod 10 = 2

collision

$h(92,1) = (92 \bmod 10 + 1) \bmod 10 = 3$ collision

$h(92,2) = (92 \bmod 10 + 2) \bmod 10 = 4$ collision

$h(92,3) = (92 \bmod 10 + 3) \bmod 10 = \underline{5}$

$h(101,0) = (101 \bmod 10 + 0) \bmod 10 = 1$ collision

$h(101,1) = (101 \bmod 10 + 1) \bmod 10 = \underline{2}$ collision

$h(101,2) = (101 \bmod 10 + 2) \bmod 10 = 3$ collision

$h(101,3) = (101 \bmod 10 + 3) \bmod 10 = 5+4$ collision

$h(101,4) = (101 \bmod 10 + 4) \bmod 10 = 5$ collision

$h(101,5) = (101 \bmod 10 + 5) \bmod 10 = 6$ collision

$h(101,6) = (101 \bmod 10 + 6) \bmod 10 = 7$ collision

$h(101,7) = (101 \bmod 10 + 7) \bmod 10 = \underline{8}$

0	
1	81
2	72
3	63
4	24
5	92
6	36
7	27
8	101
9	

→ Drawback of this technique is primary clustering were elements are store at one particular area and rest of hash table is empty. To overcome this Quadratic technique can be used.

ii) Quadratic probing -

→ Instead of searching for ith position in linear probing we search for i^2 position in Quadratic probing that overcomes primary clustering.

→ Following eqn is used to find address of an element -

$$h(k,i) = (h(k) + c_1 i + c_2 i^2) \bmod m$$

where k is key element, i = prob no. = 0, 1, 2 ---

c_1, c_2 = constant but $\underline{c_2 \neq 0}$

m = table size

$h(k) = k \bmod m$

For exp -

72, 27, 36, 24, 92, 101, 81

Initially, the table size is $\underline{10 = m}$

all slot occupy -1

Assume $c_1 = 0$, $c_2 = 3$ ($c_2 \neq 0$ (b/w 1 to 3))

Initially,

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

$h(72,0) = (72 \bmod 10 + 0 + 0) \bmod 10 = \underline{2}$

$h(27,0) = (27 \bmod 10 + 0 + 0) \bmod 10 = \underline{7}$

$h(36,0) = (36 \bmod 10 + 0 + 0) \bmod 10 = \underline{6}$

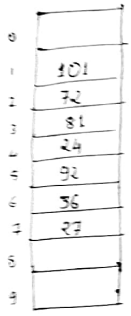
$h(24,0) = (24 \bmod 10 + 0 + 0) \bmod 10 = \underline{4}$

$h(92,0) = (92 \bmod 10 + 0 + 0) \bmod 10 = 2$ collision

c_1 can be assume 0 and c_2 b/w 1 to 3

$h(92,1) = (92 \bmod 10 + 0 + 3) \bmod 10 = \underline{5}$

$h(101,0) = (101 \bmod 10 + 0 + 0) \bmod 10 = \underline{1}$



$h(81,0) = (81 \bmod 10 + 0 + 0) \bmod 10$

$= 1$

collision

$h(81,1) = (81 \bmod 10 + 0 + 3) \bmod 10$

$= 4$

collision

$h(81,2) = (81 \bmod 10 + 0 + 12) \bmod 10$

$= \underline{3}$

iii) Double hashing -

→ This technique two hashing functions h_1 and h_2 to find the address of key element.

→ Following eqn is used to find address of an element -

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$$

where h_1 & h_2 : two hashing tables

For exp -

Let us consider table size = 10 = m

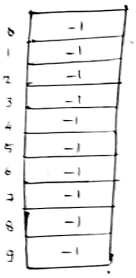
72, 27, 36, 24, 63, 81, 92, 101

$h_1(k) = k \bmod 10$

$h_2(k) = k \bmod 8$

} If not given in que h_2 can be taken

Initially, all slot occupy -1



$h(72,0) = (72 \bmod 10 + 0) \bmod 10$

$= \underline{2}$

$h(27,0) = (27 \bmod 10 + 0) \bmod 10 = \underline{7}$

$h(36,0) = (36 \bmod 10 + 0) \bmod 10 = \underline{6}$

$h(24,0) = (24 \bmod 10 + 0) \bmod 10 = \underline{4}$

$h(63,0) = (63 \bmod 10 + 0) \bmod 10 = \underline{3}$

$h(81,0) = (81 \bmod 10 + 0) \bmod 10 = \underline{1}$

$h(92,0) = (92 \bmod 10 + 0) \bmod 10 = 2$

collision

$h(92,1) = (92 \bmod 10 + 1 \cdot 92 \bmod 8) \bmod 10$

$= 6$

collision

$h(92,2) = (92 \bmod 10 + 2 \cdot 92 \bmod 8) \bmod 10$

$= \underline{0}$

$h(101,0) = (101 \bmod 10 + 0) \bmod 10 = 1$

collision

$h(101,1) = (101 \bmod 10 + 1 \cdot 101 \bmod 8) \bmod 10$

$= 6$

collision

$h(101,2) = (101 \bmod 10 + 2 \cdot 101 \bmod 8) \bmod 10$

$= 1$

collision

$h(101,3) = (101 \bmod 10 + 3 \cdot 101 \bmod 8) \bmod 10$

$= 6$

collision

0	72
1	81
2	72
3	63
4	24
5	101
6	36
7	27
8	
9	

iv) Rehashing -

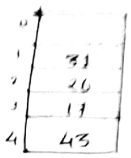
→ When the hash table becomes nearly full the number of collision increases thereby degrading the performance of insertion and search operation.

→ In such a case a better option is to create a new hash table with size double of the original hash table.

→ All the entries of ~~the~~ original hash table are transferred to new hash table.

Ex: Consider hash table size 5 and hashing function as address = element mod 5

Key = 31, 26, 17, 43



addr of 31 = 31 mod 5 = 1

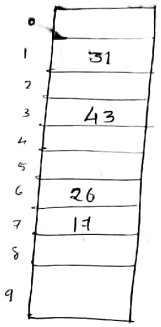
addr of 26 = 26 mod 5 = 1
collision
∴ store in 2nd slot

addr of 17 = 17 mod 5 = 2
collision
∴ store in 3rd slot

addr of 43 = 43 mod 5 = 3
collision
∴ store in 4th slot

∵ hash table is nearly full

∴ If we use rehashing then create another hash table having size 10



Elements are 31, 26, 17, 43

addr of 31 = 31 mod 10 = 1

addr of 26 = 26 mod 10 = 6

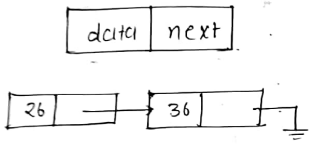
addr of 17 = 17 mod 10 = 7

addr of 43 = 43 mod 10 = 3

2) Chaining-

This technique uses the concept of linked list where each element is divided into two parts -
data and next pointer

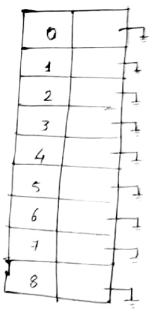
For exp -



Let us consider table size 9 and hashing function ~~is~~ modulo division

Elements are 7, 24, 18, 52, 36, 54, 11, 23

Initially



addr of 7 = 7 mod 9 = 7

addr of 24 = 24 mod 9 = 6

addr of 18 = 18 mod 9 = 0

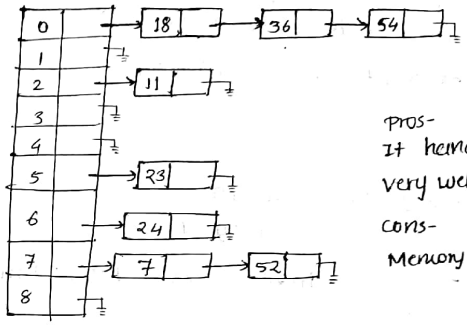
addr of 52 = 52 mod 9 = 7

addr of 36 = 36 mod 9 = 0

addr of 54 = 54 mod 9 = 0

addr of 11 = 11 mod 9 = 2

addr of 23 = 23 mod 9 = 5



Pros-
It handles collision very well

Cons-
Memory wastage in pointers

3) Bucket Hashing-

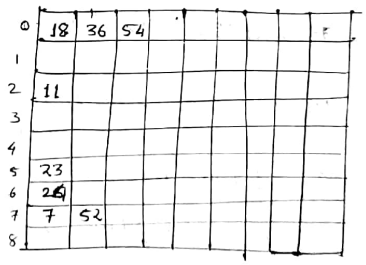
→ This technique uses two dimensional array.

→ If there is any collision then element will be stored in the same row but in the next column.

Exp -

Elements are 7, 24, 18, 52, 36, 54, 11, 23

Table size is 9 x 9 (no of rows should be equals to no of columns)



→ Using linear probing and Quadratic probing. Insert the following values in hash table of size 10. ($C_1 = 0, C_2 = 1$)

28, 55, 71, 67, 11, 10, 90, 44

⇒ Linear Probing

Initially, table size = 10

all slot occupy -1

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

$h(28,0) = (28 \bmod 10 + 0) \bmod 10 = 8$

$h(55,0) = (55 \bmod 10 + 0) \bmod 10 = 5$

$h(71,0) = (71 \bmod 10 + 0) \bmod 10 = 1$

$h(67,0) = (67 \bmod 10 + 0) \bmod 10 = 7$

$h(11,0) = (11 \bmod 10 + 0) \bmod 10 = 1$ collision

$h(11,1) = (11 \bmod 10 + 1) \bmod 10 = 2$

$h(10,0) = (10 \bmod 10 + 0) \bmod 10 = 0$

$h(90,0) = (90 \bmod 10 + 0) \bmod 10 = 0$ collision

$h(90,1) = (90 \bmod 10 + 1) \bmod 10 = 1$ collision

$h(90,2) = (90 \bmod 10 + 2) \bmod 10 = 2$ collision

$h(90,3) = (90 \bmod 10 + 3) \bmod 10 = 3$

$h(44,0) = (44 \bmod 10 + 0) \bmod 10 = 4$

0	10
1	71
2	11
3	90
4	44
5	55
6	
7	67
8	28
9	

Quadratic probing-

Initially table size = 10 = m

all slot occupy -1

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

$h(28,0) = (28 \bmod 10 + 0 + 0) \bmod 10 = 8$

$h(55,0) = (55 \bmod 10 + 0 + 0) \bmod 10 = 5$

$h(71,0) = (71 \bmod 10 + 0 + 0) \bmod 10 = 1$

$h(67,0) = (67 \bmod 10 + 0 + 0) \bmod 10 = 7$

$h(11,0) = (11 \bmod 10 + 0 + 0) \bmod 10 = 1$ collision

$h(11,1) = (11 \bmod 10 + 0 + 1) \bmod 10 = 2$

$h(10,0) = (10 \bmod 10 + 0 + 0) \bmod 10 = 0$

$h(90,0) = (90 \bmod 10 + 0 + 0) \bmod 10 = 0$ collision

$h(90,1) = (90 \bmod 10 + 0 + 1) \bmod 10 = 1$ collision

$h(90,2) = (90 \bmod 10 + 0 + 4) \bmod 10 = 4$ collision

$h(90,3) = (90 \bmod 10 + 0 + 9) \bmod 10 = 0$ collision

$h(90,4) = (90 \bmod 10 + 0 + 16) \bmod 10 = 1$ collision

0	10
1	71
2	11
3	44
4	90
5	55
6	
7	67
8	28
9	
10	

Using linear probing

$h(44,0) = (44 \bmod 10 + 0 + 0) \bmod 10 = 4$ collission

$h(44,1) = (44 \bmod 10 + 0 + 1) \bmod 10 = 5$ collission

$h(44,2) = (44 \bmod 10 + 0 + 2) \bmod 10 = 8$ collission

$h(44,3) = (44 \bmod 10 + 0 + 3) \bmod 10 = 7$

Radix sort -

- This sorting technique requires 10 buckets (bucket 0 to bucket 9) and place the numbers into buckets based on digit.
- Initially, unit place digit of all the numbers are considered and stored into buckets then 10th place, 100th place and so on.

Exp - x[] = 456, 330, 12, 9, 785, 892, 325, 53, 927

Since largest number is 927 that has 3 digits

∴ these sorting requires 3 passes

Pass 1 -

observe the last digits and fill accordingly in buckets

buckets	0	1	2	3	4	5	6	7	8	9
	330		012	053		785	456	927		009
			892			325				

x[] = {330, 012, 892, 053, 785, 325, 456, 927, 009}

Pass 2

buckets	0	1	2	3	4	5	6	7	8	9
	009	012	325	330		053			785	892
		~~009~~	927			456				

x[] = { 009, ~~009~~ 012, 325, 927, 330, 053, 456, 785, 892 }

Pass 3

buckets	0	1	2	3	4	5	6	7	8	9
	009			325	456			785	892	927
	012			330						
	053									

x[] = { 009, 012, 053, 325, 330, 456, 785, 892, 927 }

Algorithm -

Step 1 - Start

Step 2 - Enter the number of elements i.e n

Step 3 - Enter the elements and store into array x

Step 4 - Find the largest number from array x and identify the number of digits

Step 5 - Find the number of passes i.e passes = no. of digits

Step 6 - Store the numbers into buckets i.e

```
div=1
for (i=1; i<= passes; i++) // no of passes
{
    initialize all buckets to null (-1)
    for (j=0; j<=9; j++)
        bucket[j] = null
    For (k=0; k<=n-1; k++)
    {
        d = (x[k]/div) % 10
        copy x[k] to bucket[d]
    }
    copy elements from bucket to array x[]
    div = div * 10
}
```

Step 7 - Display sorted elements from array x.

Step 8 - Stop

Time complexity for radix sort -

It requires two loops - 1st loop requires for no. of passes i.e m &
2nd loop is used to store elements into buckets i.e n

∴ Time complexity is O(mn)

Heap sort -

→ This sorting technique consists of two steps - build heap & heapify

Build Heap -

→ Binary tree is created.

→ There are two ways - i) max heap
parent element is greater than its child elements
if not then perform swapping.

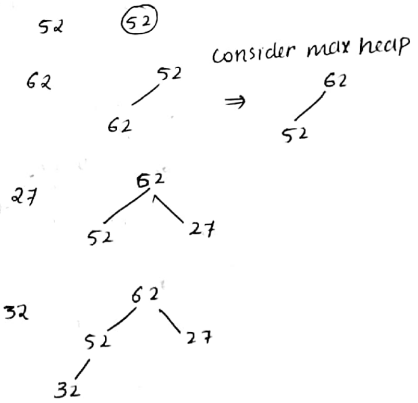
ii) Mean heap
parent element is less than its child elements
if not then perform swapping.

Heapify -

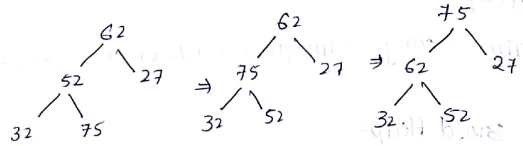
→ First element is swapped with last element so that one element is sorted and again perform step1 i.e build heap.

Exp - 52, 62, 27, 32, 75, 43, 46, 82, 78, 55

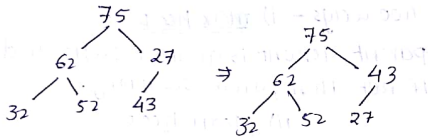
Step 1 - Build heap



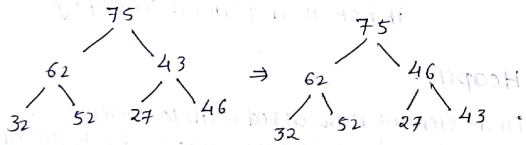
75



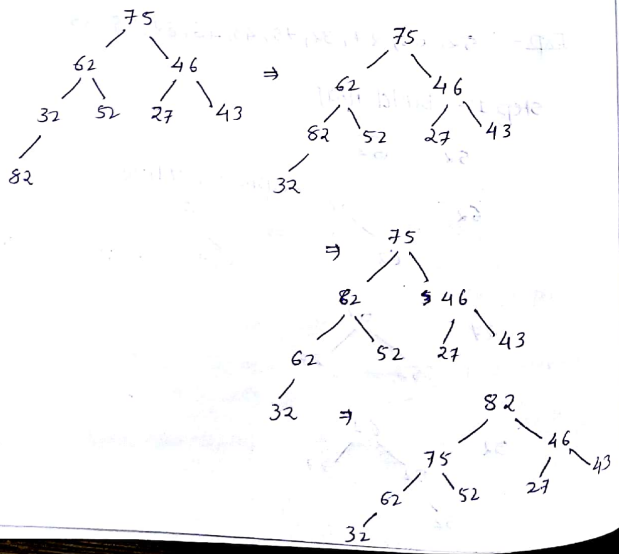
43



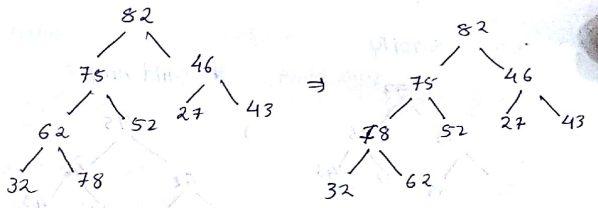
46

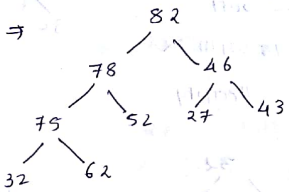


82

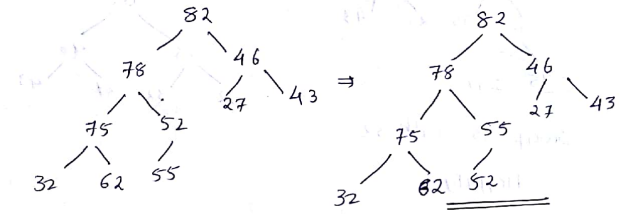


78





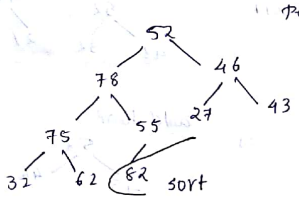
55



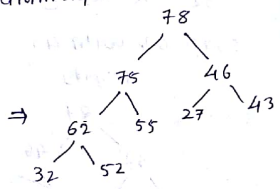
step 2 - heapify

first element last element

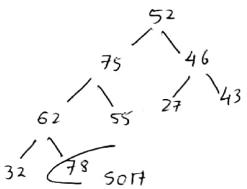
swap 82 with 52



perform step 1 - Build heap

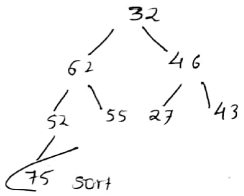


Step 2 - heapify



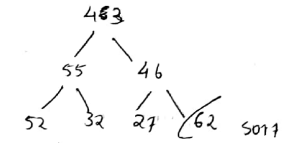
swap 78 with 52

Heapify



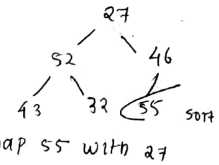
swap 75 with 32

Heapify



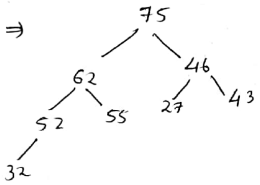
swap 62 with 43

Heapify

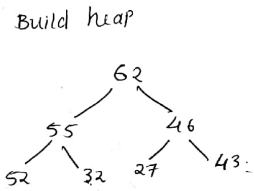


swap 55 with 27

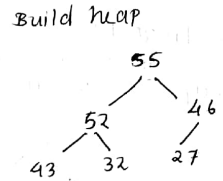
Build heap



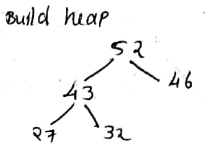
Build heap



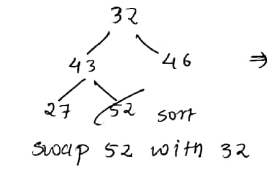
Build heap



Build heap

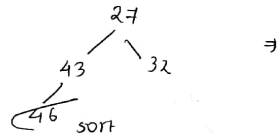


Heapify



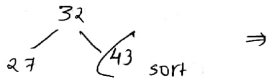
swap 52 with 32

Heapify



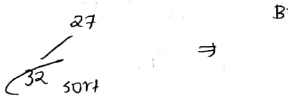
swap 46 with 27

Heapify



swap 43 with 32

Heapify

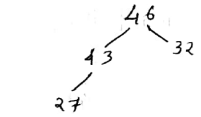


swap 32 with 27

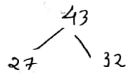
Heapify

27 sorted

Build heap



Build heap

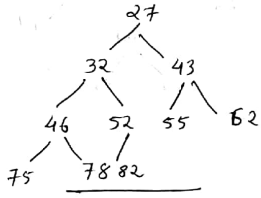


Build heap

Build heap

27

Final tree



→ 47, 12, 75, 88, 90, 73, 57, 1, 85, 50, 62

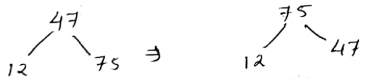
step 1 - build heap

47 (47)

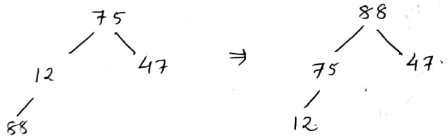
12

consider max heap
parent element should be greater

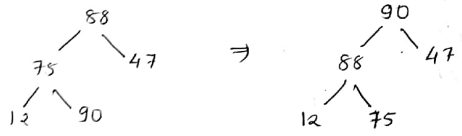
75



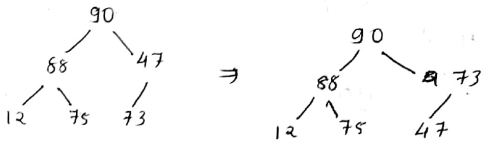
88



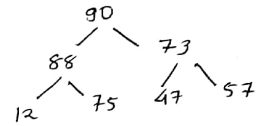
90



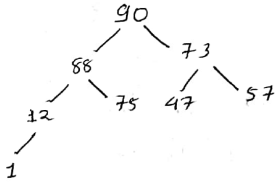
73



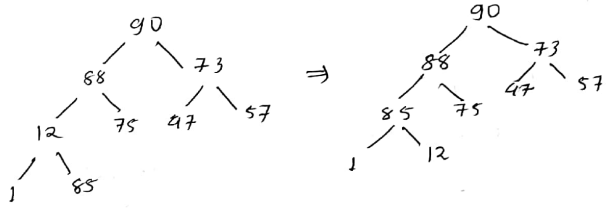
57



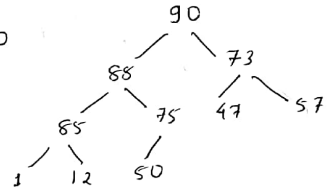
1



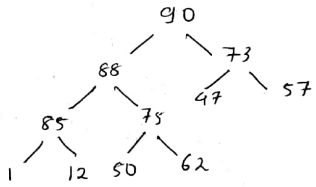
85



50

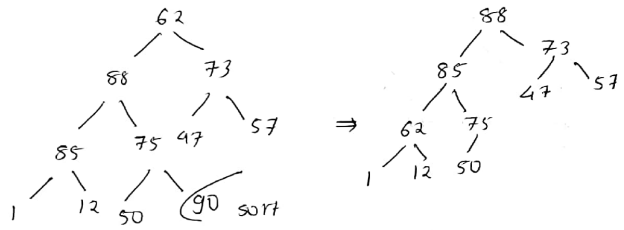


62



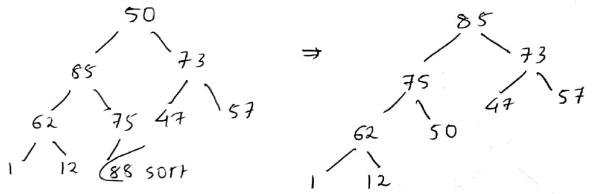
Step 2 - heapify

swap 90 with 62



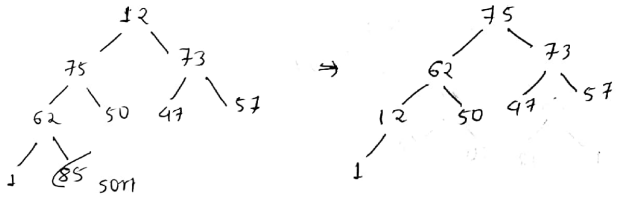
heapify

swap 88 with 50



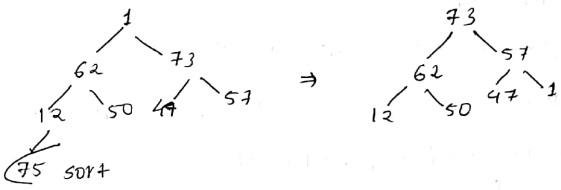
heapify

swap 12 with 85



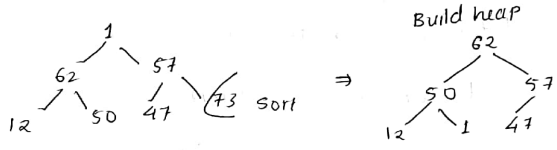
Heapify

swap 75 with 1



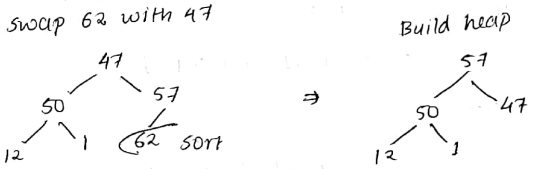
Heapify

swap 73 with 1



Heapify

swap 62 with 47

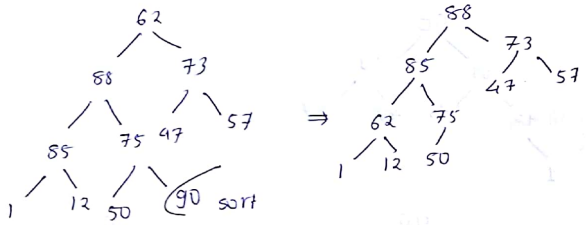


Heapify

swap 1 with 57

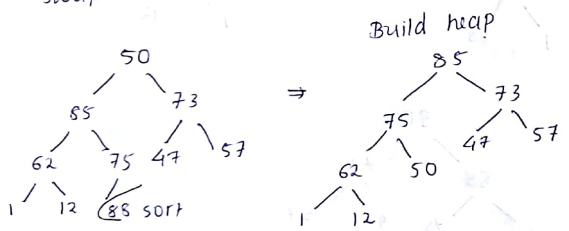
Step 2 - heapify

swap 90 with 62



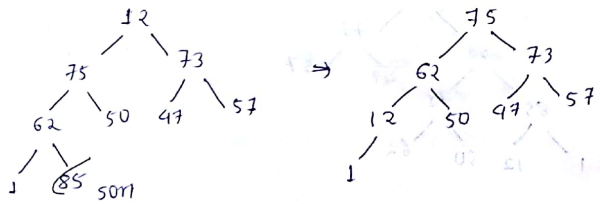
heapify

swap 88 with 50



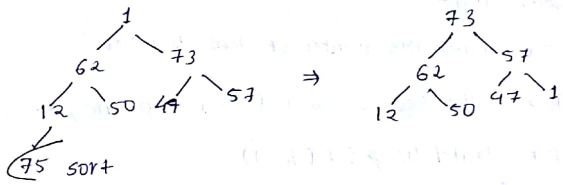
heapify

swap 12 with 85



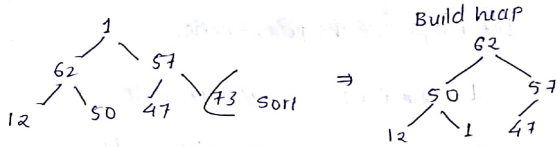
Heapify

swap 75 with 1



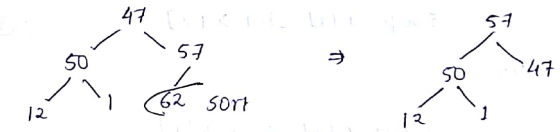
Heapify

swap 73 with 1



Heapify

swap 62 with 47



Heapify

swap 1 with 57

Algorithm -

Step 1 - Start

Step 2 - Enter the number of elements i.e n

Step 3 - Enter the elements and store into array x.

Step 4 - Build heap (x [ ], n)

{

create a binary tree so that parent element must be greater than child elements

Let i represent the parent node

l = 2 * i + 1 // left node

r = 2 * i + 2 // right node

if (x [l] > x [r])

swap x [i] with x [l]

else

swap x [i] with x [r]

}

Step 5 - Heapify (x [ ], n)

{ for (i = n-1; i >= 1; i--)
swap x [0] with x [i]

Build heap (x [ ], i)

}

Step 6 - Display sorted elements from array x.

Step 7 - Stop.

Quick sort -

→ It is also called as partition exchange sort.

→ It follows divide and conquer stragedy that consists of three step – i) Divide
ii) conquer
iii) combine

i) Divide -

→ An array of elements are divided into partitions.

ii) conquer -

→ Elements from each partition are separately sorted.

iii) combine -

→ sorted partitions are combined to get final sorted elements.

Exp - Pass-1 down, quick (x, 0, 7), partition (x, 0, 7)

0	1	2	3	4	5	6	7
(32) pivot	25	62	18	57	15	27	37 up

0	1	2 down	3	4	5	6 up	7
(32)	25	62	18	57	15	27	37

∵ down < up (2 < 6)

x [down] ⟷ x [up]

i.e 62 ⟷ 27

0	1	2 down	3	4	5	6 up	7
(32)	25	27	18	57	15	62	37

32 25 27 18 57 15 62 37

pivot

down up

∵ down < up (4 < 5)

∴ 57 ⟷ 15

32 25 27 18 15 57 62 37

down up

32 25 27 18 15 57 62 37

up down

∵ down > up (5 > 4)

x[up] ⟷ pivot

15 ⟷ 32

15 25 27 18 32 57 62 37

sort

j=4

Pass 2

partition (x, 0, j-1) x, 0, 3

partition (x, j+1, 7) x, 5, 7

down up

15 25 27 18

pivot

15 25 27 18

up down

∵ down > up (1 > 0)

x[up] ⟷ pivot

15 ⟷ 15

15 25 27 18

sort

j=0

partition (x, 0, j-1) x, 0, -1

partition (x, j+1, 3) x, 1, 3

If lower bound is (either < or = to upper bound then stop)

P(x, 0, -1) sort

P(x, 1, 3)

P(x, 5, 7)

down up

57 62 37

pivot

57 down 62 up 37

∵ down < up (6 < 7)

62 ⟷ 37

57 down 37 up 62

57 up 37 down 62

pivot

down > up (7 > 5)

x[up] ⟷ pivot

37 ⟷ 57

37 57 62

sort

5 6 7
37 57 62
sort
j=6

partition (x, j-1) x, 5, 5

partition (x, j+1, 7) x, 7, 7

P(x, 5, 5) } sorted
P(x, 7, 7)

Pass 3

P(x, 1, 3)

down up
1 2 3
(25) 27 18
pivot

down up
25 27 18

∵ down < up (2 < 3)

27 ⟷ 18

down up
1 2 3
(25) 18 27

up down
25 18 27

∵ down > up (3 > 2)

18 ⟷ 25

1 2 3
18 (25) 27
sort
j=2

P(x, 1, 1) } sorted
P(x, 3, 3)

∴ sorted elements are

→ 15 18 25 27 32 37 57 62

↦ 54 26 93 17 77 31 44 55 20

Pass 1 down up
0 1 2 3 4 5 6 7 8
(54) 26 93 17 77 31 44 55 20
pivot

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

∵ down < up (2 < 8)

93 ⟷ 44 20

up up
down
0 1 2 3 4 5 6 7 8
(54) 26 44 17 77 31 44 55 20 93
pivot 20

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

∵ down < up (4 < 5)

77 ⟷ 31 44

down up up

54 26 44 17 ~~31~~ ~~77~~ ~~55~~ ~~93~~
20 44 77 93

54 26 44 17 31 77 93 55 20
up down

down > up (~~55~~ > 31)

x[up] ⟷ pivot

~~17~~ ⟷ 54

17 26 44 54 31 77 93 55 20
sort
j = 3

P(x, 0, j-1) P(x, 0, 2)

P(x, j+1, 8) P(x, 4, 8)

X

Quick(x, 0, 8)
Partition(x, 0, 8)

down up
54 26 93 17 77 31 44 55 20
pivot

54 26 93 17 77 31 44 55 20
down up

∵ down < up

93 ⟷ 20

54 26 20 17 77 31 44 55 93
down up

54 26 20 17 77 31 44 55 93
down up

∵ down < up

77 ⟷ 44

down up

54 26 20 17 44 31 77 55 93

54 26 20 17 44 31 77 55 93
pivot up down

∵ down > up

x[up] ⟷ pivot

31 ⟷ 54

31 26 20 17 44 54 77 55 93
sort
j = 5

P(x, 0, j-1) P(x, 0, 4)

P(x, j+1, 8) P(x, 6, 8)

pass 2

P(x, 0, 4)

down up
31 26 20 17 44
pivot

up down
31 26 20 17 44

∵ down > up

17 ⟷ 31

17 26 20 31 44
sort
j = 3

P(x, 0, j-1) P(x, 0, 2)

P(x, j+1, 4) P(x, 4, 4) - sort

P (x, 6, 8)

down 6 — 77 — 7 55 — up 8 93

pivot

77 — up 55 — down 93

down > up

55 ⟷ 77

6 55 — 7 77 — 8 93

sort

j = 7

P (x, 6, 6) } sorted

P (x, 8, 8)

pass 3

P (x, 0, 2)

down 0 — 17 — 1 26 — 2 20 up

pivot

up 0 17 — down 1 26 — 2 20

down > up

17 ⟷ 17

0 17 — 1 26 — 2 20

sort

j = 0

P (x, 0, -1) – sort

P (x, 1, 2)

sorted

pass 4 P (x, 1, 2)

down 1 26 — 2 20 up

Summary-

→ First element pivot

→ First element index down, last index up

→ Increament down and
Find greater element than pivot

→ Decreament up and find lower element than pivot

→ If down < up then swap x[down] ⟷ x[up] and
procedure is continued

→ If down >= up then swap x[up] ⟷ pivot elements
Pivot element gets sorted at position ~~i~~ j

call partition (x, lb, j-1)
partition (x, j+1, ub)

Algorithm-

step 1 - start

step 2 - Enter the number of elements i.e n.

step 3 - Enter the elements and store into array x.

step 4 - quick (x, 0 lb, n-1 up) initially

```
{
    if (lb < ub)
        j = partition (x, lb, ub)
    quick (x, lb, j-1)
    quick (x, j+1, ub)
}
```

5 6 7
37 57 62
sort
j=6

partition (x, 5, j-1) x, 5, 5

partition (x, j+1, 7) x, 7, 7

P(x,5,5) } sorted
P(x,7,7) }

Pass 3

P(x, 1, 3)

down up
1 3
25 27 18

pivot

down up
25 27 18

∵ down < up (2 < 3)

27 ⟷ 18

down up
1 2 3
25 18 27

up down
25 18 27

∵ down > up (3 > 2)

18 ⟷ 25

1 2 3
18 25 27
sort
j=2

P(x, 1, 1) } sorted
P(x, 3, 3) }

∴ sorted elements are

→ 15 18 25 27 32 37 57 62

---

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

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

pivot

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

∵ down < up (2 < 8)

93 ⟷ 44 20

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

pivot
20 44

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

∵ down < up (4 < 5)

77 ⟷ 31 44

down up

(54) 26 44 17 77 31 55 93
20 44 77 93

(54) 26 44 up 17 31 down 77 93 55 20

down > up (5>3)

x[up] ⟷ pivot

17 ⟷ 54

17 26 44 54 31 77 93 55 20
sort
j=3

P(x, 0, j-1) P(x, 0, 2)

P(x, j+1, 8) P(x, 4, 8)

X

Quick(x, 0, 8)
Partition(x, 0, 8)

down up
↦ (54) 26 93 17 77 31 44 55 20
pivot

(54) 26 down 93 17 77 31 44 55 up 20

∵ down < up

93 ⟷ 20

(54) 26 down 20 17 77 31 44 55 up 93

54 26 20 17 down 77 31 up 44 55 93

∵ down < up

77 ⟷ 44

(54) 26 20 17 down 44 31 up 77 55 93

(54) 26 20 17 44 up 31 down 77 55 93
pivot

∵ down > up

x[up] ⟷ pivot

31 ⟷ 54

31 26 20 17 44 54 77 55 93
sort
j=5

P(x, 0, j-1) P(x, 0, 4)

P(x, j+1, 8) P(x, 6, 8)

pass 2

P(x, 0, 4)

down up
(31) 26 20 17 44
pivot

up down
(31) 26 20 17 44

∵ down > up

17 ⟷ 31

17 26 20 31 44
sort
j=3

P(x, 0, j-1) P(x, 0, 2)

P(x, j+1, 4) P(x, 4, 4) - sort

P (x, 6, 8)

down 6 — 77 — pivot; 7 — 55; up 8 — 93

77 — up 55 — down 93

down > up

55 ⟷ 77

6 — 55; 7 — 77 — sort; j = 7; 8 — 93

P (x, 6, 6) } sorted
P (x, 8, 8) }

pass 3

P (x, 0, 2)

down 0 — 17 — pivot; 1 — 26; up 2 — 20

up 0 — 17; down 1 — 26; 2 — 20

down > up

17 ⟷ 17

0 — 17 — sort; j = 0; 1 — 26; 2 — 20

P (x, 0, -1) — sort

P (x, 1, 2) — sorted

pass 4 P (x, 1, 2)

down 1 — 26; up 2 — 20

P (x, 1, 2) — down 1 — 20 — PIVOT; up 2 — 26

down, up

20 ⟷ 26

Summary -

→ First element pivot

→ First index down, last index up

→ Increament down and
Find greater element than pivot

→ Decreament up and find lower element than pivot

→ If down < up then swap x[down] ⟷ x[up] and
procedure is continued

→ If down >= up then swap x[up] ⟷ pivot elements
Pivot element gets sorted at position j

call partition (x, lb, j-1)
partition (x, j+1, ub)

Algorithm -

Step 1 - start

Step 2 - Enter the number of elements i.e n.

Step 3 - Enter the elements and store into array x.

Step 4 - quick (x, 0, n-1) Initially lb, up

```
{
    if (lb < ub)
        j = partition (x, lb, ub)
    quick (x, lb, j-1)
    quick (x, j+1, ub)
}
```

step 5 - partition (x, lb, ub)

```
{
    down = lb
    up = ub
    pivot = x[lb] or x[down]

  while (down < up).
  {
    while (x[down] < pivot)
      down ++
    while (x[up] > pivot)
      up --
    if (down < up)
    {
      int temp = x[down]
          x[down] = x[up]
          x[up] = ~~x[down]~~ temp
    }
  }
      temp = x[up]
      x[up] = pivot
      pivot = temp
  }  returnup
```

step 6 - Display sorted elements from array x

step 7 - stop

Bubble sort -

It is also called ~~extended~~ exchange sort, in which one element is sorted in each pass.

Hence to sort n elements, it requires n-1 passes.

Example -

25 67 12 5 4 62 52

Pass-1

25 12 5 4 62 52 67
sort

Pass-2

~~25~~ 12 5 4 25 52 62 67
sort

Pass-3

5 4 12 25 52 62 67
sort

Pass-4

4 5 12 25 52 62 67
sort

Pass-5

4 5 12 25 52 62 67
sorted

Algorithm-

Step 1 – start

Step 2 - Enter the number of elements i.e n

Step 3 - Enter the elements and store into array x.

Step 4 - Bubble (x[], n)

```
{
    flag = 1;
    for(i=0; i<n && flag == 1; i++)
    {
        flag = 0;
        for (j=0; j < n-i ; j++)
        {
            if ( x [j] > x[j+1])
            {
                flag = 1;
            }
        }
    }
}
```

Step 5 - Display sorted elements from array x.

Step 6 - Stop

Merge sort-

→ This sorting follows divide and conquer strategy that consists of three steps – i) Divide
ii) conquer
iii) combine

i) Divide -

→ An array of elements are divided into partitions.

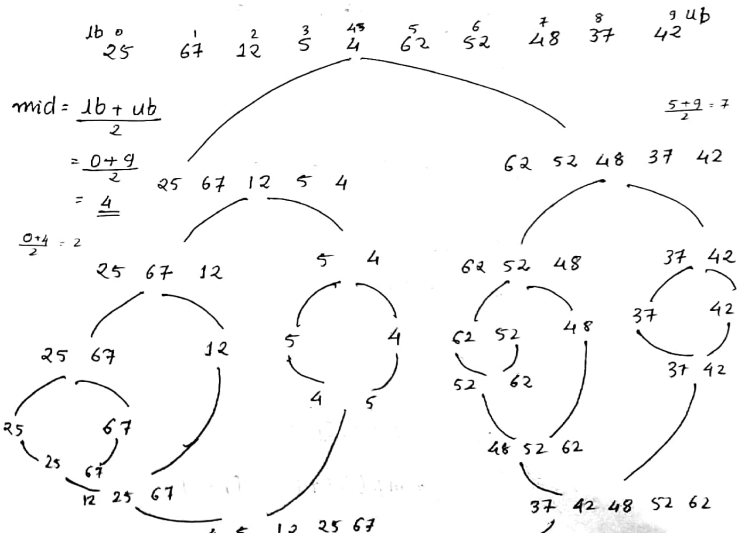
ii) Conquer-

→ Elements from each partition are separately sorted.

iii) combine-

→ sorted partitions are combined to get final sorted elements.

Example-



Algorithm-

step 1- start

step2- Enter the number of elements i.e n.

step 3- Enter the elements and store into array x.

step 4- mergesort (x[], lb, ub)

```
{
  if (lb < ub)
  {
    [mid = (lb+ub)/2]   mid = (lb + ub)/2;
    mergesort (x, lb, mid);
    mergesort (x, mid+1, ub);
    merge (x, lb, mid, ub);
  }
}
merge (x[], lb1, ub1, ub2)
{
    i = lb1;
    j = ub1 + 1;
    k = 0;
 while (i <= ub1 && j <= ub2)
 {
    if (x[i] < x[j])
          temp[k++] = x[i++];
    else
          temp[k++] = x[j++]
 }
 while (i <= ub1)  // remaining elements
          temp[k++] = x[i++]
 while (j <= ub2)  // remaining elements
          temp[k++] = x[j++]

  copy elements from temp[] to x[]

 i.e    x[] ⟵ temp[]
```

step 5- Display sorted elements from array x

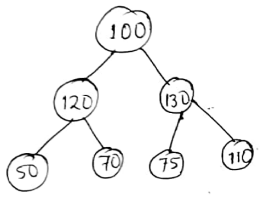
step 6 - stop.

Tree and Graph

Binary Tree-

→ It is a type of tree in which every node have atmost two sub trees i.e left sub tree and right sub tree.

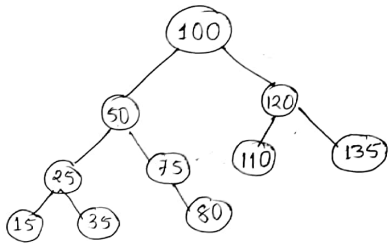
Exp-



Binary search Tree (BST)-

→ It is a binary tree i.e every node have atmost two sub trees i.e left and right sub tree but left sub tree data must be less than parent node and right sub tree data must be greater than or equal to parent node.

Exp- 100, 50, 120, 25, 75, 35, 110, 135, 15, 80



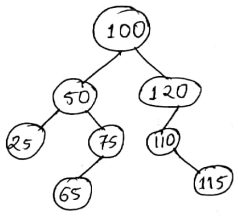
Representation of Binary Tree-

There are two ways - i) Array representation
ii) Linked list representation

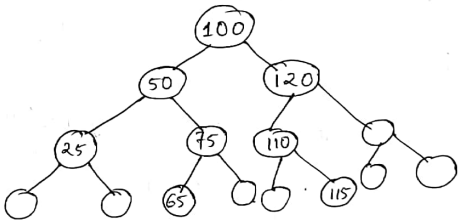
1) Array representation-

First, convert the given binary tree into complete binary tree i.e every node must have either 0 or 2 subtrees.

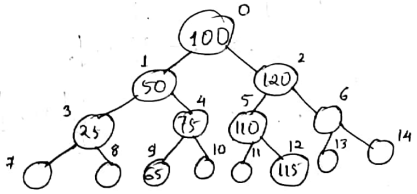
forexp -



→ complete Binary tree can be drawn as follows -



→ Assign index from root node

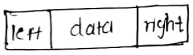


→ x[] =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
100	50	120	25	75	110	/0	/0	/0	65	/0	/0	115	/0	/0

2) Linked list representation-

Every node is divided into three parts - i) data
ii) left pointer and
iii) right pointer

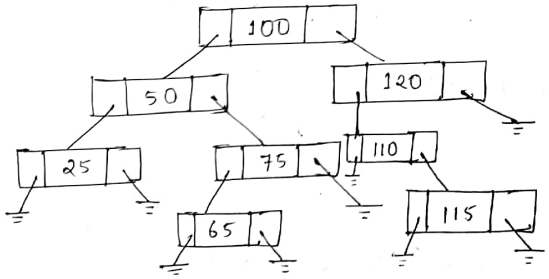


→ Structure representation of a node -

```
struct node
{
    int data;
    struct node * left, * right;
};
```

} Self ref referential structure

→ Linked list representation for above binary tree can be drawn as follows-



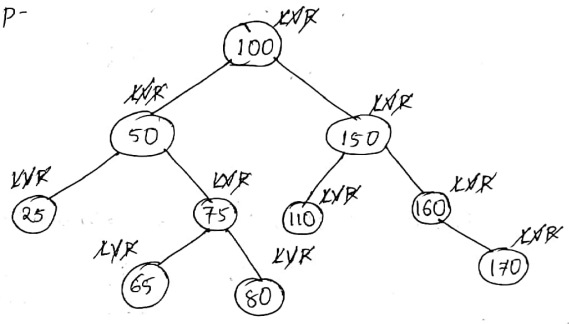
→ If there are n elements then number of null pointers are n+1, hence memory waste to represent binary tree by using linked list.

Binary Tree Traversal -

i) Inorder Traversal-

It follows L V R policy i.e Left Visit Right

Exp-



∴ Inorder Traversal is : 25, 50, 65, 75, 80, 100, 110, 150, 160, 170

Implementation-

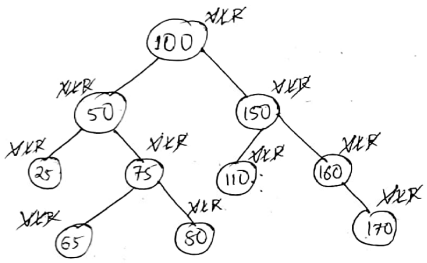
→ Inorder (struct node * P) ← root ~~root node always~~

```
{
    if (P != null)
    {
        inorder (P→ left)
        print (P→ data)
        inorder (P→ right)
    }
}
```

ii) Preorder Traversal-

It follows VLR policy i.e visit Left Right

Exp -



Preorder Traversal is : 100, 50, 25, 75, 65, 80, 150, 110, 160, 170

root node always comes first

Implementation -

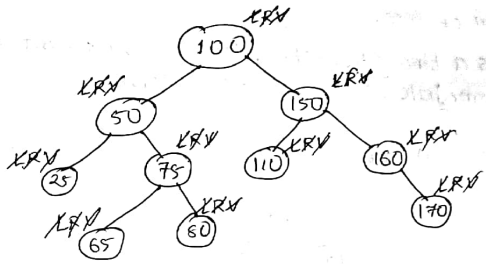
→ Preorder (struct node * P) ← root

```
{
    if (P != null)
    {
        print (P → data)
        preorder (P → left)
        preorder (P → right)
    }
}
```

Post order Traversal-

It follows LRV policy i.e Left Right visit

Exp -



Post order Traversal is : 25, 65, 80, 75, 50, 110, 170, 160, 150, 100

root node always comes last

Implementation -

→ Postorder (struct node * P) ← root

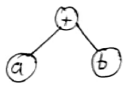
```
{
    if (P != null)
    {
        postorder (P → left)
        postorder (P → right)
        print (P → data)
    }
}
```

Expression tree -

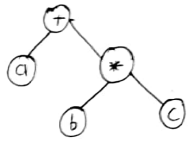
→ This tree is used to represent mathematical expression in the form of tree.

→ It is a binary tree whose leaf nodes represent operands and intermediate node represent operators.

Exp - i) a + b

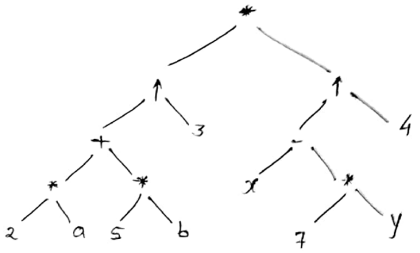


ii) a + b * c



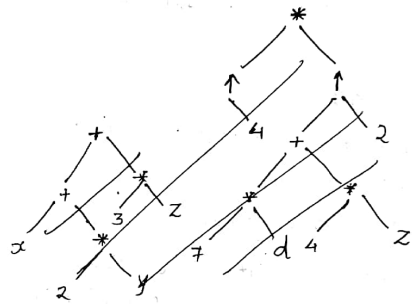
iii) $(2a+5b)^3 (x-7y)^4$

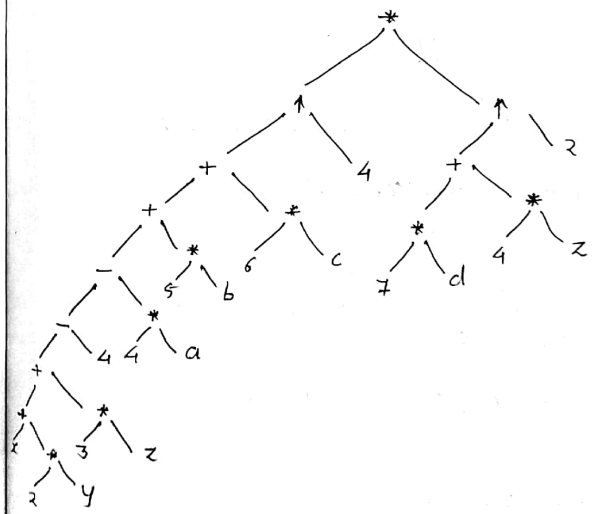
=((2 * a + 5 * b)↑3) * ((x - 7 * y)↑4)



iv) $(x+2y+3z-4-4a+5b+6c)^4 (7d+4z)^2$

= ((x + 2*y + 3*z - 4 - 4 * a + 5 * b + 6 * c)↑4) *((7*d+4*z)↑2)





→ Construct a binary tree for the following traversal.

Inorder: 25, 50, 60, 75, 80, 100, 110, 115, 120, 125

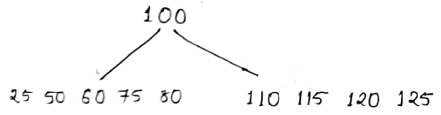
Preorder: 100, 50, 25, 75, 60, 80, 120, 110, 115, 125

100

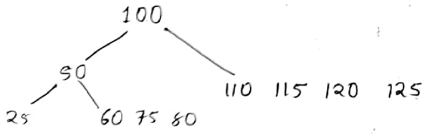
→ In preorder traversal, root node always comes first
∴ 100 be the root node

Find these root node in ~~preo~~ inorder traversal.

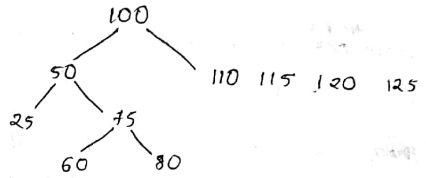
Nodes which are present on the left side of 100 are attached as a left sub tree and the nodes which are present on the right side of 100 are attached as the right side sub tree.



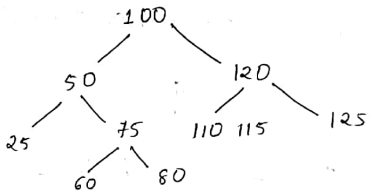
Next node in preorder traversal is 50



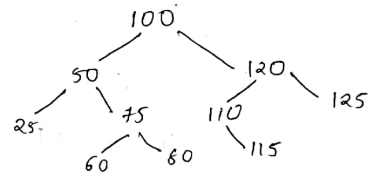
Next node is 25 (Already arranged) ∴ its 75



Next preorder 60, 80 are arranged ∴ its 120



Next preorder is 110



Next preorder 115, 125 are arranged

→ Construct a binary tree for following traversal

Inorder - D, G, B, A, H, E, I, C, F

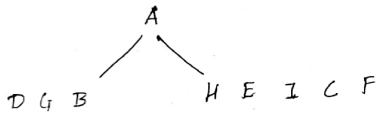
post order - G, D, B, H, I, E, F, C, A

→ In post order traversal last element root node always comes last

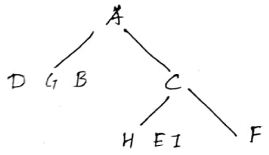
∴ A be the root node

Find these root node in inorder traversal

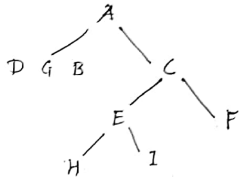
Nodes which are present on left side of A are attached as left sub tree and the nodes which are present on right side of A are attached as the right sub tree



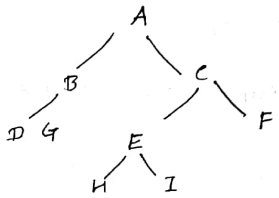
Next order node in postorder traversal is C



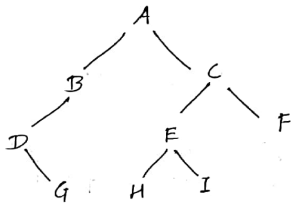
Next node in post order traversal is F (its arranged) ∴ it is E



Next node is I and H are arranged ∴ it is B



Next post order traversal is E D



Graph-

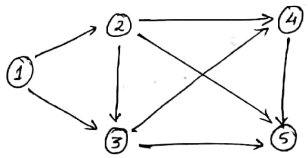
→ It is a collection of vertices and edges denoted by G = (V, E), where

V = set of vertices, E = set of edges

1) Directed Graph:-

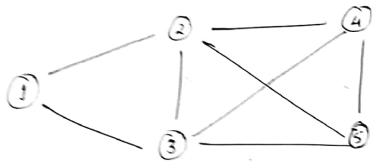
Flow of direction can be consider in one way.

Exp-



ii) Undirected or bidirected graph :-

Flow of direction can be consider in two way.

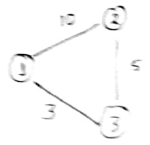


Minimum spanning Tree (MST)-

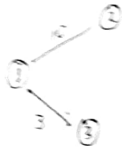
It is a subset of graph in which every vertex is reachable from other vertices but cycle should not be formed.

For any graph there will be multiple spanning tree and a spanning tree whose cost is minimum is called as Minimum Spanning Tree (MST).

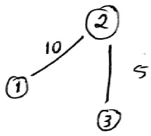
Ex:-



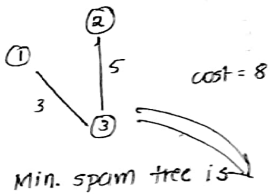
Spanning trees



cost = 13



cost = 15



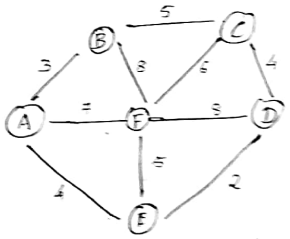
cost = 8

∴ Min. spam tree is

MST Algorithms-

↦ Kruskal's Algorithm and the Prim's Algorithm

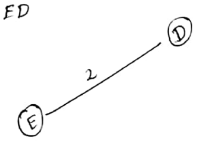
↦ Kruskal's Algorithm-



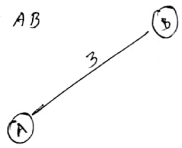
Arrange all the edges in ascending order-

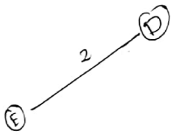
FD, AB, AE, CD, BC, EF, CF, AF, BF, DF

min = ED

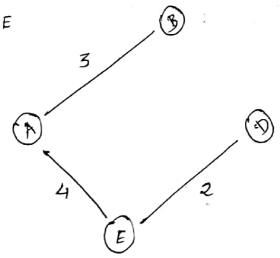


min = AB

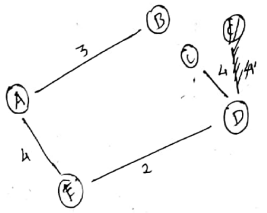




min = AE



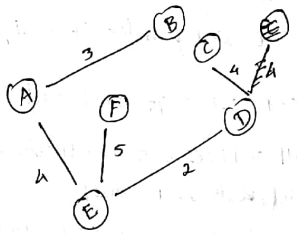
min = CD



min = BC

since, it form cycle, reject BC

min = EF



min = CF

since, it form cycle, reject EF

min = AF

since it form cycle, reject AF

min = BF

since, it forms cycle, reject BF

min = FD

Since, it forms cycle, reject FD

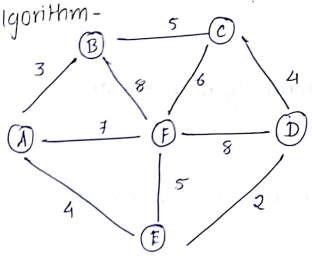
Hence, above graph be the min spanning tree

cost = 18

Algorithm-

1) Start
2) Enter the no. of vertices and edges.
3) Enter the edges among the vertices.
4) Sort the edges in ascending order of weight.
5) consider minimum weight cost edge and add to the MST.
6) Repeat step 5 till all vertices are reachable from each other but cycle should not be formed.
7) Add the weights of all considered edges and display it
8) Stop

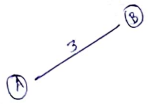
Prim's Algorithm-



Let A be the source code

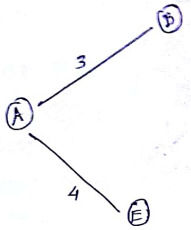
Edges are {AB, AE, AF}

∴ min = AB



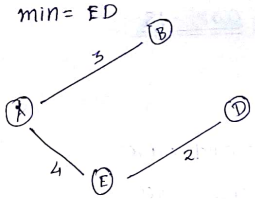
Edges connected to A, B = {AE, AF, BC, BF}

min = AE



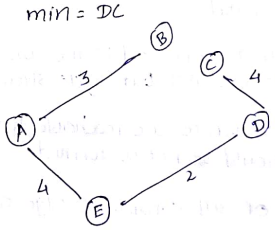
Edges connected to A, B, E = {AF, EF, BF, BC, ED}

min = ED



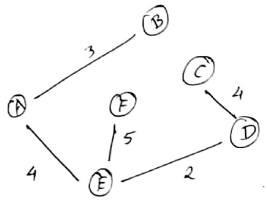
Edges connected to A, B, D, E = {AF, EF, BF, BC, DF, DC}

min = DC



Edges connected to A, B, C, D, E = {AF, BF, BC, CF, FF, DF

min = FF



Hence, above graph be the min spamming tree =

cost = 18

Algorithm-

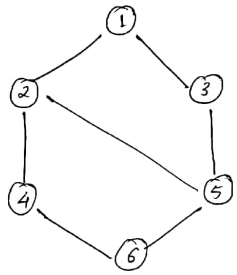
1) start.
2) Enter the no. of vertices and edges.
3) Enter the edges among vertices
4) consider source node, find the connected edges and add minimum cost edge to the MST.
   Mark vertices as visited.
5) Find the edges which are connected to the visited vertices and add min cost edge to the MST but cycle should no be formed.
6) Repeat step 5 till all vertices are reachable from each other but cycle should not be formed.
7) Add the weight of all considered edges and display it.
8) stop

Graph Traversals-

→ Depth First Search (DFS)
→ Breadth First search (BFS)

1) Depth First search (DFS)-
This traversals starts from any vertex and uses stack data structure.

Exp -



Let's start from vertex 1

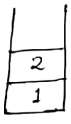
Push 1

visit 1

top = 1

adj. vertex are 2, 3

push 2, visit 2



top = 2

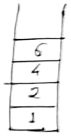
adj. vertex one 4, 5

push 4, visit 4

top = 4

adj. vertex is 6

push 6, visit 6



top = 6

adj. vertex is 5

push 5, visit 5

top = 5

adj. vertex is 3

push 3, visit 3

top = 3

since, vertex 3, doesn't have unprocessed adjacent vertices

∴ pop 3

since vertices 1, 2, 4, 6 and 5 doesn't have unprocessed adjacent vertices

∴ pop these vertices

since, stack is empty stop the traversal.

∴ DFS traversal is 1, 2, 4, 6, 5, 3

Algorithm -

1) start
2) Enter the no. of vertices and edge's.
3) Enter the edges among the vertices.
4) select any vertex from which DFS traversal starts.
5) Push selected vertex on to the stack and mark as visited.
6) Find top of stack and its adjacent vertices.
7) select any adjacent vertex and push on to the stack. Mark as visited.
8) If top of stack (top) vertex doesn't have unproccessed adjacent vertices then pop these vertex.
9) Repeat steps 7 and 8 until all vertices are visited or stack becomes empty.
10) Display DFS traversal in the order vertices are visited.
11) STOP

2) Breadth First search - (BFS) -

The traversal starts from any vertex and uses Queue datastructure