

22/8/23:

last base class
class

Desonates
constitutions
about

Operator Overloading:

cannot be
overloaded

- dot operator
- ? - conditional operator
- size of - size operator
- :: - scope resolution.

Overloading - same operator different functions.

Syntax

return type classname :: operator op (arg)
 → key word.

{
body of the function

}

Unary operators:

OP a (00)

→ operand

++ a (06)

a → operator

a ++

→ fun - no arg
→ fun - 1 arg

Binary operators:

x op y
 ↑ ↑
 | |

 star

→ fun - 1 arg
→ fun - 2 arg

→ operator

$a + y$

Overloading of unary operator:

(unary op as normal function)

class Space

```
{  
    int a, b, c;
```

public:

void getData (int a, int b, int c)

```
{  
    a = a;  
    b = b;  
    c = c;  
}
```

void disp()

{

cout << a << b << c;

}

void operator -(); // declaration

} ;

return type ← void Space :: operator -();

should be
specified

{

$a = -a;$

... ...

```

y = -y;
z = -z;
}

int main()
{
    Space s;
    s.getData (-10, 20, 30);
    s.disp();
    -s; invoked and it will
    s.disp();
    return 0;
}

```

there
go
on
with

(unary operators for end function)

Class Space

```

{
    public:
        friend void operator - (Space);
}

```

↳ key word

void operator - (Space D)

```

{
    D.x = - D.x;
}
```

$$\begin{aligned} D.y &= -D.y \\ D.z &= -D.z \end{aligned}$$

}

Binary operators overloading:

(normal function)

class complex

```
{ float x;
float y;
```

public:

complex(1 d)

complex(float real, float img)

{

x = real;

y = img;

}

complex operator +(complex c)

{

complex temp;

temp.x = x + c.x;

temp.y = y + c.y;

It will go here
as '+' is invoked

```
    return temp;  
}  
  
void disp();  
{  
    cout << a << " + " << y;  
}  
};  
  
int main()  
{  
    complex c1, c2, c3;  
    c1 = complex(2.5, 4.3);  
    c2 = complex(1.6, 2.3);  
    c3 = c1 + c2;  
    c1.disp();  
    c2.disp();  
    c3.disp();  
    return 0;  
}
```

(friend function)

class Space

```

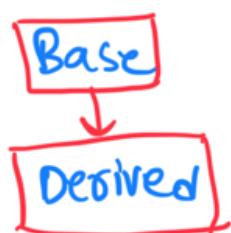
{
    public:
        friend void operator -(Complex D);
    };
    friend Complex operator +(Complex, Complex);
    Complex operator +(Complex A, Complex B)
{
    Complex temp;
    temp.x = A.x + B.x;
    temp.y = A.y + B.y;
    return temp;
}

```

23/8/23:
~ ~ ~

Inheritance:
~ ~ ~ ~

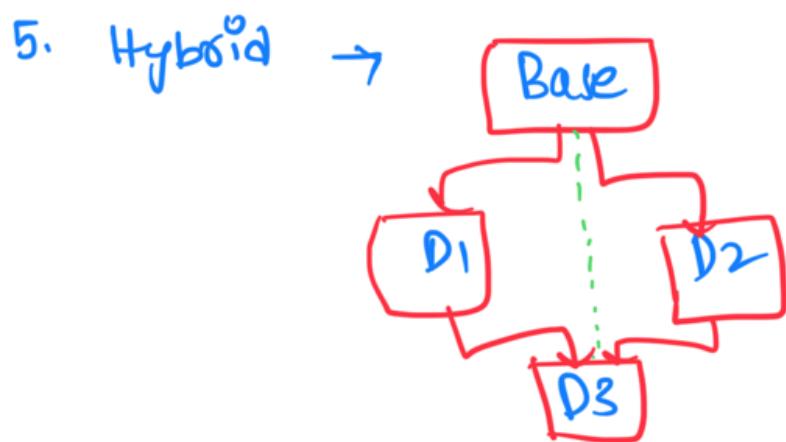
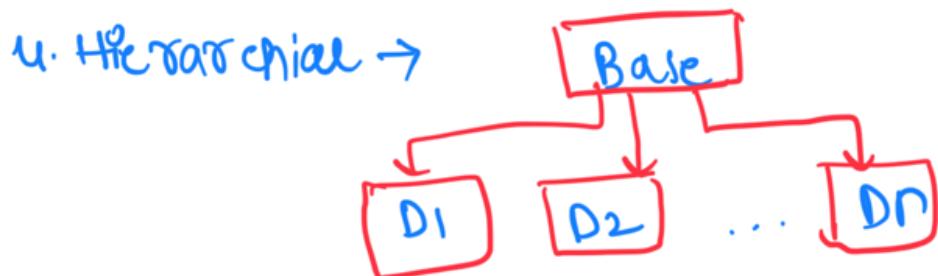
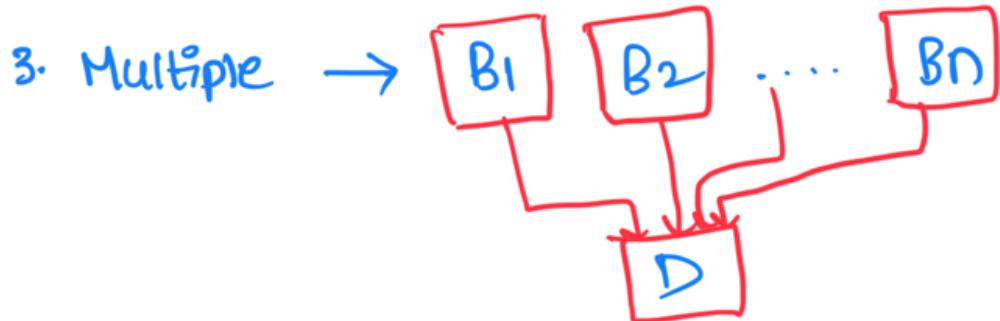
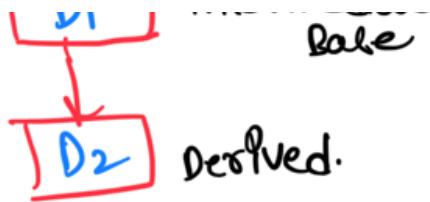
1. single



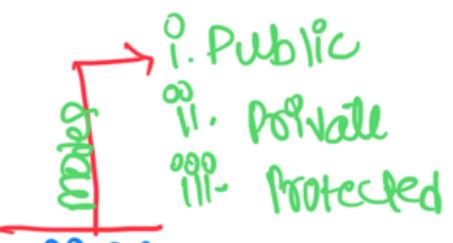
Parent → Base class.
The one inherited → derived class

2. Multilevel →





Defining a derived class:



class derivedclassname : visibility basclassname
{ }

Body of derived class:

};

e.g.: class D : public B

class D : B // By default it will take as
private.

class B : public B1, B2 // B1 is public
and B2 is considered
as default private

Public visibility:

class XYZ

{
int a; // not inheritable
public:

void get-a();

}

class ABC : public XYZ

d

int a;
public : int y;
void disp();

Note: whatever is outside
public mode will not be
inherited (i.e. protected, private)

Derived:

Private:

a

Public:

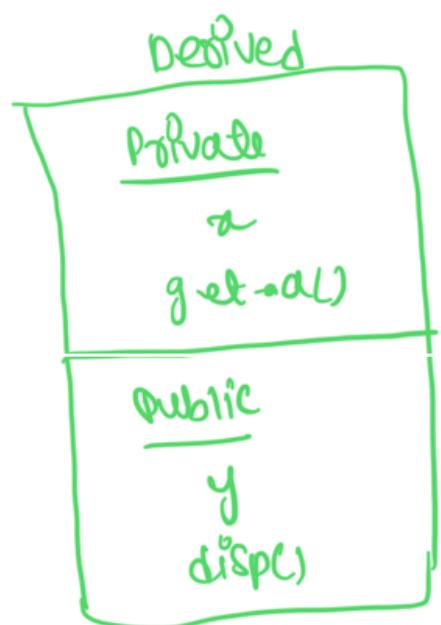
y,
disp(),
get()

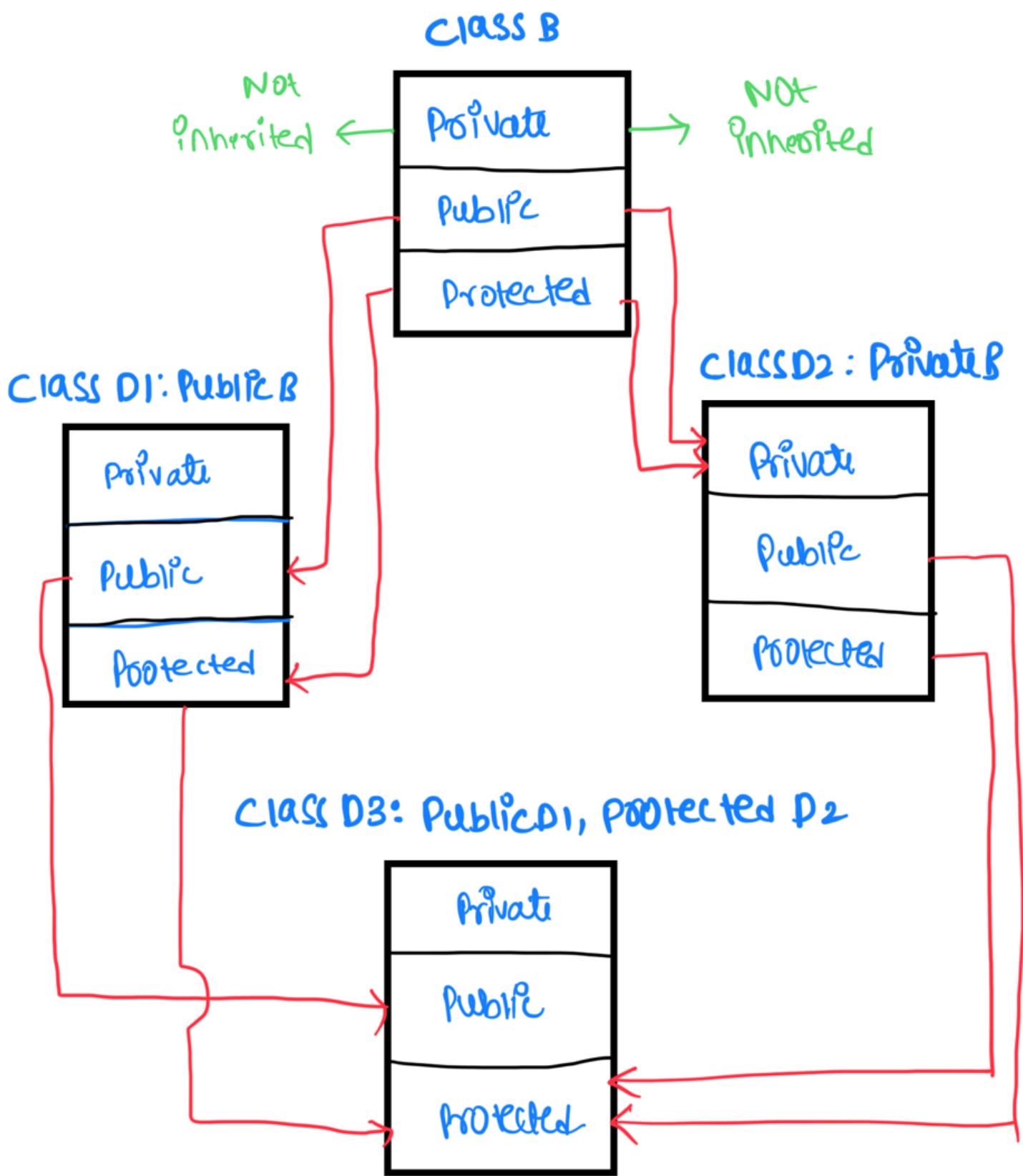
```
int main()
{
    ABC abc;
    abc.get-a();
    abc.y = 10;
    return 0;
}
```

Private visibility:
~~~~~

class ABC : Private XYZ

{  
cout << get-a();  
}





★

Base class | Derived class Visibility

| visibility | private       | public        | protected     |
|------------|---------------|---------------|---------------|
| private    | NOT Inherited | NOT Inherited | NOT Inherited |
| public     | private       | public        | protected     |
| protected  | private       | protected     | protected     |

Single Inheritance:

class B

{

int;

public:

int b;

void get-ab()

{

a=5;

b=10;

}

int get-a()

{

return a;

}

```
void show_a()
{
    cout << a;
}

};

class D: public B
{
    int c;
public:
    void mul();
    {
        c = b * get_a();
    }
    void disp();
    {
        cout << get_a();
        cout << b << c;
    }
};

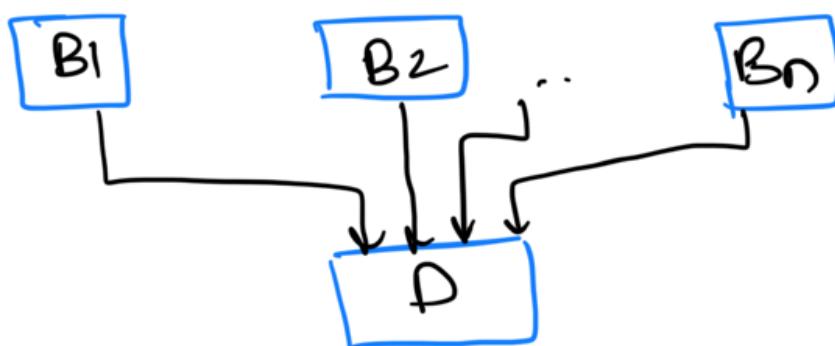
int main()
{
    D d;
```

```
d. get-ac();  
d. mul();  
d. show-ac();  
d. disp();  
d. b=20;  
d. mul();  
d. disp();  
return 0;  
}
```

29/8/23:  
~~~~~

Inheritance:
~~~~~

Multiple Inheritance  
~~~~~



ClassD: Public B1, Public B2

Eg:
~~~~~ class M  
S

-

Protected:

int m;

Public:

void get\_m(int);

}

class N

{

Protected:

int n;

Public:

void get\_n(int);

}

class P : Public M, Public N

{

Public:

void display();

}

void M::get\_m(int a)

{

m=a;

}

-

-

```
void N:: get-n(p+n)
```

```
{
```

```
n=b;
```

```
}
```

```
void P:: disp()
```

```
{
```

```
cout << m << n;
```

```
cout << m * n;
```

```
}
```

```
int main()
```

```
{
```

```
P p;
```

```
p.get-m(20);
```

```
p.get-n(10);
```

```
p.disp();
```

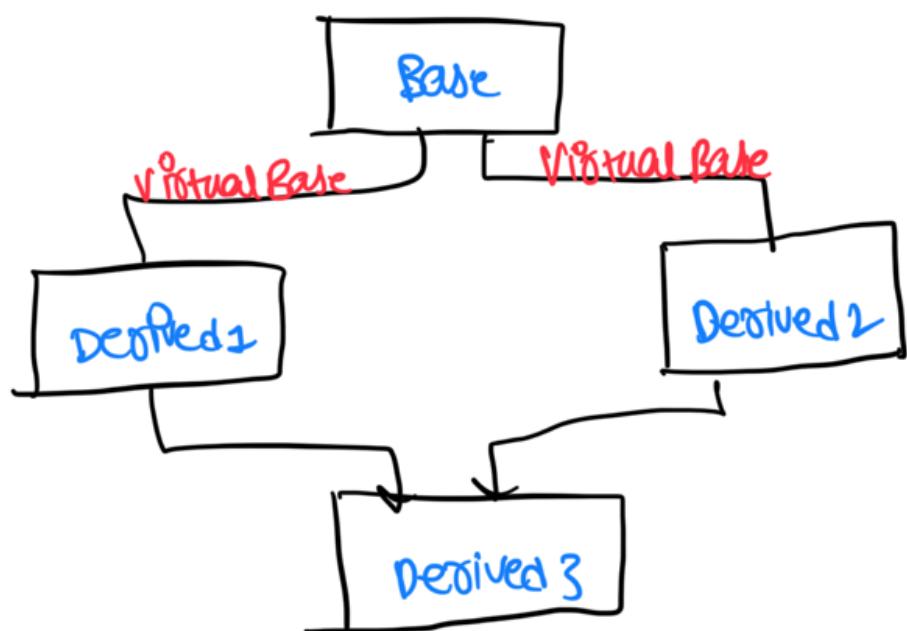
```
return 0;
```

```
}
```

Multi level inheritance: ?  
~~~~~ ~~~~

Hierarchical: ?
~~~ ~~ ~

Hybrid :



- If we use virtual base, then only one copy of member function in base will be inherited to D3. Else 2 copies (duplication) will be inherited.

Class D1: public virtual B;

Class D2: virtual public B;

Class D2: public D1, public D2, virtual public Base;

Notes: when we feel any duplication happening then we need to use virtual (it may be base class, next class that should be decided by programmers)

void M :: get (int a)

{  
m=a;

}

void N :: get (int b)

{

n=b;

}

~ n .

Same function  
name can be  
used in two classes.

BUT

We need to use  
scope resolution  
operator in function

```

int main()
{
    P p;
    P.M :: get(10);
    P.N :: get(12);
}

```

↑ previous main function  
class name

30/8/23:

construction in derived class:

e.g.: class D : Public A, Public B

{ int d;

Public:

D(int a, int b, float c, float d, int e) : A(a), B(b)

{

    d := e;

}

};

int main()

{

    D d2(4, 5, 5.8, 6.8, 3);

    return 0;

}

Syntax:

Derived constructor (Arg1, Arg2, ... Argn) : base1(Arg1), base2(Arg2).

{  
body of derived constructor;  
}

Execution of Base class constructors:  
~~~~~ ~ ~~~~~ ~~~~~

Type:

1. class B: public A{};
2. class A : Public B, Public C{};

3. class A : public B,
virtual public C
{ };

Order of execution:
~~~~~ ~ ~~~~~ ~~~~~

A()

B()

B()  $\rightarrow$  Base 1

C()  $\rightarrow$  Base 2

AC()  $\rightarrow$  Derived.

C()  $\rightarrow$  Virtual base

B()  $\rightarrow$  Non virtual base

AC()  $\rightarrow$  Derived.

Ex:

class alpha

{  
int a;  
}

public:

alpha (int i)

{

a=i;

cout << "alpha";

```
    }  
    void show_x()  
    {  
        cout << a;  
    }  
};  
  
class beta  
{  
    int y;  
public:  
    beta (int j)  
    {  
        y=j;  
        cout << "beta";  
    }  
    void show_y()  
    {  
        cout << y;  
    }  
};
```

```
class gamma : public beta, public alpha  
{  
    int m;  
public:
```

```
gamma (int a, int b, int c, int d):  
    alpha(a), beta(b)
```

{

m=c;

n=d;

cout &lt;&lt; "gamma";

}

void show\_mn()

{

cout &lt;&lt; m &lt;&lt; n;

}

};

int main()

{

gamma g(1,5,6,7);

g.show\_alpha();

g.show\_gamma();

g.show\_mn();

return 0;

}

Op:

beta

alpha

gamma

Note: when you have the parameterized constructor  
so we need to use constructor in derived

4  
5  
6  
7

class.

Abstract class:  
~~~~~ ~~~

It acts like Base class, does not carry any object. It will show the programmer where the inheritance is started. (Starting point of inheritance).

5|9|23:
~~~~~

Templates: (class templates & Function templates).

→ Function overloading:

```
int add (int a, int b)
{
}
int main()
{
    add (2,3);
    add (2.3,4.2);
```

Function templates:  
~~~~~ ~~~~~

template < class T type >

data type used.

return type fun-name (arg)

{

body of fun

}

eg: #

template < class T >

T add (T a, T b) // function.

{

T result = a + b; // T is data type as we
return result; specify it will take.

}

int main()

{

consider int i=2, j=3; // we want to take same

consider as float m=2.3, n=4.2;

cout << add (i,j);

cout << add (m,n);

return 0;

}

- Here the use of Template is, The variable type is not defined initially we can define as T and then we can use it as int or float in main as we want.

- This will be not possible in functions

Class template:

```
template < class T >
class A
{
}
```

} Syntax.

Obj: A< T > a;
 $T \rightarrow$ type

Ex:

```
#include < iostream.h >
class A
{
public:
    int num1=5;
    int num2=6;
    void add()
    {
        num1+num2;
        cout
    }
}
```

int main();

```
{
    A < int > d;
    d.add();
    return 0;
}
```

Ex 2: Multiple parameters.

template < class T1, class T2 >

#include

```
class A
{
    T1 a, T2 b;
public:
    A(T1 a, T2 b)
    {
        a = a; b = b; // reassigned as
        b             // a & b are private.
    }
    void disp()
    {
        cout << a << b;
    }
};

int main()
{
    A<int, float> d(5, 0.5);
    d.disp();
    return 0;
}
```

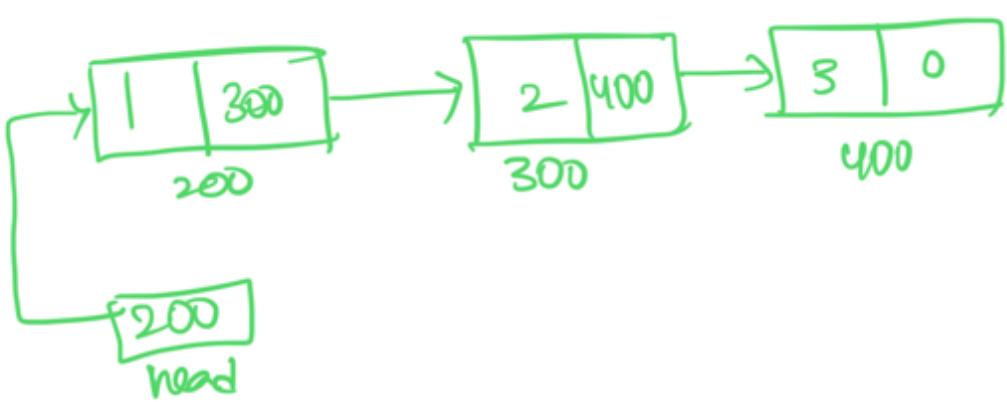
Recursion is self study and question
can be asked in exam

9/9/23;
~~~

```
class node  
{  
public:  
    int data;  
    node *data;  
};
```

```
int main()  
{  
    node *one = NULL;  
    node *two = NULL;  
    node *three = NULL;  
    node *head; *ptr0;  
  
    one = new node();  
    two = new node();  
    three = new node();  
  
    one->data = 1;  
    two->data = 2;  
    three->data = 3;  
  
    one->next = two;  
    two->next = three;  
    three->next = NULL;  
  
    head = one; ptr = head; printlist(head);  
    }  
    return 0;
```

Creating link list  
using class.  
same for structure  
also just class is  
replaced by struct



Traversal:  
algorithm:

(traversal same as  
printing more or less)

Step-1: set  $\text{ptr} = \text{head}$

Step-2: If  $\text{ptr} = \text{NULL}$   
 write 'empty list'  
 Goto step 6  
 end if

Step-3: repeat step 4 & 5 until  $\text{ptr} \neq \text{NULL}$

Step-4: print  $\text{ptr} \rightarrow \text{data}$

Step-5:  $\text{ptr} = \text{ptr} \rightarrow \text{next}$

Step-6: exit.

Same code continues and underlined part  
 was initialized to continue the code.

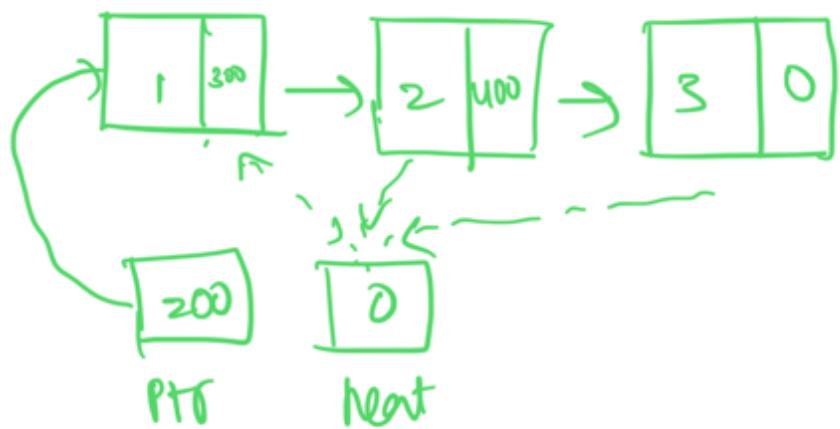
```

void printList (node *n)
{
    ...
}
  
```

```

while (n!=NULL)
{
    cout << n->data;
    n = n->next;
}

```



- Traversal is visiting of each node
  - ↳ Traversing each and every element in a data structure

Inser~~t~~ a node in a l<sup>i</sup>nk list:

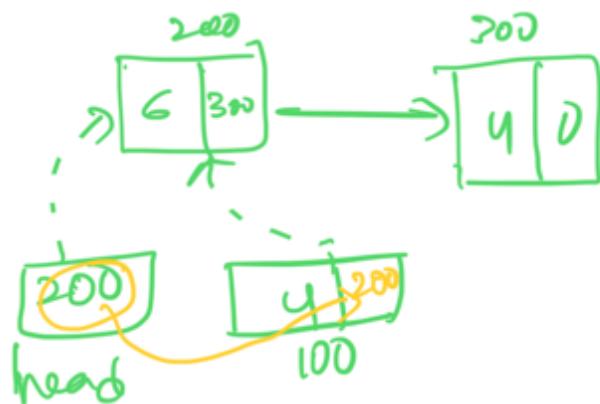
1. At the begining .
2. At the end .
3. At a given location.

① void push (Node \*\* head-ref, int new-data)

```

    Node* new-node = new Node();
    new-node->data = new-data();
    new-node->next = *head-ref;
    *head-ref = new-node;
}

```



```

int main()
{
    push(&head, 1);
    push(&head, 2);
}

```

O/P  
 // Order will be bottom-top  
 // Node 1: 22

12/9/23:  
 ~~~~

Inser~~tion~~ a node at the end:

- when ever you're doing Inser~~tion~~ traversal should be done.

```

void append(Node** head-ref, int new-data)
{
}

```

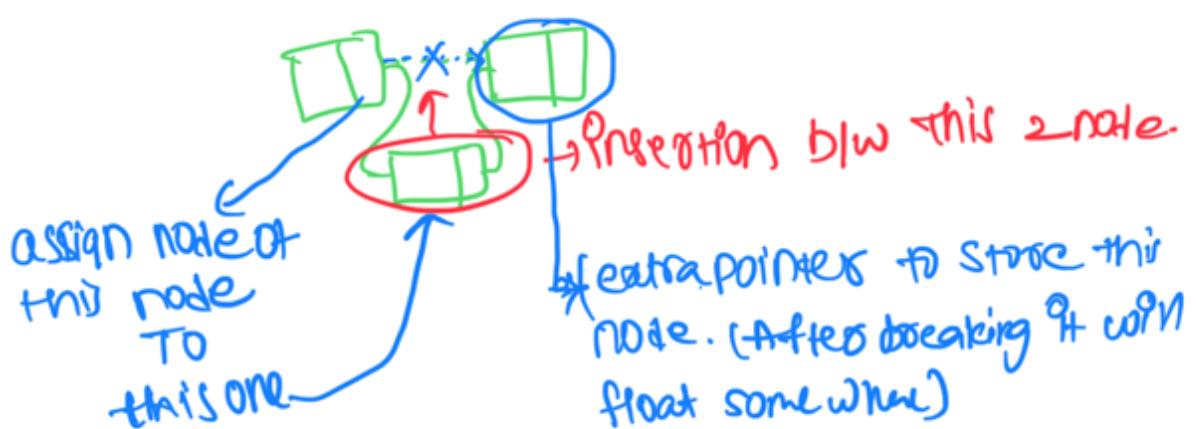
```

node * new_node = newnode();
new_node->data = new_data;
node * last = * head_ref;
new_node->next = NULL;
if (* head_ref == NULL)
{
    * head_ref = new_node;
    return;
}
}

// While loop is
// Traversal
{
    while (last->next != NULL)
    {
        last = last->next;
    }
    last->next = new_node;
}

```

Inserting a node at any given position: (B)W2node;



```

void insertafter( node * prev-node, int new_data)
{
    if (prev_node == NULL)

```

```

cout << "prev-node cannot be NULL";
node *new-node = newnode();
new-node->data = new-data;
new-node->next = prev-node->next;
prev-node->next = new-node;
}

```

How

Try it at which position it can be inserted using another loop

Till tomorrow class mid term syllabus

13/9/23:

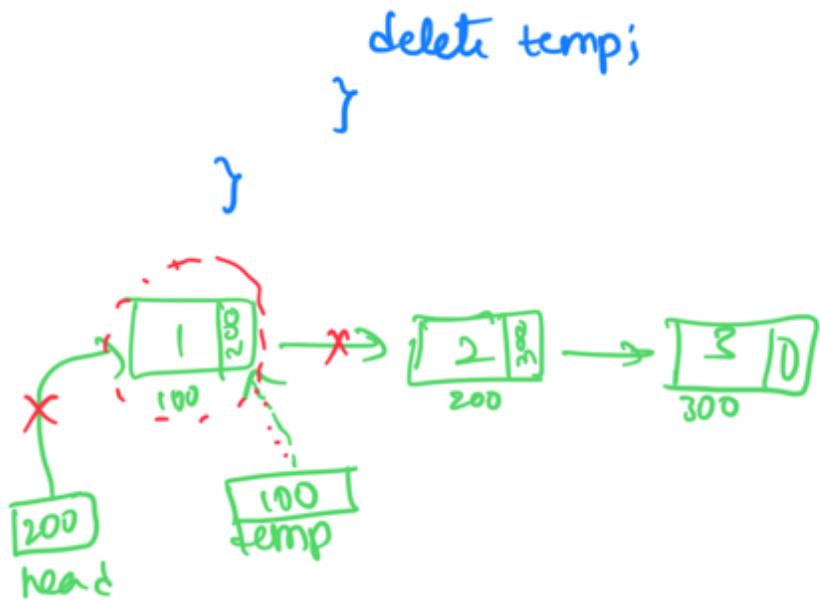
Deleting a node from list:

1. At beginning:

```

void deleteFromBeg()
{
    if (head == NULL)
        cout << "empty list";
    else
    {
        node *temp;
        temp = head;
        head->head = next;
    }
}

```



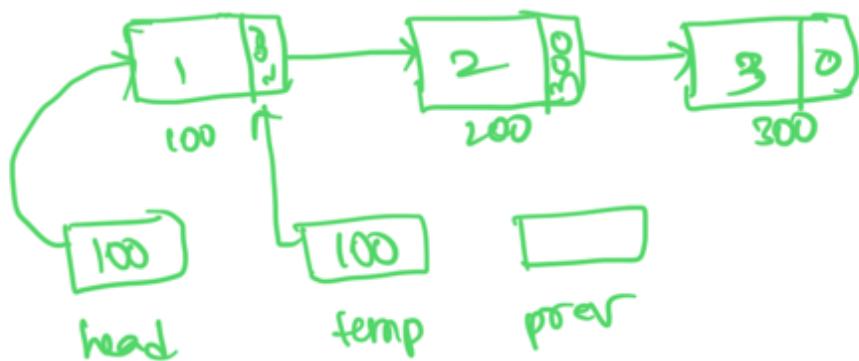
2. At the end:

```

Void deleteFromEnd()
{
    node *temp, *prev;
    temp = head;
    while (temp->next != NULL)
    {
        prev = temp;
        temp = temp->next;
    }
    if (temp == head)
    {
        head = NULL; delete temp;
    }
    else
        prev->next = NULL;
    delete temp;
}

```

}

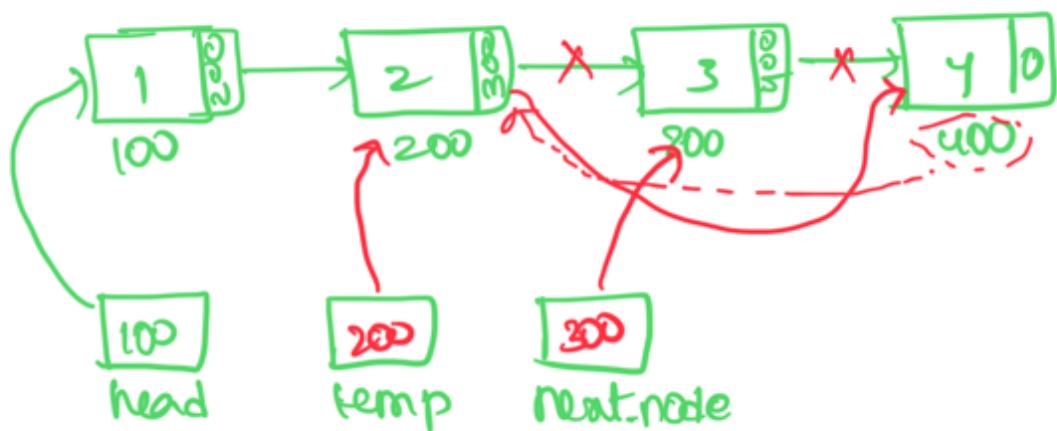


At a given location: (below nodes)

```

void deleteAGivenLocation()
{
    node *temp, *next_node;
    int pos, i=1;
    temp = head;
    if (n > pos)
        while (i < pos - 1)
    {
        temp = temp->next;
        i++;
    }
    next_node = temp->next;
    temp->next = next_node->next;
    delete next_node;
}
  
```

J



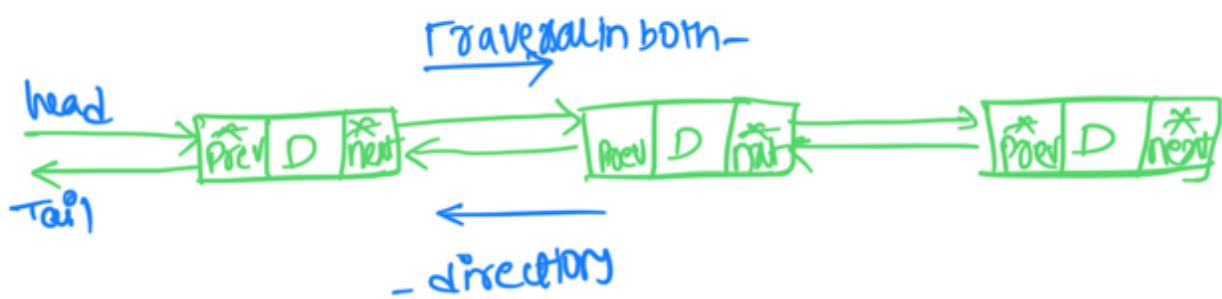
linked list with a tail pointer:

tail pts will also be there then it is
not necessary to use extra pts for traversal.

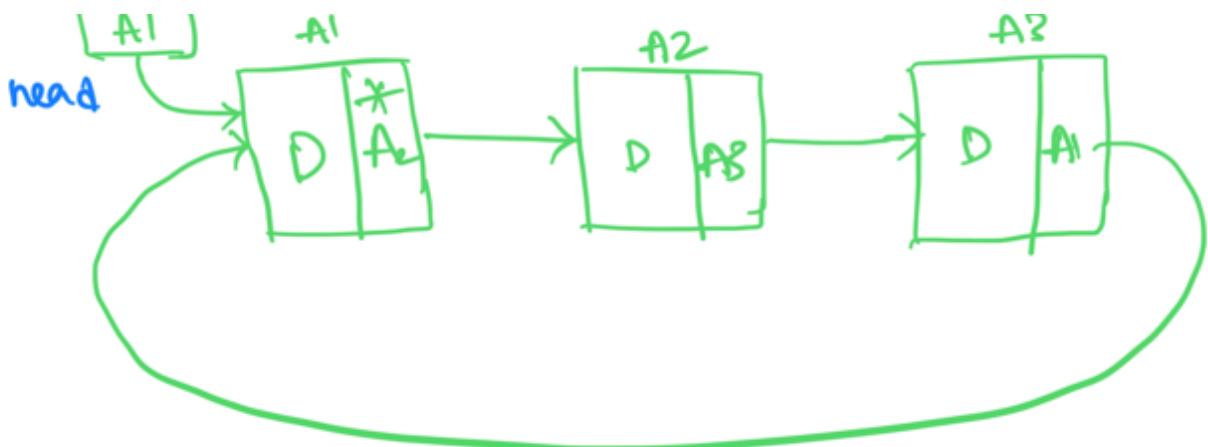
position for midsem

2019/23:

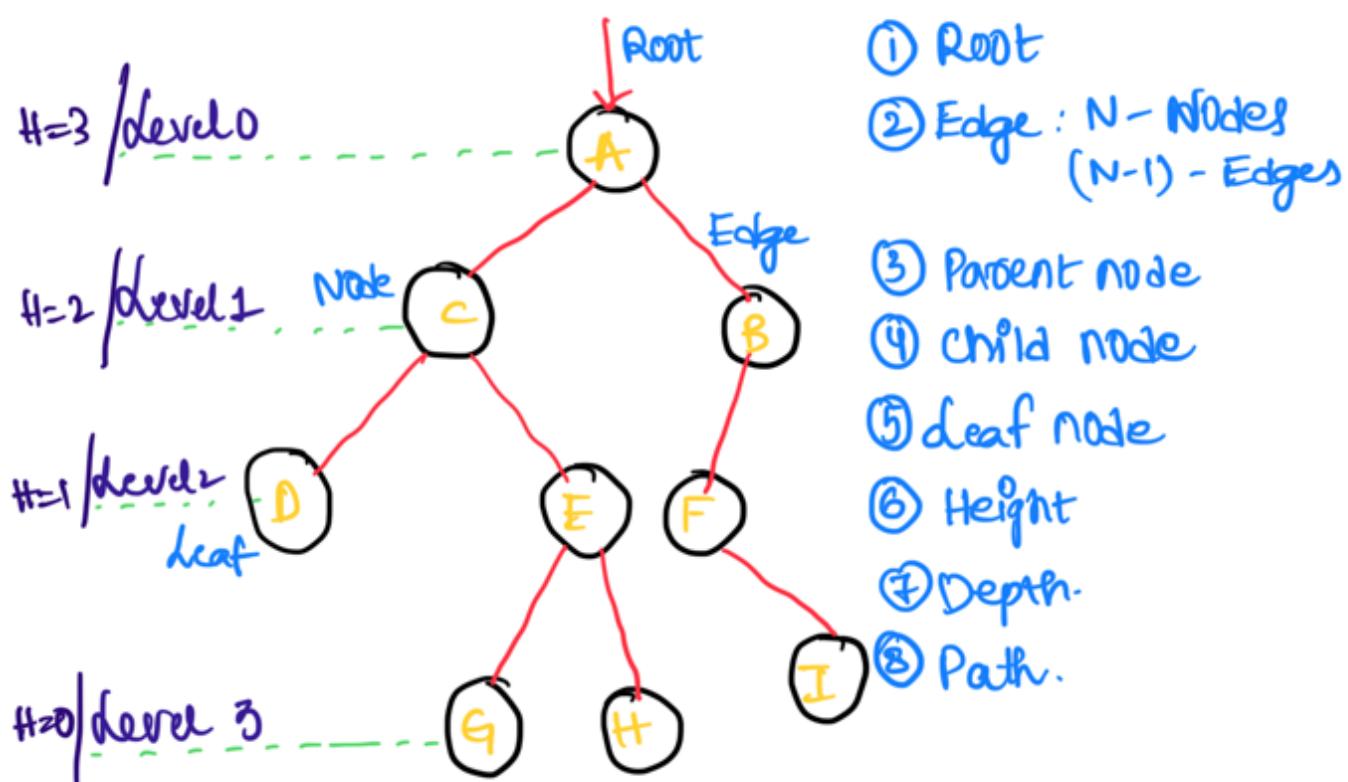
Doubly linked list:



Circular linked list:



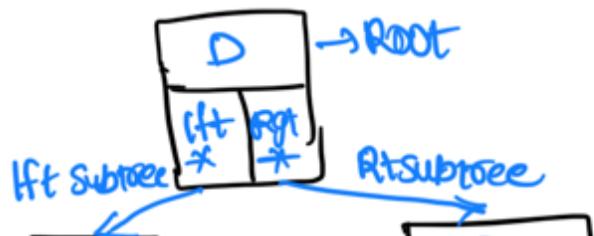
Tree's:



Binary Tree:

when a tree has at most $2^{(0,1,2)}$ leaf nodes. (above one is a binary tree).

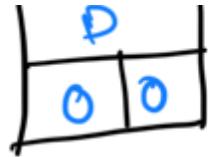
class node
 & Public:



```

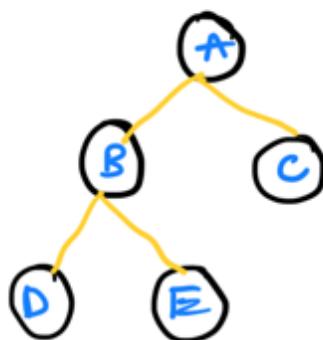
int data;
node *left;
node *right;
};


```



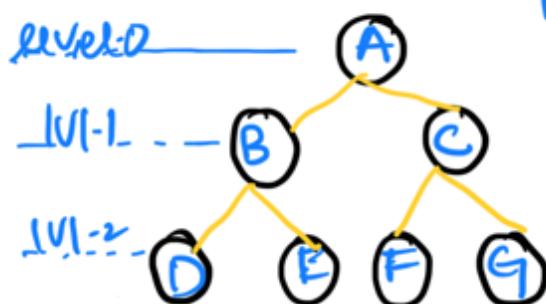
1) Full binary tree:

If a parent node consist of either '0' or 2 child (leaf nodes).



2) Perfect binary tree:

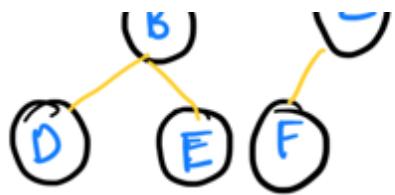
Both left & right Subtrees should be on same level.



3) Complete binary tree:

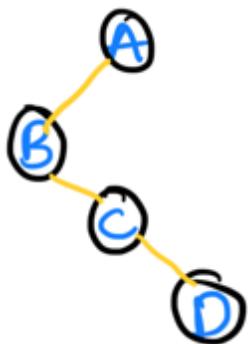
All the nodes should be full else if 1 node is there it should be on left side.





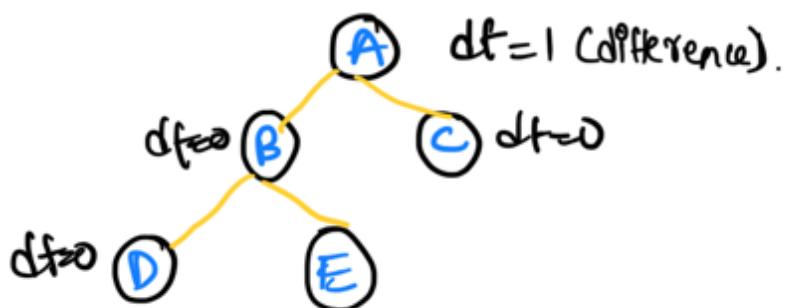
4) Degenerate tree:

$\sim \sim \sim \sim \sim$ either to left or Right (Single node).



5) Balanced tree:

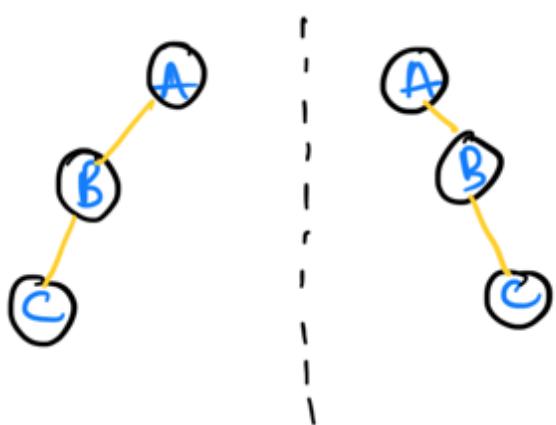
$\sim \sim \sim \sim \sim$



6) Skewed Tree:

$\sim \sim \sim \sim$

Towards left or towards right



3/10/23:
~~~

### Properties of Binary Tree:

- ① If 'n' levels, <sup>max</sup> no. of nodes in that  $Wt = 2^n$ .
- ② If 'h' height, then max no. of nodes  $= 2^{h+1} - 1$ .
- ③ Max height:  $n=h+1$

$$\boxed{h=n-1}$$

Min height:

$$n=2^{h+1} - 1$$

$$\frac{n+1}{2} = n+1$$

$$\boxed{h = \lceil \log_2(n+1) \rceil - 1}$$

### Tree Traversal:

~~~~~ (Visiting each node in linked list)

visiting all subtree's (recursively).

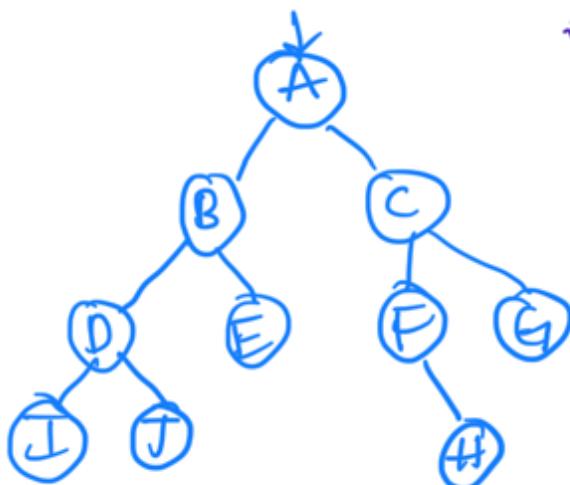
- ① Inorder
- ② pre Order
- ③ post Order.

Inorder: [L - Root - R]

~ or ~ no. all the visit in a left subtree

- (a) first visit all the nodes in the left subtree.
- (b) Then the root node.
- (c) visit all the nodes of right subtree.

Eg:



for code

- inorder (root \rightarrow left)
- preorder (root \rightarrow right)
- postorder (root \rightarrow right)

Printed: I - D - J - B - E - A - F - H - C - G.

It will usually give Ascending order. (Root Only
binary search
trees).

Pre Order: (root \rightarrow left \rightarrow right)

- (a) visit root node
- (b) visit all nodes in left subtree
- (c) visit all nodes in right subtree.

Printed: A - B - D - I - J - E - C - F - H - G

Post Order:

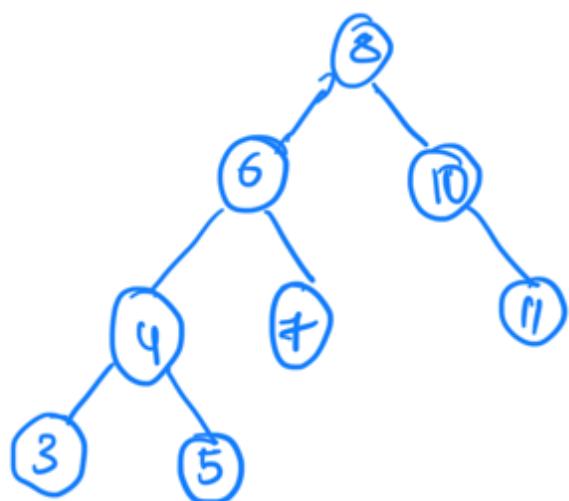
- (a) visit all nodes in left subtree.
- (b) visit all nodes in right subtree.
- (c) visit root node.

▼ ▼ ▼ ▼ ▼ node

Pointed: I - J - D - E - B - H - F - G - C - A

Binary Search tree:
~ ~ ~ ~ ~

sp1 - ① left subtree $< \text{root}$
property Right subtree $> \text{root}$

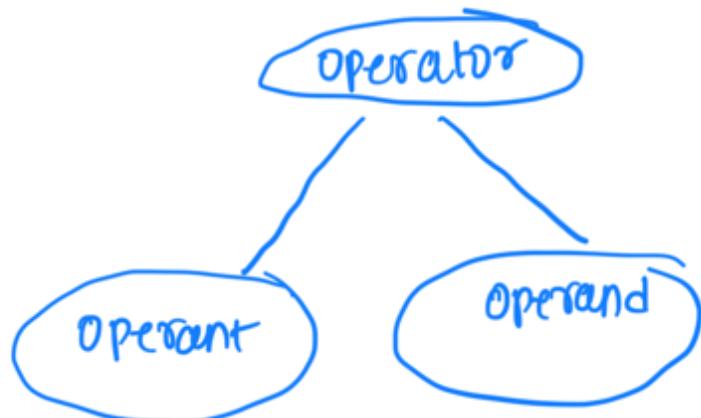


—

11/10/23:
~~~~~

Binary expression Tree:

- ① Infix:  $\rightarrow$  Lt - Root - Rt  
 $\langle$  operand  $\rangle$   $\langle$  operator  $\rangle$   $\langle$  operand  $\rangle$
- ② Prefix:  $\rightarrow$  Root - Lt - Rt  
 $\langle$  operator  $\rangle$   $\langle$  operand  $\rangle$   $\langle$  operand  $\rangle$
- ③ Postfix:  $\rightarrow$  Lt - Rt - Root  
 $\langle$  operand  $\rangle$   $\langle$  operand  $\rangle$   $\langle$  operator  $\rangle$



Q) construct the binary expression tree using infix.

$a * b / c + e f * g + k - a * j$ .

least precedence.

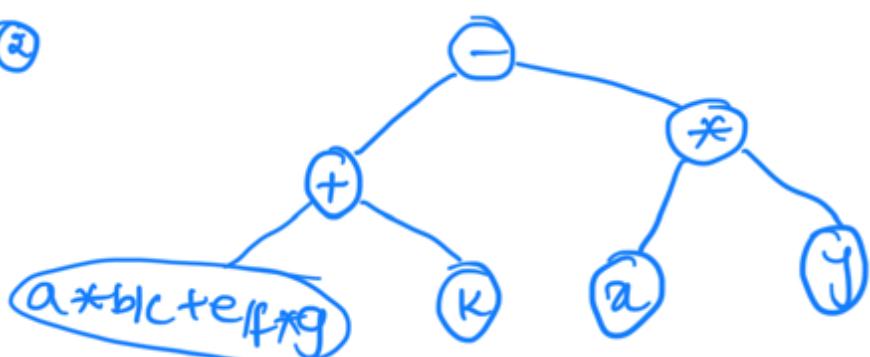
A) left to right.

The least precedence is root node.

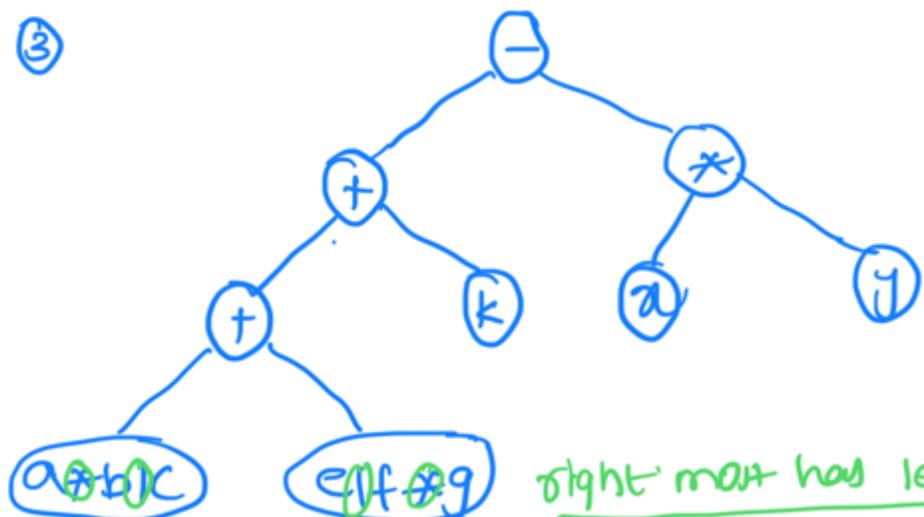
Step:



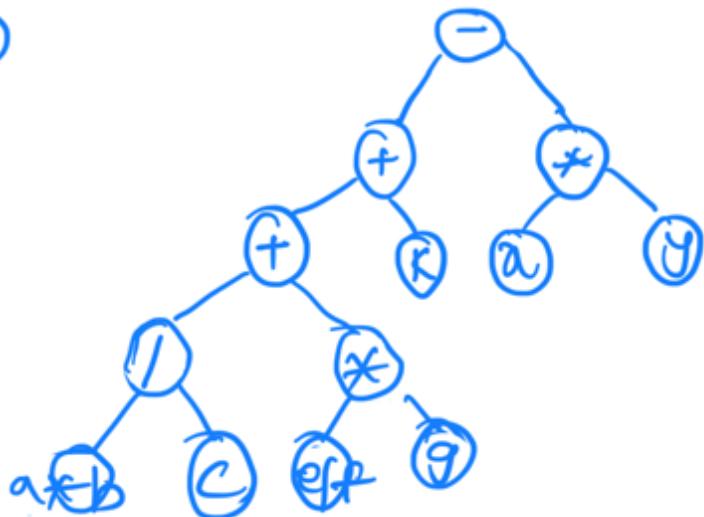
②



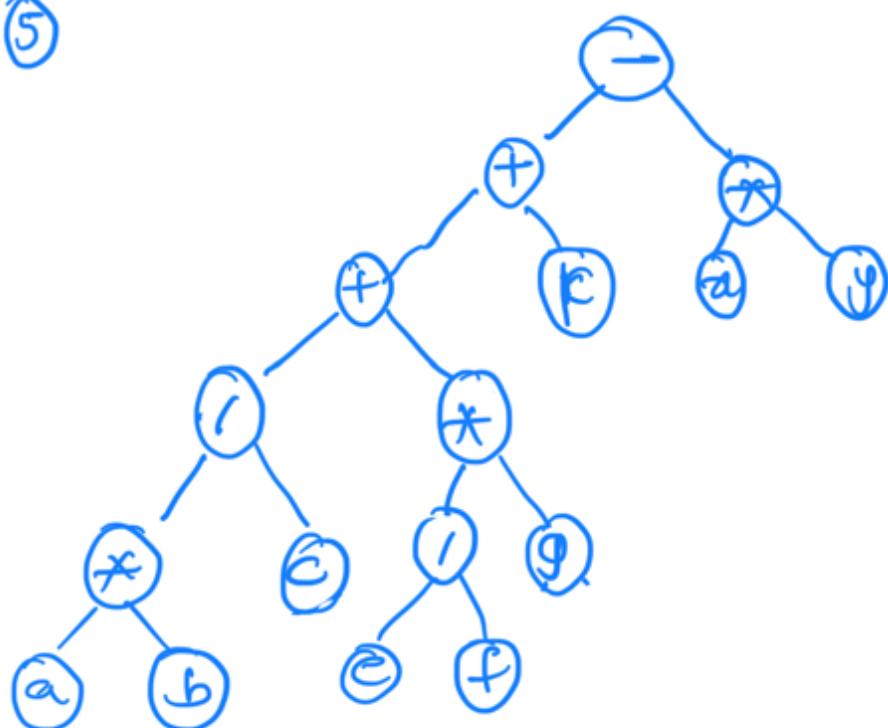
③



④



⑤



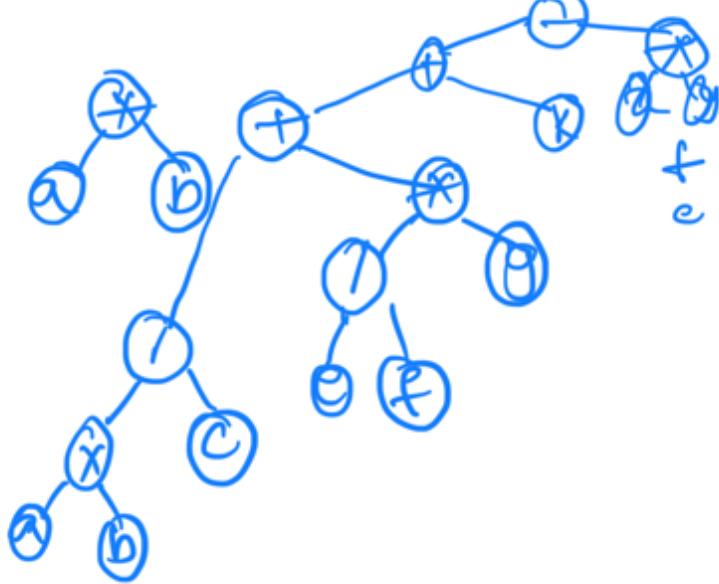
Prefix = - + + / \* a b c \* / c + g k \* a y  
(root - lt - rt)

Postfix = a b \* c / e + / g \* + k + a y \* -  
(lt - rt - root)

Q) construct a binary expression tree using a post fix expression.

① LIFO

ab\*x\*c / e + f / g \* h + k + l \* m - n



②



14/10/23:

conversion of infix to postfix:

without stack: (precedence of operators consider take high precedence  $\Rightarrow$ )

Eg:  $A + B * C \rightarrow$  INfix

postfix → ?

<operator><operator><operator>

①  $A + B * C$

②  $A B C * +$

I/P	Stack	O/P
A	(empty)	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC*+

- ⑧ convert the infix expression to postfix using stack

$$K + L - M * N + (O \wedge P) * W \downarrow V \downarrow V \downarrow T \downarrow Q.$$

A)

IP	Stack	OP
K	(empty)	K
T	+	K
L	+	KL
-	-	KL+
M	-	KL+M
*	-*	KL+M
N	-*	KL+MN
T	+	KL+MN*-
(	+()	KL+MN*-
)	+()	KL+MN*-D
N	+()	KL+MN*-D
P	+()	KL+MN*-DP
)	+	KL+MN*-DPN
*	+*	KL+MN*-DPN
W	+*	KL+MN*-DPNW
/	+/	KL+MN*-DPNW*
V	+/	KL+MN*-DPNWV*
/	+/	KL+MN*-DPNWV*

Braces not included in postfix

V	' /	KL+MN * -OP NW*UN
*	+*	KL+MN * -OP NW*UNV/
T	+*	KL+MN * -OP NW*UNVIT
+	+	KL+MN * -OP NW*UNVIT*
Q	+	KL+MN * -OP NW*UNVIT*Q+

conversion of Infix to prefix  
 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

without Stack:  
 ~ ~ ~

eg: A + B \* C → Infix

prefix <operator> <operand> <operand>.

① A + \* BC

② + A \* BC

with stack:  
 ~ ~ ~

① Reverse Infix exp

② Reverse DIP at last.

DIP	Stack	Op	C * B + A
C	(empty)	C	
*	*	C	
B	*	CB	
+	+	CB *	
A	+	CB * A	
		CB * A +	↑

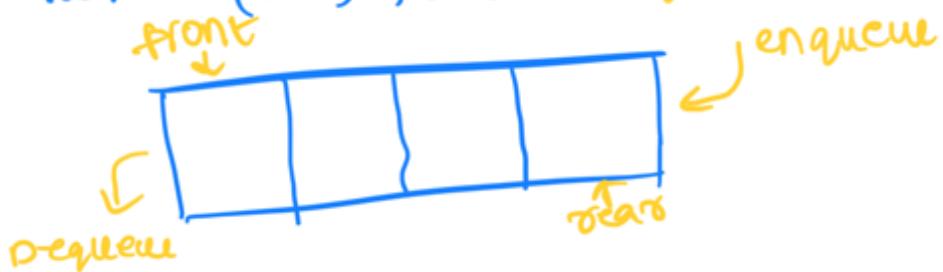
010 + A\*B\*C

queue:

FIFO

we need 2 pointers.

Insert: (tail)  $\rightarrow$  rear enqueue



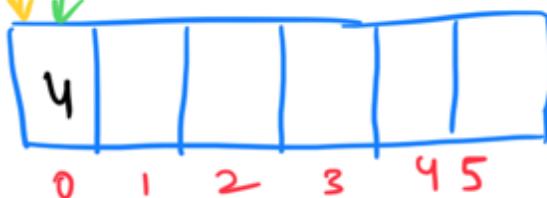
Delete: (head)  $\rightarrow$  Front, rear dequeue.

rear = size - 1  
Front = 0

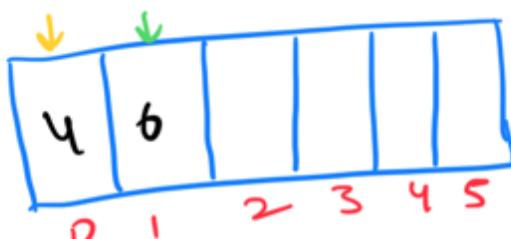
(front = rear = -1)

Empty

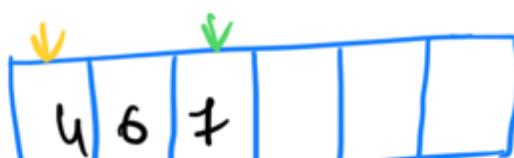
front  $\leftarrow$



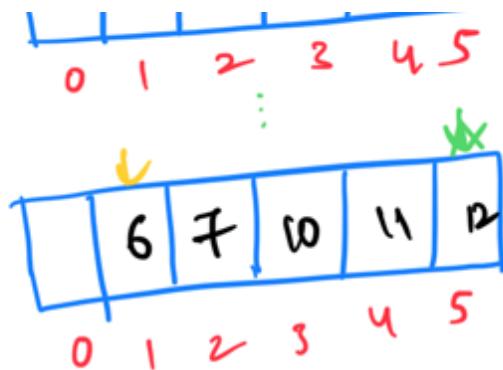
enqueue(4)



enqueue(6)



enqueue(7)



dequeue()  
front = front + 1

- ① Linear queue
- ② circular queue (Ring buffer)
- ③ priority queue
- ④ Dequeue

*for*  
Algorithm *Enqueue operation* (Linear queue)  
*www www www*

```
Procedure enqueue(data)
    If queue is full
        Return overflow
    End If
    rear = rear + 1
    queue[rear] = data
    Return true
End procedure.
```

*Algorithm for Dequeue operation: (Linear queue)*  
*www www www*

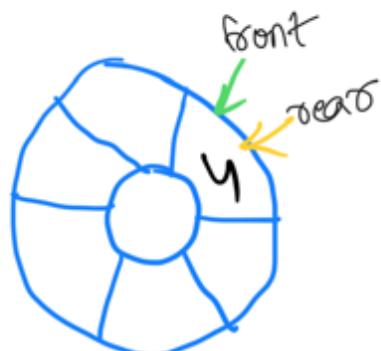
```
Procedure dequeue
    Or ..... Program
```

```

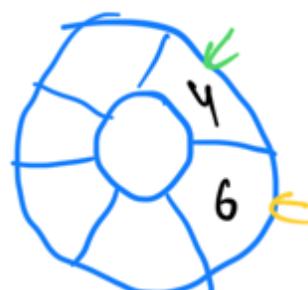
if queue is empty
    return underflow
end if
data = queue(front)
front = front + 1
return true
end procedure.

```

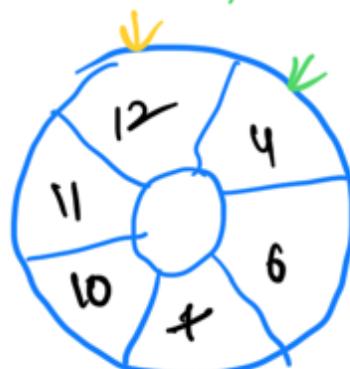
Circular Queue:



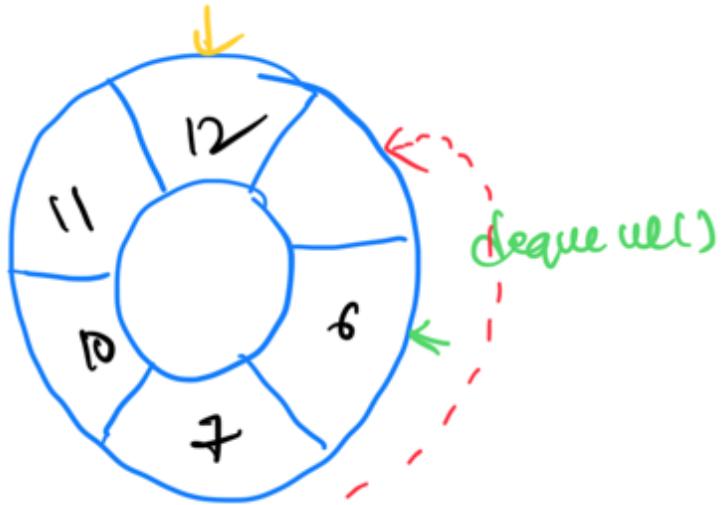
enqueue(11)



enqueue(6)



queue is full.



we can we enqueue again and that will assign here--

27/10/23:

circular queue:

Enqueue Algorithm:

① Check if queue is full

check( $\text{rear} \leq \text{size} - 1$  &  $\text{front} = 0$ )

② If it's true, display queue is FULL

③ If not, check ( $\text{rear} == \text{size} - 1$  &  $\text{front} != 0$ )  
if it's true, set  $\text{rear} = 0$  & insert element.

Dequeue Algorithm:

① If  $\text{front} == \text{rear} == -1$

size = 1

wrote 'underflow'

GOT Step 4

[End of if]

② set val=queue [front]

③ If front=rear

set front=rear=1

else if front=Size-1

set front=0

else

Set front=front+1

[End of if]

[End of if]

Priority Queue

~~~~~

1. ascending order priority

[Highest priority will get]
dequeued first

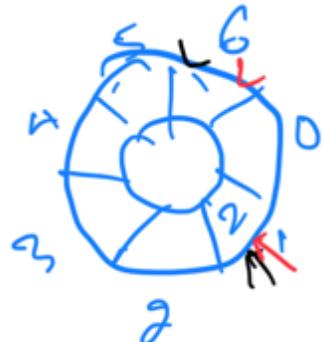
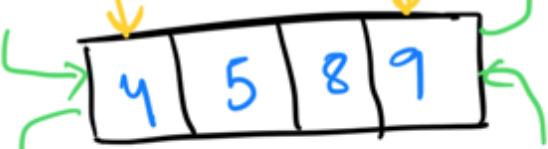
2. descending order priority

Double ended Queue (Deque)

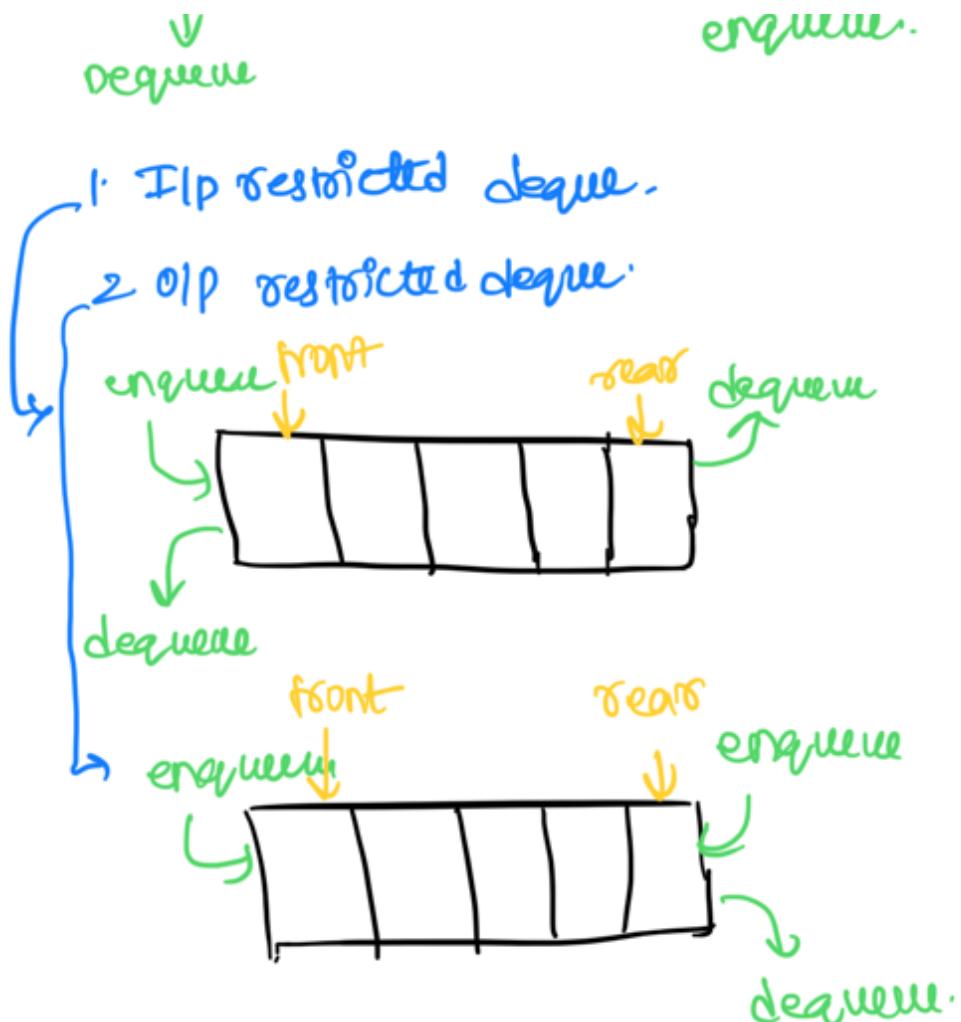
enqueue front

rear

dequeue



front = rear
front = -1



operations:

- ① Insertion at front.
- ② Insertion at rear.
- ③ Deletion at front.
- ④ Deletion at rear.

①

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 4 | 5 | 6 | | |

check queue is full?

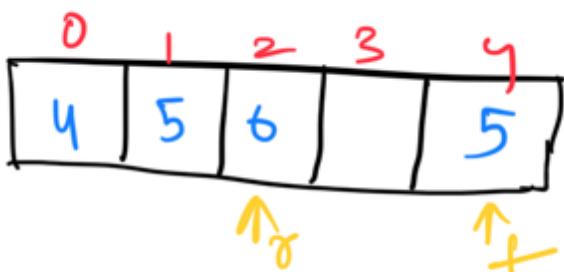
If $\text{front} = \text{rear} = -1$

Computer Fund

Set front = rear = -1 [Initializes array]

If front < 1

Set front = size - 1



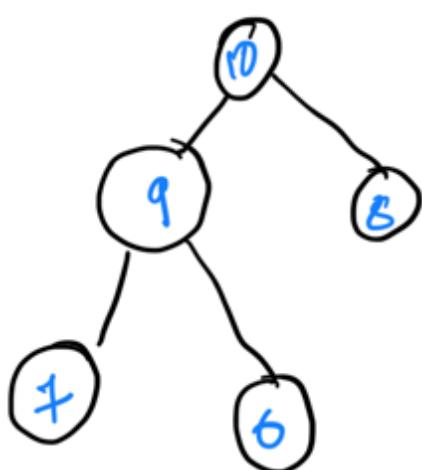
- ② Same as Insertion (enqueue) } Linear arr.
- ③ Same as deletion (dequeue)

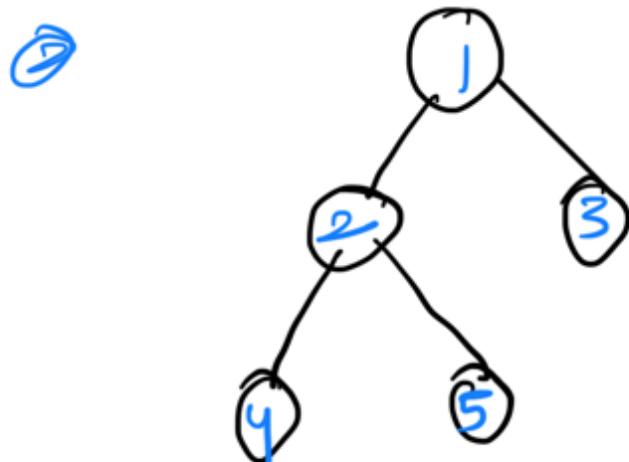
Heap Data Structure:

complete binary tree: (follow)

- ① Max heap (root node will have max value)
- ② Min heap (root node will have min value)

①





Operations:

- ~ ~

① heapify

② Insertion

③ Deletion-

3|1|1|23:
~ ~ ~

heapify algorithm: (Max heap)

Heapify (array, size, i)

Set i as largest

left child = $2^i + 1$

right child = $2^i + 2$

if $\text{left child} > \text{array}[\text{largest}]$

set left child index as largest

if $\text{right child} > \text{array}[\text{largest}]$

set right child index as largest