DS using C++

22/8/23:

last bit of class { Destructors, constructors } I may talk about

Operator overloading:

cannot be overloaded ← { . - dot operator
?: - conditional operator
sizeof - size operator
:: - Scope resolution. }

Overloading - same operator different functions.

Syntax

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

{
body of the function
}

→ fun - no arg
→ fun - 1 arg

Unary operator:

→ operand

Op a (or) a op → operator

++ a (or) a ++

Binary operator:

→ operand

→ fun - 1 arg
→ fun - 2 arg

a op y

→ operator

a + y

Overloading of unary operator:

(unary op as normal function)

```
class Space
{
    int x,y,z;
    public:
        void getdata (int a,int b,int c)
        {
            x=a;
            y=b;
            z=c;
        }
        void disp()
        {
            cout << x<<y<<z;
        }
        void operator -(); // declaration
};
void Space :: operator -();
{
    x=-x;
```

return type should be specified ←

```
        y=-y;
        z=-z;
    }
int main()
{
    space s;
    s.getdata(-10,20,30);
    s.disp();
    -s;
    s.disp();
    return 0;
}
```

there go it will Invoked and

(unary operators friend function)

```
class space
{
  public:
    friend void operator -(space);
};
  void operator -(space D)
  {
    D.x = -D.x;
```

operator → key word

```
    D.y = -D.y;
    D.z = -D.z;
}
```

Binary operator overloading:

(normal function)

```
class complex
{
    float x;
    float y;
    Public:
        complex() { }
        complex(float real, float img)
        {
            x = real;
            y = img;
        }
```

It will goes here as 't' is invoked

```
complex operator +(complex c)
{
    complex temp;
    temp.x = x + c.x;
    temp.y = y + c.y;
```

```
        return temp;
    }

    void disp();
    {
      cout << x << "+j" << y;
    }
};

int main()
{
    complex c1, c2, c3;
    c1 = complex(2.5, 4.3);
    c2 = complex(1.0, 2.3);
    c3 = c1 + c2;
    c1.disp();
    c2.disp();
    c3.disp();
    return 0;
}
```

(friend function)

class space

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

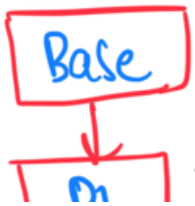
23/8/23:

Inheritance:

Parent → Base class.
The one inherited → derived class

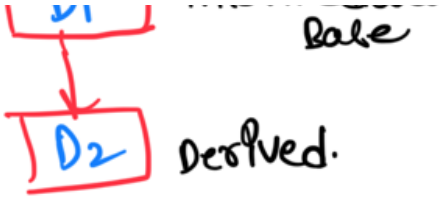
1. single

2. Multilevel →

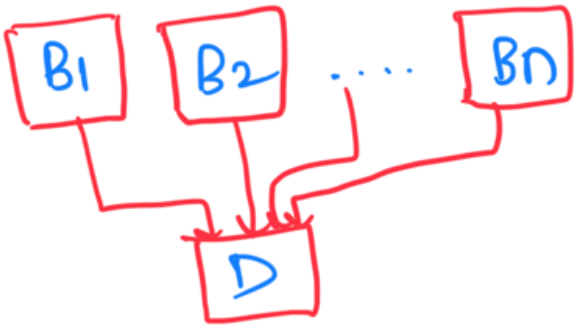


Base

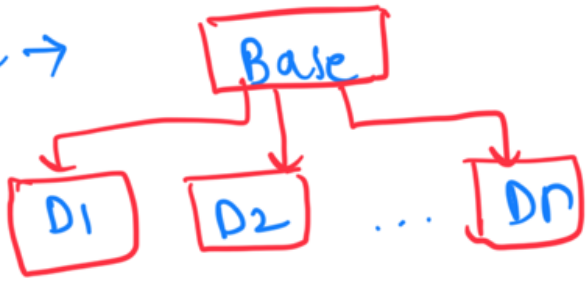
intermediate



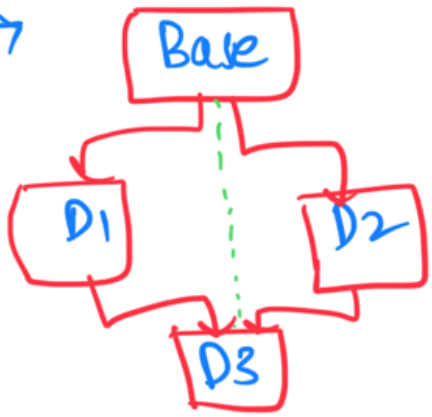
3. Multiple →



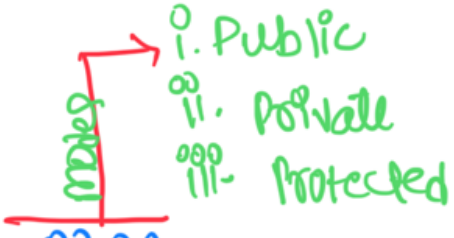
4. Hierarchial →



5. Hybrid →



Defining a derived class:



class derived class name : visibility base class name

{

Body of derived class;

};

eg:

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:

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

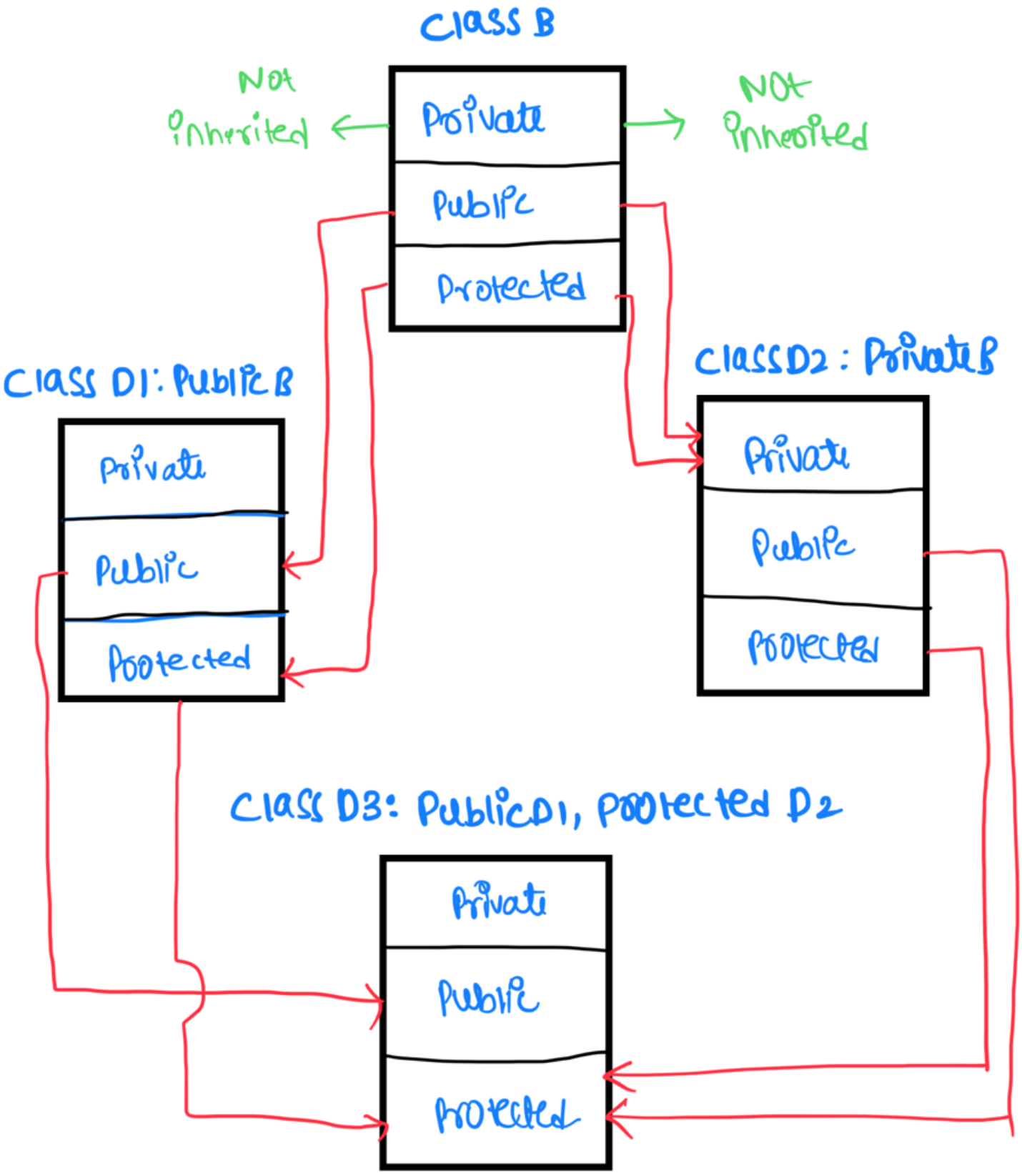
```
class XYZ
{
  int a; // not inheritable
  public:
    void get-a();
};
class ABC : public XYZ
{
    int a;
    public : int y;
    void disp();
```

```
}

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 a;
  Public:
    int b;
    void get_ab()
      {
        a=5;
        b=10;
      }
    int get_a()
      {
        return a;
      }
```

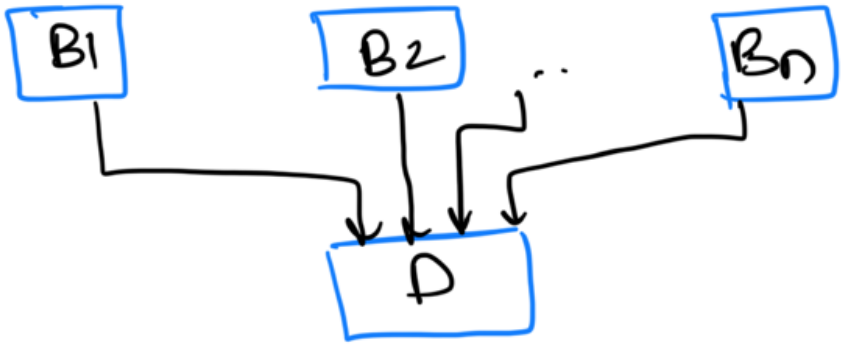
```
        void show_a()
          {
           cout<<a;
          }
    };

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

int main()
  {
     Dd;
```

```
d. get_a();
d. mul();
d. show_a();
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

{

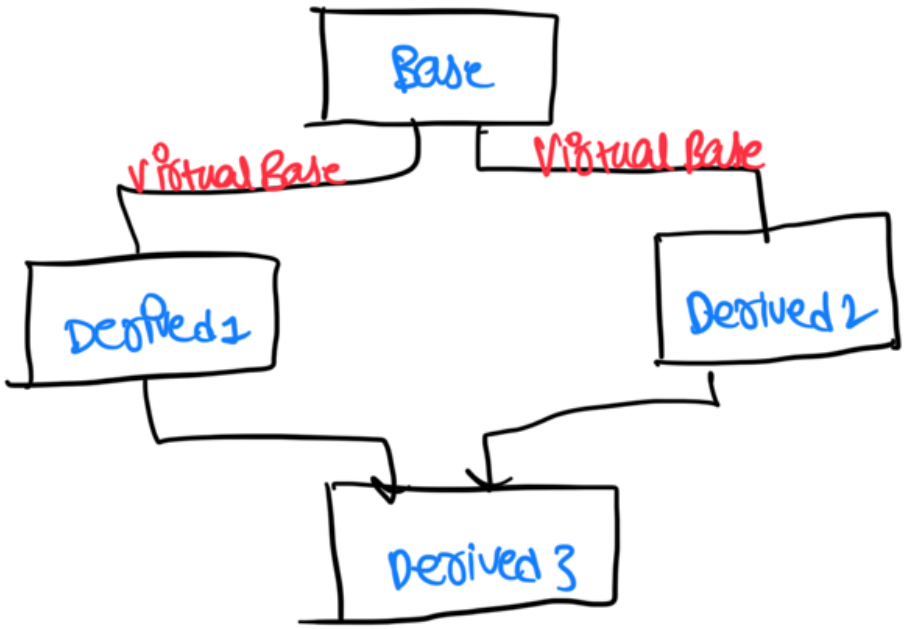
```
~
    Protected:
        int M;
    Public:
        void get_m(int);
    };
class N
  {
    Protected:
        int n;
    Public:
        void get_n(int);
  }
  class P : Public M, Public N
    {
      Public:
          void display();
      }
      void M:: get_m(int a)
       {
          m=a;
        }
```

```
void N:: get_n(int b)
  {
    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;
  }
```

Multilevel Inheritance: ?

Hierarchial: ?

Hybrid :


Base
Virtual Base
Virtual Base
Derived 1
Derived 2
Derived 3

- If we use virtual Base, then only one copy of member functions 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;
```

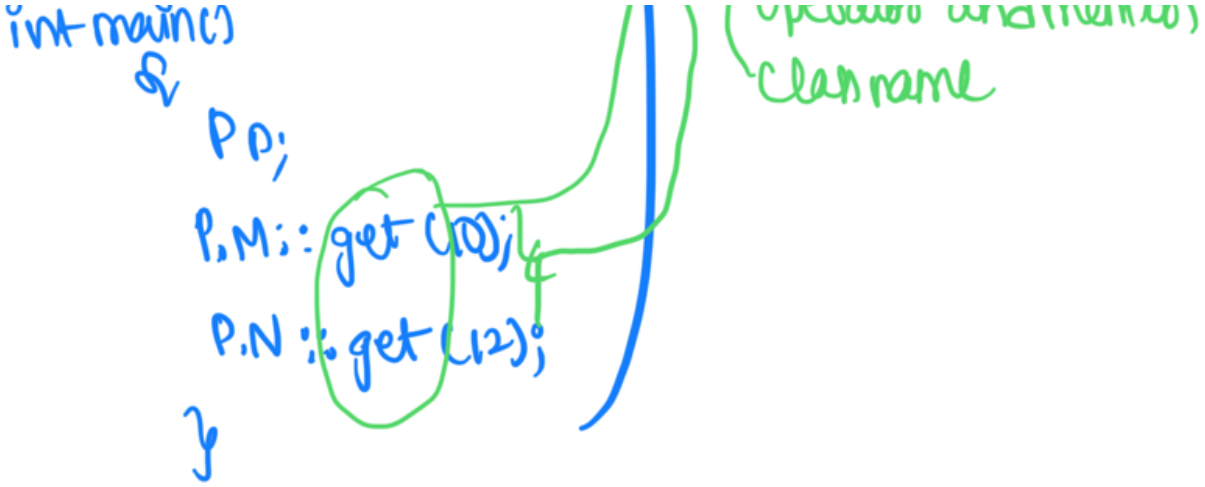
Note: when we feel any duplication happening then we need to use virtual (It may be base class, next class that should be decided by programmer)

```
void M :: get (int a)
{
  m=a;
}
void N :: get (int b)
{
  n=b;
}
```

Same function name can be used in two classes.

BUT

we need to use scope resolution operator as in the



30/8/23:

constructors in derived class:

ex:
```
class D : public A, public B
{ int d1;
  public:
    D(int a, int b, float c, float d, int e) : A(a,b), B(c,d)
      {
        d1=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 constructor:

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() → Base 1
C() → Base 2
A() → Derived.

C() → Virtual base
B() → Non Virtual base
A() → Derived.

Ex:

```
class alpha
{
  int a;
  Public:
    alpha (int i)
      {
        a=i;
        cout<<"alpha";
```

```
    }
    void show_a()
    {
      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<<"gamma";
  }
  void show_mn()
    {
      cout<<m<<n;
    }
};

int main()
  {
    gamma g(4,5,6,7);
    g.show_ab();
    g.show_xy();
    g.show_mn();
    return 0;
  }
```

O/P :

beta
alpha
gamma

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

4 class.

5

6

7

Abstract class:

It acts like Base class, doesnot carry any object. It will show the programmer where the inheritance is started. (starting point of inheritance).

5/9/23:

Templats: (class templats & Function templats).

→ Function overloading:

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

Function templates:

template < class Ttype > → data type used.

```
returntype 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 specify it will take.
    return result;
}
int main()
{
    int i=2,j=3; // we want to take same data type
    float m=2.3, n=4.2;
    cout << add (i,j);
    cout << add (m,n);
    return 0;
}
```

consider as int

consider as float

- 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 → type

Ex:

```
# Template < class T >
   class A
   {
      Public:
      T num 1=5;
      T num 2 =6;
      void add()
      {
         num1+num2;
      }
   };
Int main();
   {
      A < Int >d;
      d.add();
      return 0;
   }
```

}

Eg 2: Multiple parameters.

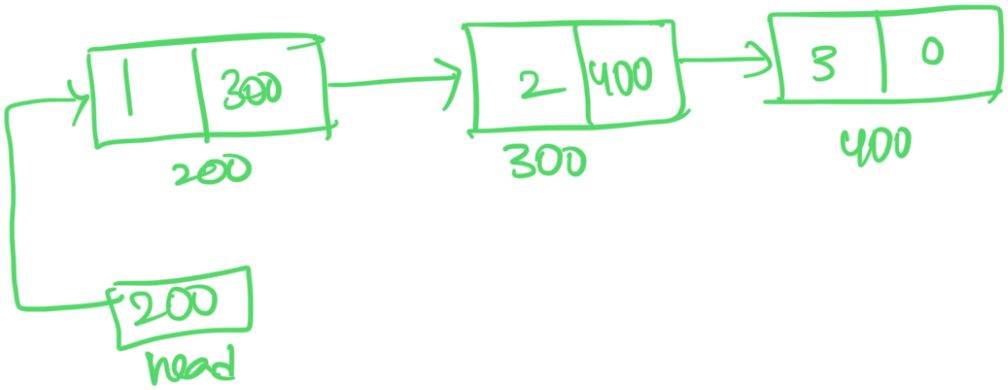
```
template < class T1 , class T2 >

#Include

   class A
    {
      T1 a, T2 b;
      Public;
        A(T1 a, T2 y)
         {
           a=x ; b=y; // reassignment as
                        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:

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

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

int main()
 {
    node *one = NULL;
    node *two = NULL;
    node *three= NULL;
    node *head; *ptr;

    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);
  }  return0;
```



Traversal algorithm:

(traversal same as printing nodes only)

Step-1: set ptr = head

Step-2: If ptr = NULL
write 'empty list'
Goto step 6
endif

Step-3: repeat step4 & 5 until ptr != NULL

Step-4: print ptr → data

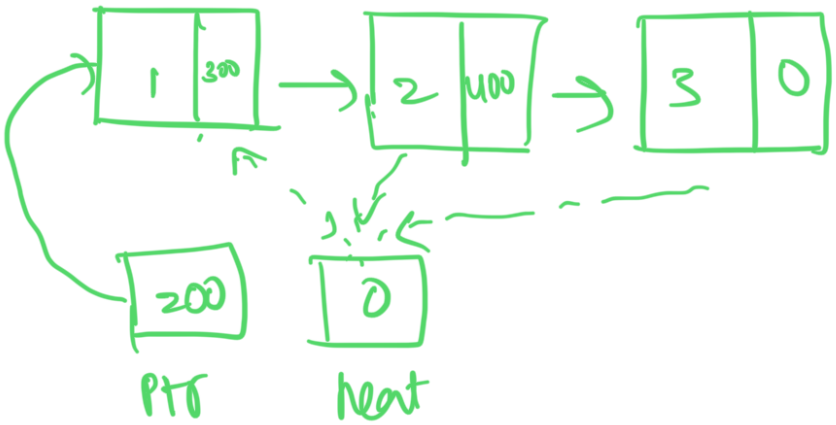
Step-5: ptr = ptr → next

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

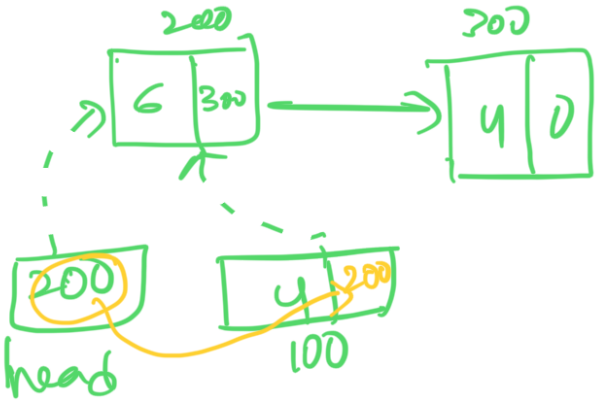
Insert a node in a link list:

1. At the beginning.
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

// here i 2 1

12/9/23:

Insertion a node at the end:

- when ever you're doing insertion traversal should be done.

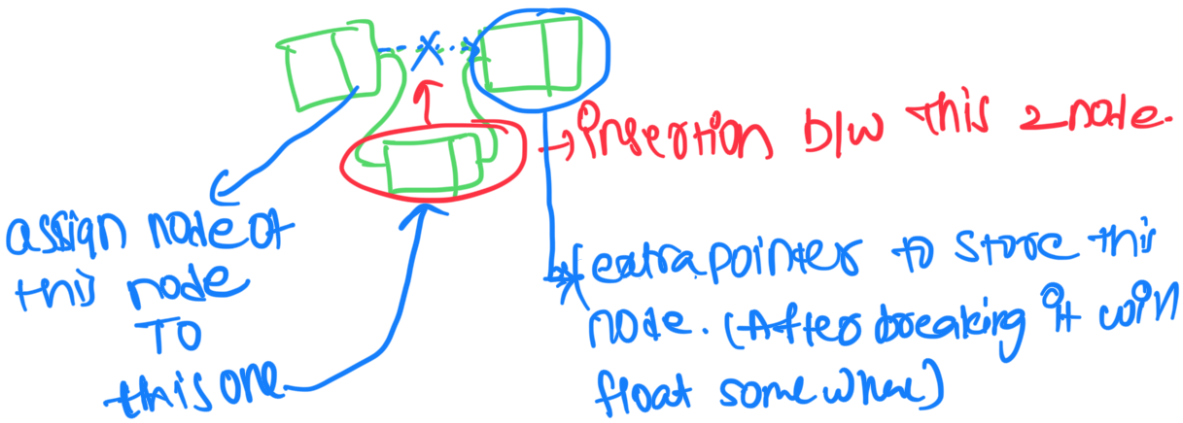
```
void append(node **head_ref, int new_data)
{
```

```
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;
      return;
    }
    last->next = new_node;
```

Inserting a node at any given position: (B/W 2 nodes)




```
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;
}
```

Hw

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

Till tommorow class mid term syllabu

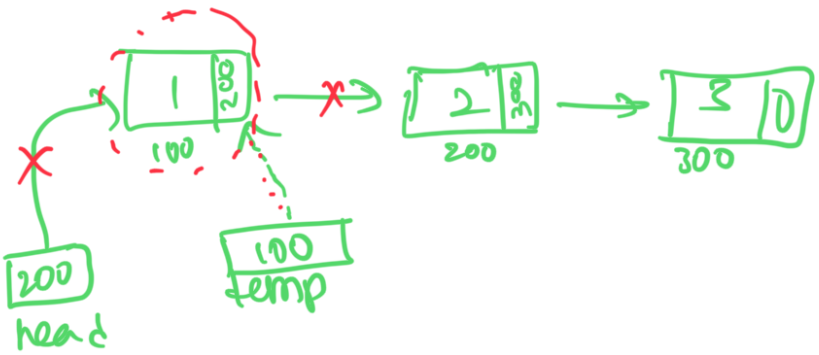
13/9/23:

Deleting a node from list:

1. At beginning:

```
void delete from beg()
{
  if (head == NULL)
    cout << " empty list";
  else
    {
      node *temp;
      temp=head;
      head -> head = next;
```

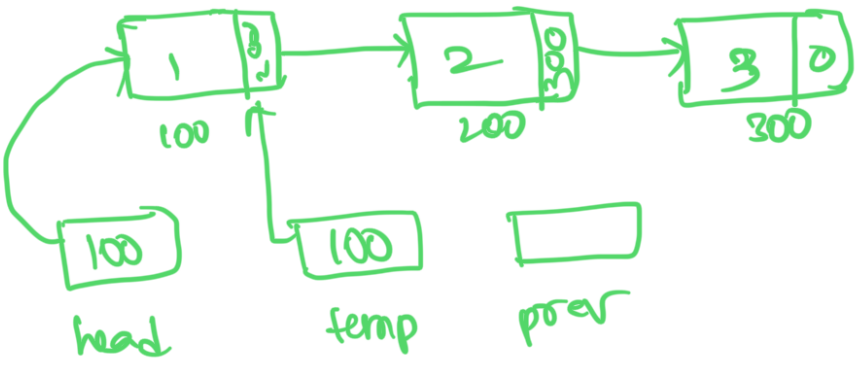
```
        delete temp;
    }
}
```



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;
```

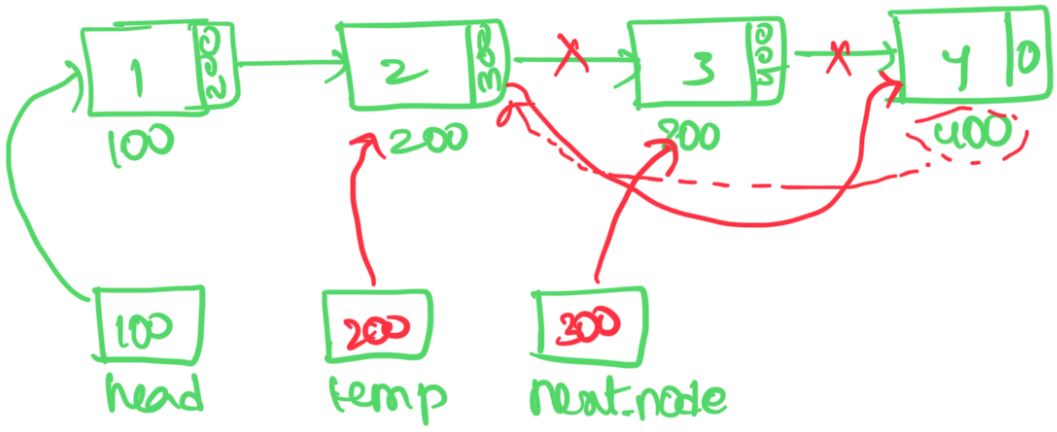
```
}
```



At a given location: (b/w nodes)

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

J



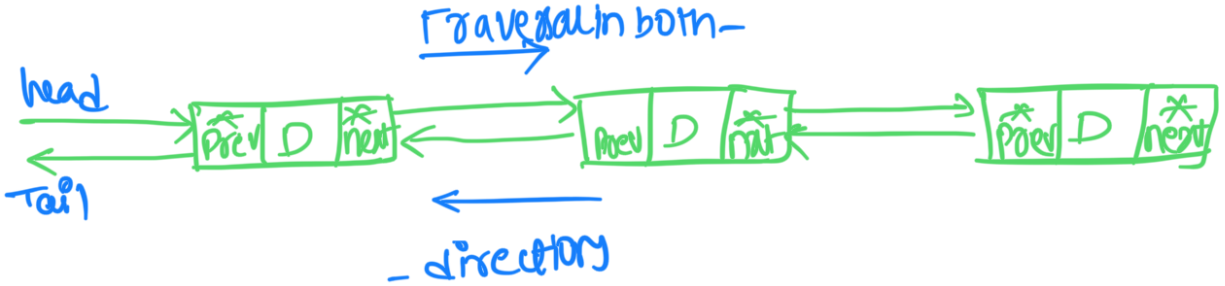
Linked list with a tail pointer:

tail pts will also be there then ft is not necessary for using extra pts for traversal.

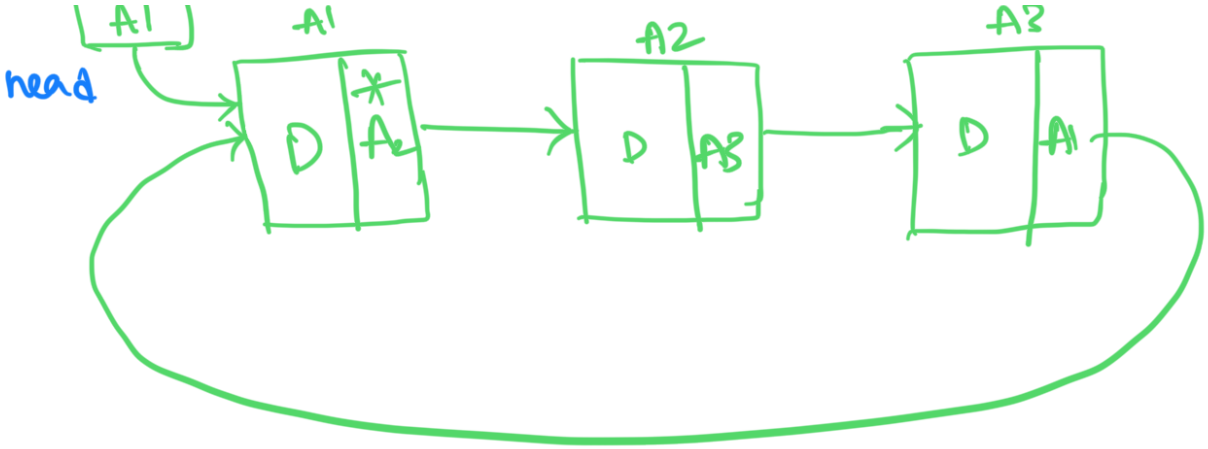
portion for midsem

2019/23:

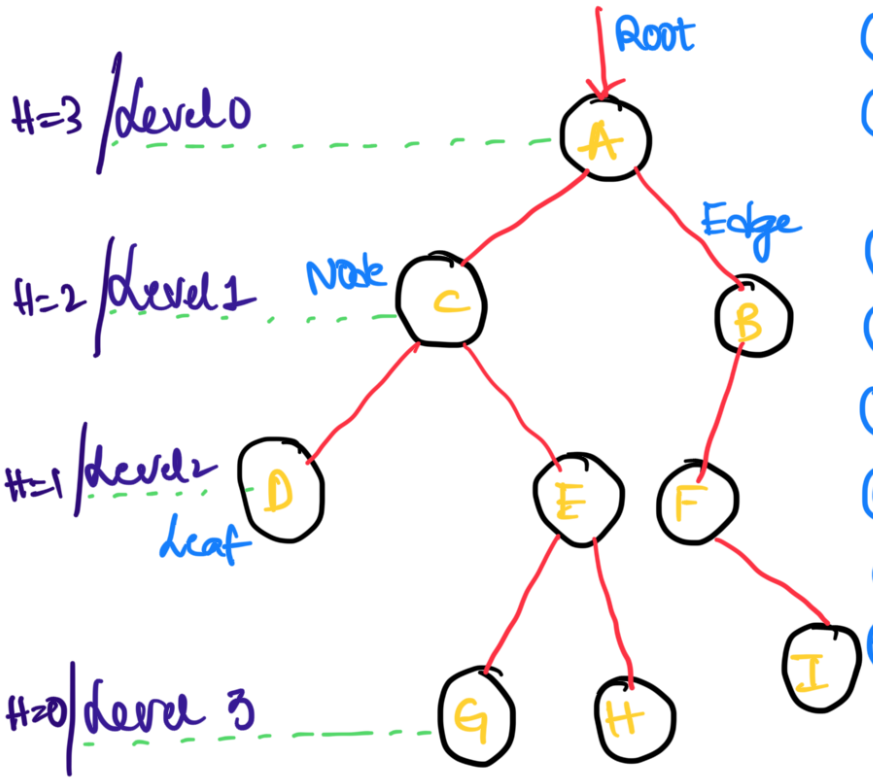
Doubly linked list:



circular linked list:



Tree's:

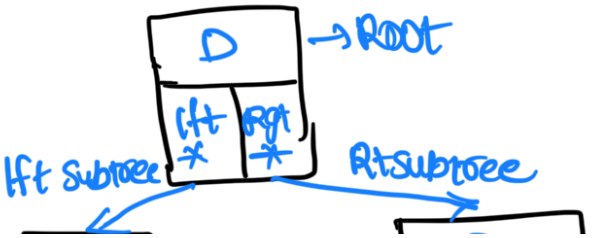


① Root
② Edge : N - Nodes
(N-1) - Edges
③ Parent node
④ child node
⑤ Leaf node
⑥ Height
⑦ Depth.
⑧ Path.

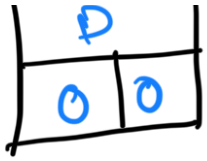
Binary Tree:

When a tree has atmost 2 (0, 1, 2) leaf nodes. (above one is a binary tree).

class node
{
public :

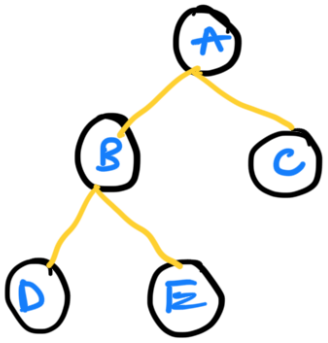


```
int data;
node *left;
node *right;
};
```



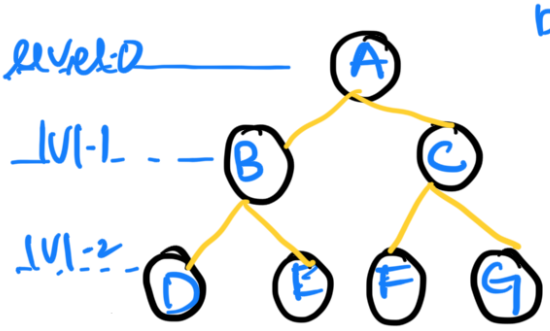
1) Full binary tree:

It a parent node cosist of either '0' or '2' child / leaf nodes.



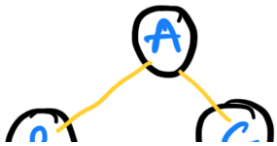
2) Perfect binary tree:

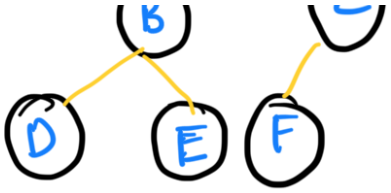
Both of Lt subtrees should be on same level.



3) Complete binary tree:

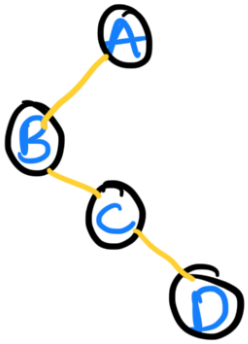
All the nodes should be full else if node is there it should be on left side



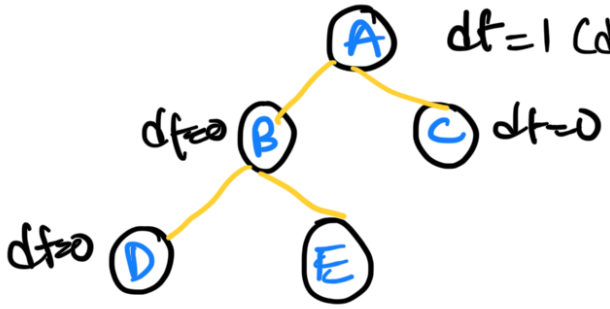


4) Degenerate tree:

either to left or Right (Single node).



5) Balanced tree:



6) Skewed Tree:

Towards left or towards right