C++

Unit I:- Introduction

1. Structured vs OO development
2. elements of OO programming
3. Abstraction - Importance
4. Polymorphism - " "
5. Encapsulation - " " 6. Static & Dynamic Binding

Unit -II: Classes and Objects:

1. Classes and Objects
2. Accessing members of class
3. member func & Inline func
4. Accessing mem. func within class
5. Data hiding and member accessibility
6. Empty classes
7. Constructors & overloading
8. "new", "delete", "this"
9. friend class and friend functions

Unit-III: Overloading:

1. function overloading
2. Operator overloading
   - 'operator' keyword
   - unary operator overloading
3. Generic Programming with templates
   - function templates
   - class templates.

Unit-IV Inheritance:

1. Base-class - derived class relationship
2. Forms of Inheritance
3. "Constructors" & "destructors" in derived class
4. Constructor Invocation
5. Data Conversion (from one obj to other)

** 6. Virtual base classes, virtual functions

Unit V Exceptional handling & files:

1. Exception model
2. E. handling constructs
3. List of exceptions, catching & handling
4. Opening & closing file
5. file modes
6. file ptrs & manipulation
7. seq. access & RAM access
8. Error handling during file manipulations

Structured	OOP
1. focus on algo	1 emphasis on data
2. No control over data flow & visibility	2 Control over data flow & visibility (data hidden)
3. program is divided into ~~object~~ function	3. pgm is divided into Objects & functions
4. ~~Static~~ top-down Design	4 Bottom up Design
5 No dynamic binding	5. Dynamic binding
6.	6 Message passing
7. difficult to design, maintain & upgrade the s/w	7. Easy to design, maintain & enhance the s/w.

- classes are user defined datatype

2. Elements of Object Oriented Programming:

1. Class : logical entity, user defined data type and collection of objects, of same attributes & behavior
2. Object : physical entity, and has a state & behavior.
3. Data ~~Encapsulation~~

3. Data Abstractions:-

Creating new data types using encapsulated items, that are well suited to an application to be programmed, is known as "Data Abstraction" and data types are "Abstract Data types"

(or)

Hiding internal details and showing only required functionality is known as "Abstraction"

4. Data Encapsulation :-

Binding code and data together into a single unit is known as "Encapsulation".

In this data cannot be accessed directly & allowed to access only through functions.

Thus, it enables important concept of Data hiding possible

5. Inheritance:-

*. It is the process, by which one object can acquire the properties of another.

* allows declaration & implementation of one class to be based on an existing class

→ Reusability

→ along with dynamic binding, polymorphism enhances the s/w.

6. Polymorphism:-

→ same operation may behave differently, in different situations. (many forms)

Ex:- if you call "move()" function with Human object it performs differently when it is called with other animal object.

7. Dynamic Binding:-

C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time.

this is called Dynamic Binding (or) Run-time Binding (or) Late Binding;

Ex:-

```
class Animal {
virtual move() {
    }
};
class Human : public Animal {
  move() {
  }
};
class Dog : public Animal {
    move() {
    }
};
```

```
int main() {
```

classes :-

Consider following example :

1.

```
class Test {
    char a;
    int b;
    char c;
};
int main() {
  Test t;
  cout << sizeof(t);
}
```

output:

12

2.

```
class Test {
    char a;
    char b;
    int c;
};
int main() {
    Test t;
  cout << sizeof(t);
}
```

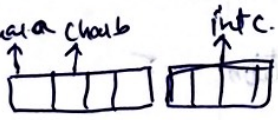
output:

8

* Here the placing of variables in class also defines the size of the object.

* In case 1 :

In case 2 :



* Size of empty class objects will be "1"

Operator Overloading:

- Static polymorphism:

Designed to apply the following principle area.

1. Extending capability of operators on user defined data
2. Data Conversion

Overloadable Operators:

Operator Category	Operators
Arithmetic	+, -, *, /, %
Bit-wise	&, |, ~, ^
Logical	&&, ||, !
Relational	>, <, ==, !=, <=, >=
Assignment (or) Initialisation	=
Arithmetic assignment	+=, -=, *=, /=, %=, &=, |=, ^=
Shift	<<, >>, >>=, <<=
Unary	++, --
subscripting	[]
function call	()
Dereferencing	->
Unary sign prefix	+, -
Allocate and Free	new, delete

Syntax for operator overloading:

Returntype operator OperatorSymbol ([arg1, arg2])
{

}

Complex operator + (complex c2)

{
// one object 'c1' implicit & other explicit 'c2'
}

c3 = c1 + c2;
↓
function call

for istream & ostream

only using friend function & find functions cannot be invoked using objects. so it expects two arguments

& istream & operator >> (istream &in, complex &c1)
{

}

for prefix & postfix:

void operator ++ (int x) → postfix
{
}

void operator ++() → prefix
{
}

6 Overloading New and delete Operators

Data Conversion:

1. Conversion b/w Basic Data Types

weight = (float) age // typecast operator

2. Conversion b/w Objects and Basic Types:

a. Conversion from basic to User defined

define a constructor and assign actual values by converting to object values.

```
Constructor (Basic type)
{
    // steps for converting.
    // Basic type to object attributes
}
```

b) Conversion from UserDefined to Basic Type

```
operator BasicType ()
{
    // steps for converting
    // object attributes to basic type
}
```

- It doesn't specify any return value, but it returns the data type into which we have to convert.

Ex:-

```
operator float ()
{
    float lengthcms;
    lengthcms = length * 100.0;
    return (lengthcms);
}
void main ()
{
    length = M1;
}
```

Ex: object from Basic:

```
Metre (float lcms)
{
    length = lcms * 100;
}
void main () {
    M1 = lengthcms;
        (or)
    M1 (lengthcms);
}
```

3. Conversion b/w objects of two Different classes:

```
Class A object a;
Class B object b;

Object a = Object b;
```

Conversion Routine in Source Object : Operator function

```
class classA
{

};
class classB
{
    private:
    public:
        operator classA()
        {
        }
};
```

Conversion Routine in Destination Object:

Constructor function

```
class classA {

private:
    public:
        classA (classB obj)
        { converting 'B' to 'A';
        } };
```

```
class classB
{
};
```

Subscript Operator Overloading:-

All operators cannot be Overloaded:

Category	operator
member access	. (dot operator)
Scope Resolution	::
conditional	?:
Pointer to member	*
Size of Datatype	sizeof();

Exception Handling

→ **Synchronous Exceptions**

exceptions which occur during the pgm execution due to some fault in the i/p data (or) technique that is not suitable to handle the current class of data.

→ **Asynchronous:-**

Caused by events or faults unrelated (external) to pgm & beyond the ctrl of the pgm. Ex:- Keyboard, disk failure

Inheritance:

The technique of building new classes from the existing classes is called "Inheritance"

```
class A {
}
class B : public A   (or) private (or) protected
{
}
```

↳ if private 'B' cannot access members of 'A'

: if protected 'A's private members become private in 'B'.

if public 'A's functions or members can be accessed by 'B'.

Member Functions Accessibility:

Function type	Access directly to		
	Private	Protected	Public
class member	✓	✓	✓
derived class member	x	✓	✓
friend	✓	✓	✓
friend class member	✓	✓	✓

Forms of Inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid & Inheritance
6. Multipath Inheritance

- Inheritance & member accessibility:

Public Inheritance:

```
class D : public B
```

Private Inheritance

```
class D : private B
```

→

* "We cannot have constructor with arguments without ~~the~~ ~~default~~ defining the default constructor in C++"

~~It~~ shows error in above case

Constructors in Derived classes:

1. No constructors in Derived & Base class.

2. Constructor only in B.C

3. C only in D.C

4. Const. in both B.C & D.C

5. multiple ↑ Consts in b.C & a single Inherited D.C

for every object of derived class, Base class default constructor will be surely ~~called~~ invoked

6. Const in B & D. classes without default const in both:

→ error

7. Explicit invocation in the absence of default const:

```
class D : public B
{  D(int a) : B(a){
   }
};
```

8. Constructor in a multiple inherited class with default invocation:

```
class D : public B2, public B1
{
}
```

order of constructor call

first B2
↓
B1
↓
D

9. Const in multiple inherited class with explicit invocation:

```
class D: public B1, public B2
{ public:
      D() : B2, B1()
      {
      }
};
```

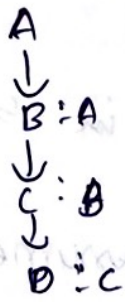
10. Constructor in base & derived classes in multiple inheritance:

```
class D: public B1, virtual B2
{
};
```

B2 ← (virtual first)
↓
B1
↓
D

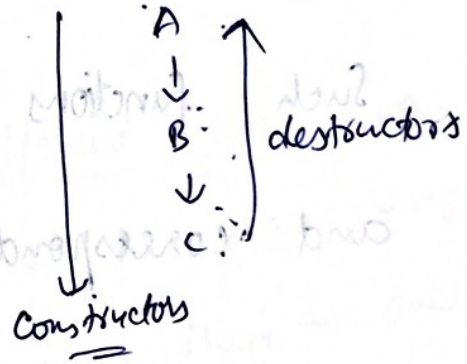
Const in multi-level Inheritance:

~~class A~~



Destructors in Derived classes:

```
class A
{
};
class B : public A
{
};
class C : public B
{
};
```



Resolution of Ambiguity in multiple Inherited classes

:: (Scope Resolution Operator)

Obj. A::fun();

Virtual functions :-

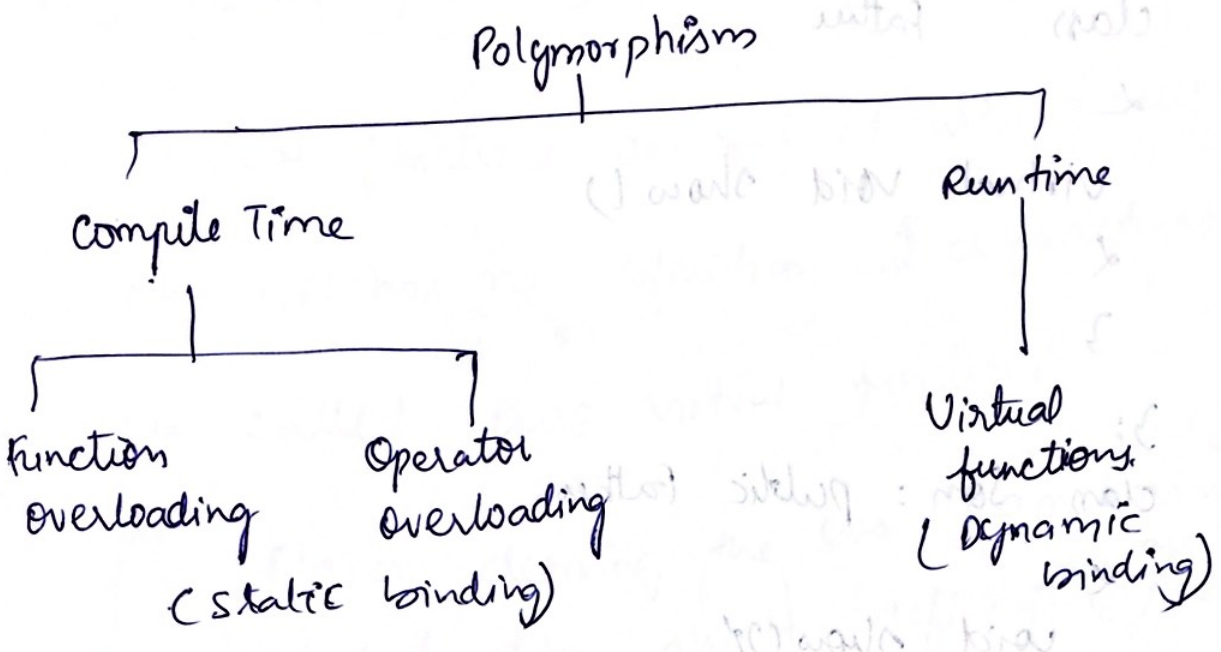
* Polymorphism means in biology, the ability of an organism to assume a variety of forms

* In C++, it indicates the form of a member function that can be changed at runtime

Such functions are called "Virtual Functions" and corresponding class is called "Polymorphic class"

* The objects of the polymorphic class are addressed by pointers, change at runtime and respond differently for the same message.

* Such mechanism requires postponement of binding of function call to the member function until "Run time". So, "Late binding"



Need for virtual functions:

→ when objects of different classes in a class hierarchy, react to the same message in their own unique ways, they are said to exhibit polymorphic behavior.

→ Resolving a function call at a compile time is known as "compile-time" (or) "Early" (or) "static" binding.

→ Resolving a function call at run time is known as "run-time" (or) "late binding" (or) "Dynamic" Binding

1)

```
class Father
{
  virtual void show()
  {
  }
};
class Son : public Father
{
  void show()
  {
  }
};
int main()
{
  Father *fp;
  Father f;
  fp = &f;
  f->show();
  Son s;
  fp = &s;
  fp->show();
}
```

Pure Virtual functions

* Virtual functions are defined with a null body; it has no definition. Such functions are called pure virtual functions

* Classes deriving the base class having pure virtual function should define it again or declare it as virtual.

* classes having pure.v.func cannot be used to define any objects of its own. and those are called "Abstract Classes"

Abstract classes:

→ Abstract classes (classes with atleast one virtual function) can be used as a framework upon which new classes can be built to provide new functionality.

Ex: Debugging process

Virtual Destructors:

Virtual Constructor ✗

B ↓ D ↑ destructor. Same as virtual function

→ Dynamic Binding:

Rules for Virtual func:

1. Virt. func def is compulsory in base class but pure virt. func is not necessary
2. cannot be static members
3. can be a frnd func to another class
4. accessed using obj pointers
5. base ptr → derived
6. prototype identical in both base & derived
7. cannot have virt const
8. benifit → public

Templates
(Generic Programming)

generic function for Swapping

```
#include <iostream.h>
template <class T>
void swap (T &x, T &y)
{
    T t;
    t = x;
    x = y;
    y = t;
}
```

Finding Maximum

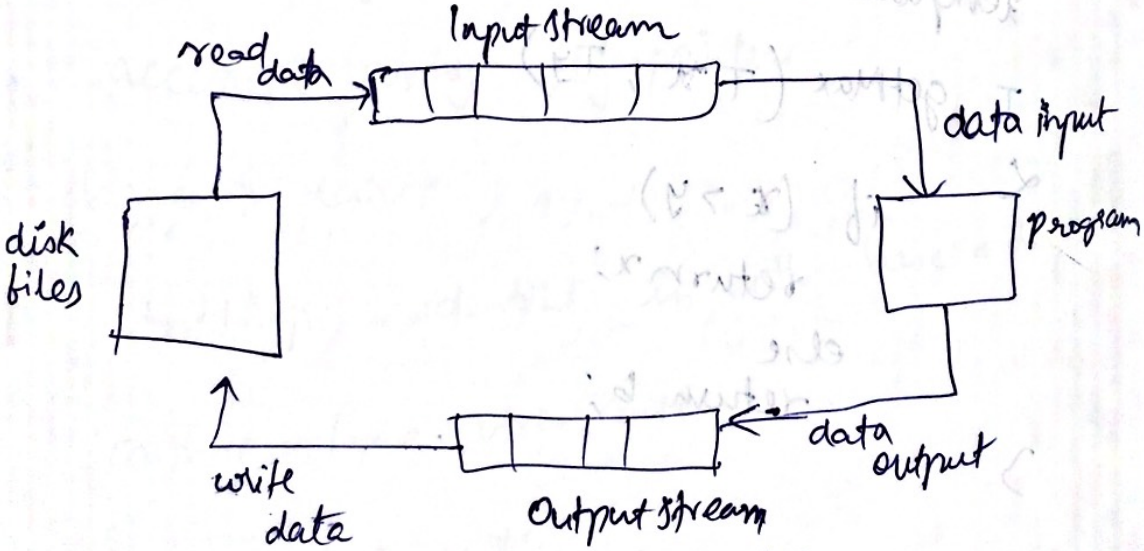
```
template <class T>
T getMax (T x, T y)
{
    if (x > y)
        return x;
    else
        return b;
}
```

class Templates:-

```
template < class T1, class T2, ..... >
class CN
{
    T1 data;
    void func1( T1 a, T2 &s);
    T2 func2 ( T2 *x, T2 *y );
};
```

Stream Computation with files

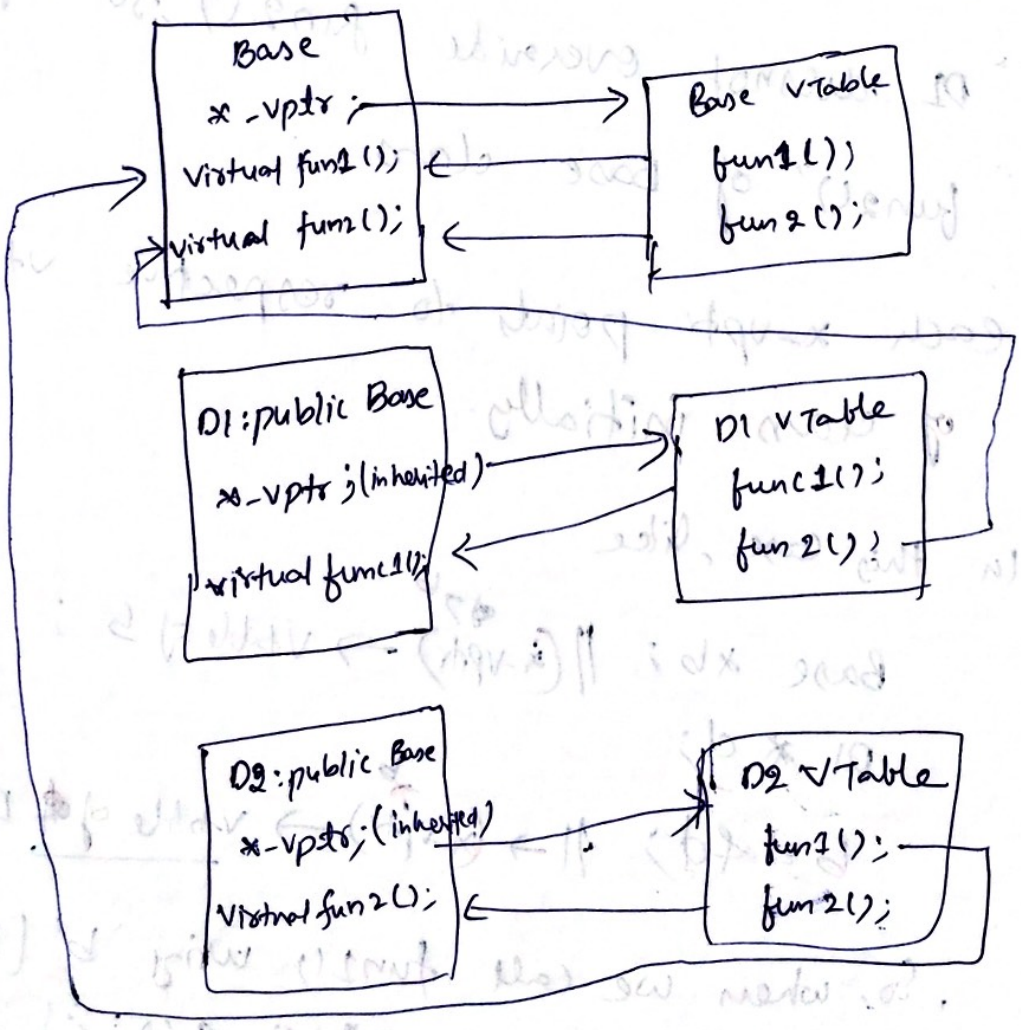
File manipulation services such as create, open, write, read, rewind, close and delete



- ofstream - for handling o/p files
- ifstream - for handling i/p files
- fstream - for handling both i/p & o/p

```
void main()
{
    ofstream fout("student.out");
    cin >> name;
    fout << name;
}
void main()
{
    ifstream fin("  ");
    fin >> name;
}
```

Virtual Tables:- (VTables):



* Virtual table & *-vptr (hidden ptr) are set-up at compile time by compiler for each class

* 'vtable' consists of total no. of 'V. functions' in the environment placed like static array, each element is a virtual func

* On each virtual function in vtable of each class maps to the most derived function of the class like.

D1 doesnot override 'fun2()' so it points 'fun2()' of Base class

* each *-vptr points to respective vtable of class initially.

* In this case, like

Base *b; // (*-vptr) → vtable of b;

D1 *d;

b = &d; // → (*-vptr) → vtable of D;

, So, when we call fun2() using 'b' (b->fun2())

it calls "D::fun1();"

Operating System

An O.S is a program that acts as an intermediary/interface between a user of a computer and the computer hardware

OS Functions

1. concurrency
2. virtual memory
3. multiprocessor support
4. File system
5. Security & Protection

features

→ Abstractions:
hide details of H/w
configuration

→ Arbitration:
manages access to shared resources

Services provided by OS

1. Pgm creation
2. Pgm execution
3. access to i/o files
4. System access

Goals:

Primary - convinience
secondary - efficiency

types:

- Batch OS
- multiprogramming
  - multiprocessing → many cores
  - multitasking
  - Real time → dead line

Process Management

Attributes of a Process :-

PCB → Context

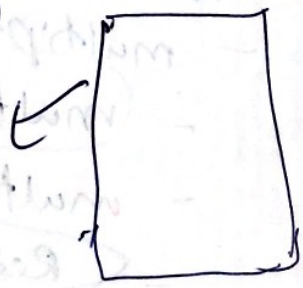
1. Process id
2. Program counter
3. Process state (Ready, Running, waiting)
4. Priority
5. General purpose Registers → (Register values which are present during pre-emptions are stored in GPR of PCB)
6. List of open files
7. List of open devices
8. Protection

PCB
Context

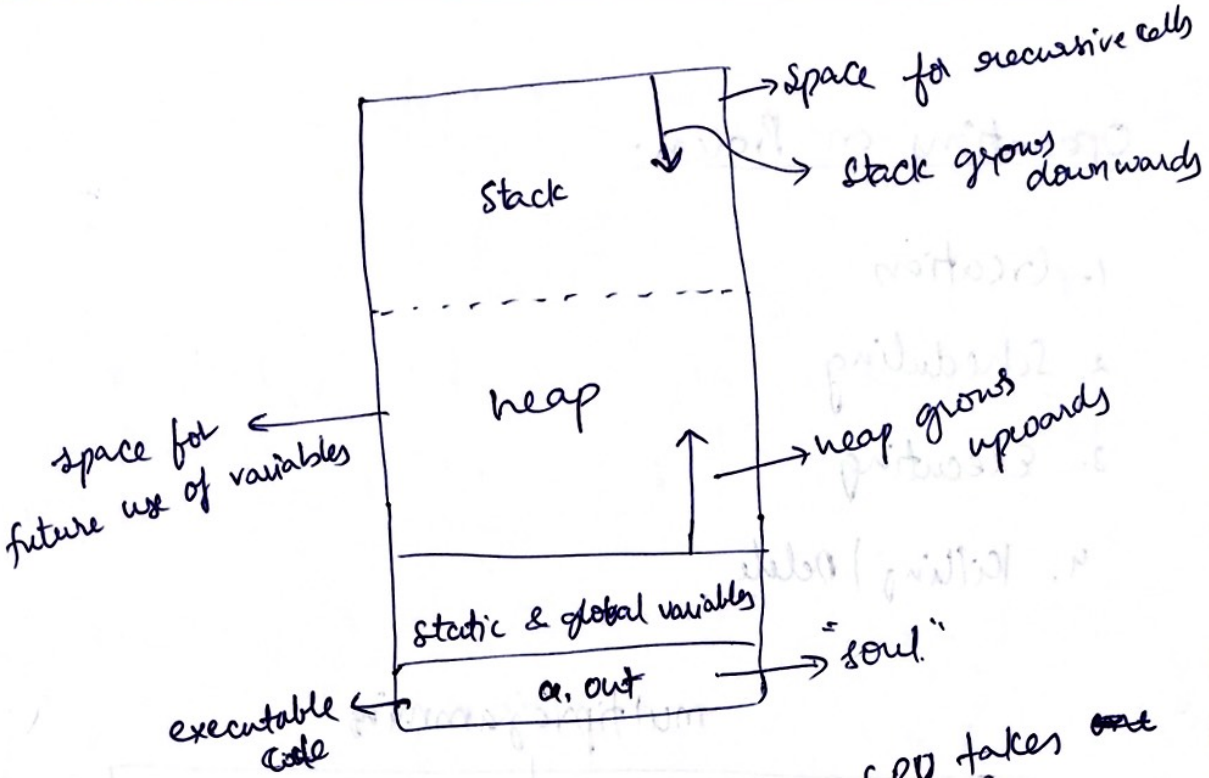
Process when we should start execution it creates a following structure

Ex.c
⇓
compiler
↓
a.out

* All the attributes of a process are maintained in a PCB (Program control block)



PCB



→ when a process is executed, the CPU takes line by line and starts execution.

→ The C.P.U should access only the "process boundaries otherwise it generates "segmentation" fault

States of a Process

1. New (secondary m)
2. Ready (main m)
3. Run (m m)
4. Block (or) wait (m m)
5. Termination (or) Completion (PCB & every trace about process will be deleted)
6. Suspend Ready (m m (run) → s m)
6. Suspend wait (or) Suspend block (from waiting (m m) to second m)

Operations on Process:

1. Creation
2. Scheduling
3. Executing
4. Killing / Delete

multiprogramming

- with pre-emption (multi-tasking) — Time sharing
- without pre-emption

→ Long Term Scheduler (new → Ready)
↗ no. of process that can be created

→ Short term Scheduler / Dispatcher (Ready → Run)

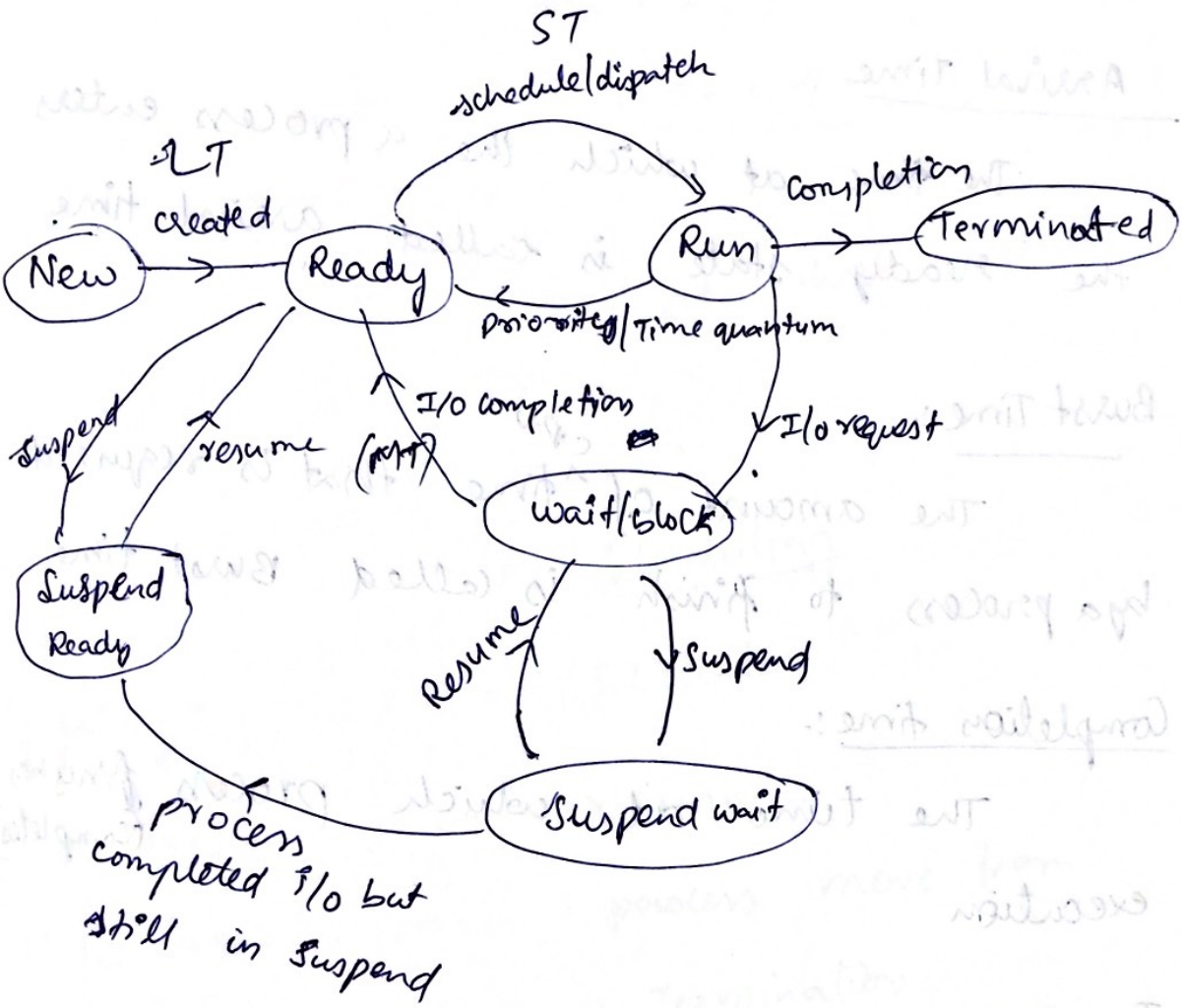
→ medium term Scheduler (wait → Ready)

Degree of Multi-programming:

⇒ The maximum no of process that can be in Ready state is called (D.O.M.P)

→ Long Term Scheduler

Process State Transition Diagram



→ CPU Bound
more CPU required

→ I/O Bound process
more I/O required

Q consider a systems with 'N' CPU processors and 'm' processes then

	min	max
ready	0	M
running	0	N
block	0	M

Important Parameters of Processes :-

1. **Arrival Time**

The time at which a process enters the ready state is called arrival time.

2. **Burst Time :-**

The amount of CPU time that is required by a process to finish is called Burst time.

3 **Completion time :-**

The time at which process finishes (completes) execution

4. **Turn Around Time :-**

The difference b/w completion time & arrival time is - turnaround time.

~~CT - AT = B.T~~

(Completion time) - (Arrival time) = B.T + W.T

TAT.

$\Rightarrow$ **TAT = B.T + W.T**

5. Waiting Time :-

$$W.T = TAT - B.T$$

$$W.T = (C.T - A.T - B.T)$$

6. Response Time :-

The time at which a process first hits CPU

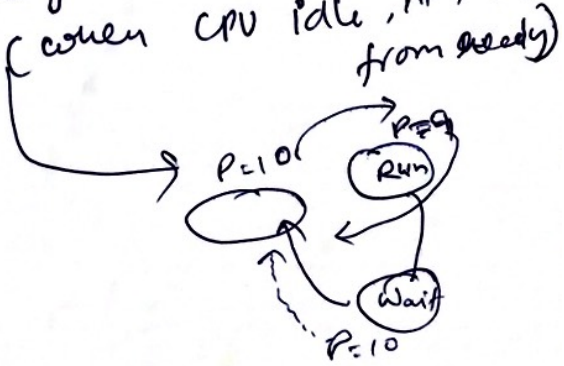
CPU Scheduling

who → short term scheduler

where → ready → Running

when → when a process move from

1. Run → Termination
2. Run → wait
3. Run → Ready
2. New → Ready i.e when a process is just created (high priority)
3. wait → Ready (when CPU idle, it takes from ready)



FCFS (first come first serve)

Criteria : Arrival time

mode : non-pre-emptive

Implementation of C++

1. Company - In & outs
   - clients
   - financial matters (share holdings)

class StudentInfo, name, age, department:

```
class StudentInfo {
    char name[20];
    int age;
    char dept[10]; int deptId;
public:
    StudentInfo() {
        name = "";
        age = 0.0;
        dept = "";
    }
    StudentInfo(name, age, dept) {
        this.name = name;
        this.age = age;
        this.dept = dept;
    }
    void Input() {
        cout<<"
        cin>> name;
        cin>> age;
        cin>> dept;
    }
```

```
void display ()
{
    cout << " Name : " << name ;
    cout << " Age : " << age ;
    cout << " dept : " << dept;
}
};

class EmployeInfo {
 protected:
  char name [20];
  char degn [20];
  char dept [10];
  char doj [10];
   int empId;
   int sal ;
public:
  void input() {
      cin >> name;
      cin >> degn;
      cin >> dept;
      cin >> doj;
      cin >> empid;
      cin >> sal;
  }
  void display() {
      cout << name << "\n" << degn << "\n" << dept
      << "\n" << doj << "\n" << empid << "\n" << sal;
  }
};
```

class TeachingStaff : public EmployeeInfo

{
private:
int no_of_classes;
char free_hours [20];

SRS (UML)
|
SDS
|
HLDD (High level design)
|
LLDD

class diagram
↑
(Architect)

};

class NonTeachingStaff : public EmployeeInfo

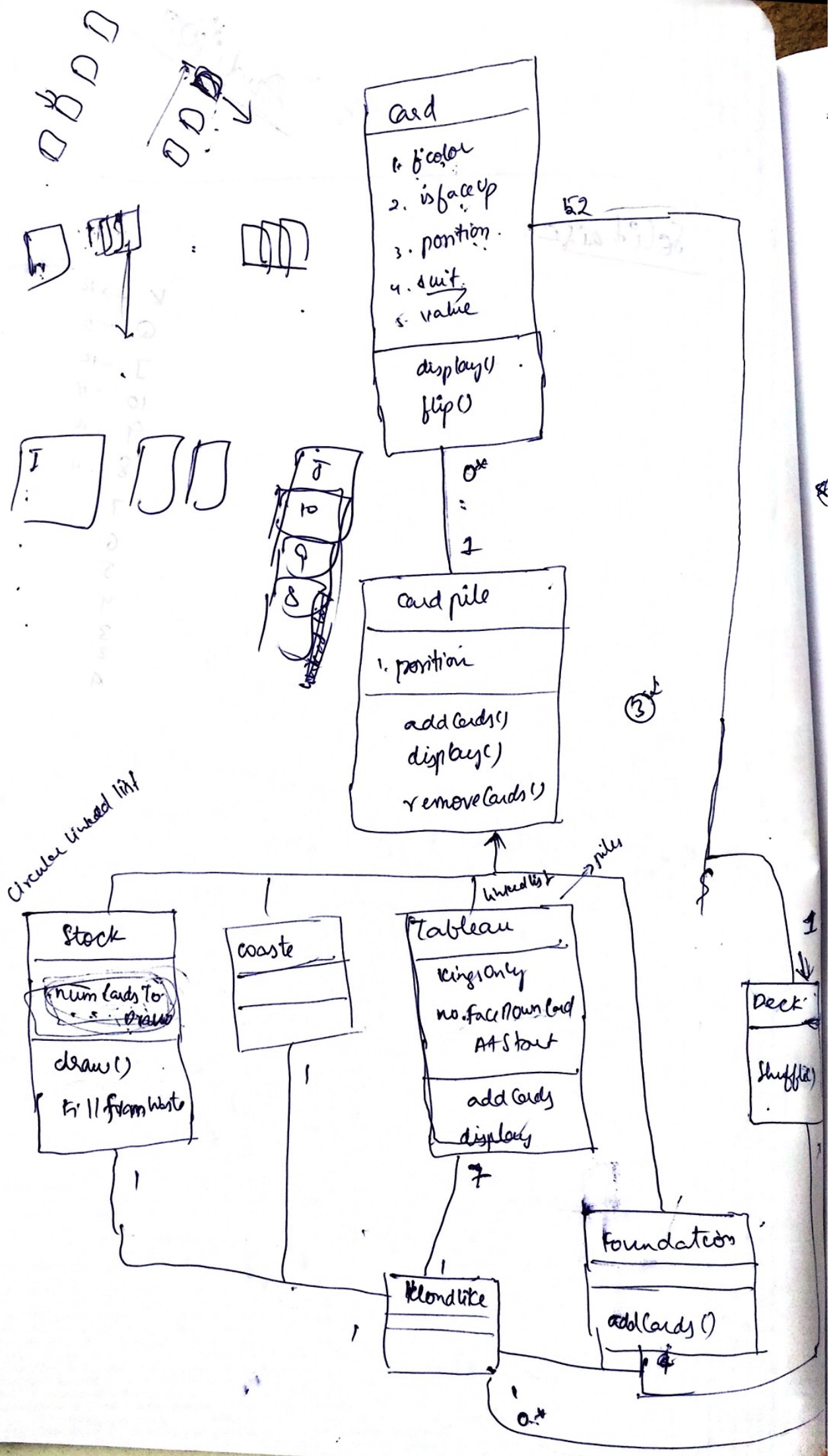
{ private:
int

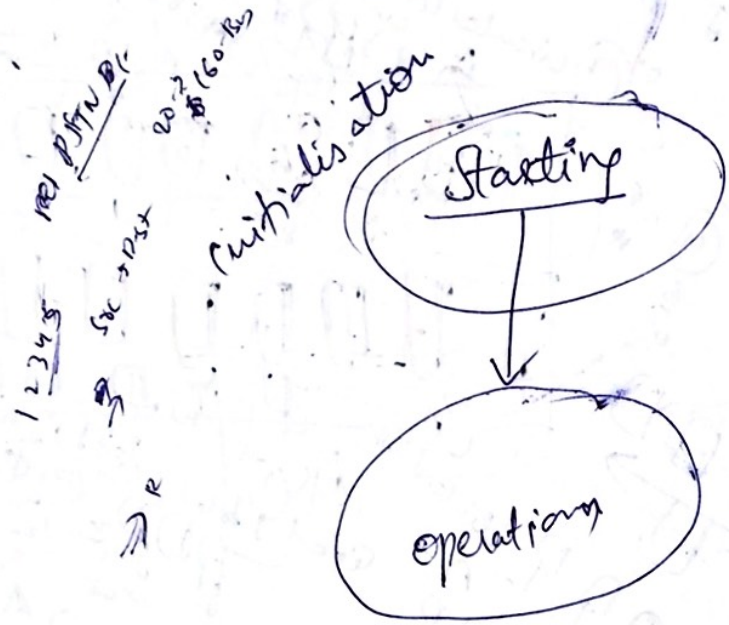
};

public Department {

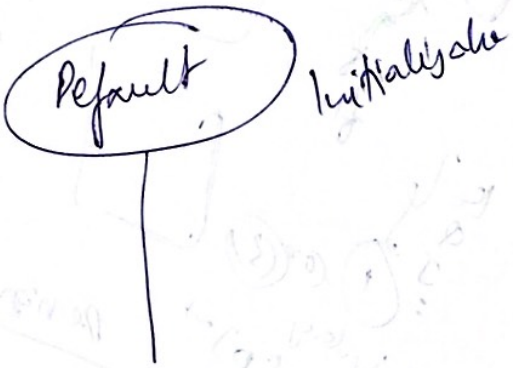
private:
char dept[20];
→ int deptid;
int no_of_t_staff;
int no_of_stud;
int no_of_non_t_staff;

};


Card
1. fColor
2. isFaceUp
3. position
4. suit
5. value
display()
flip()
52
0..*
1
CardPile
1. position
addCards()
display()
removeCards()
3 out
Circular Linked list
LinkedList
piles
Stock
numCardsTo Draw
draw()
fill from Waste
Waste
Tableau
King Only
no. faceDownCard At Start
addCards
display
Deck
Shuffle()
Foundations
addCards()
Klondike
0..*



q - mem



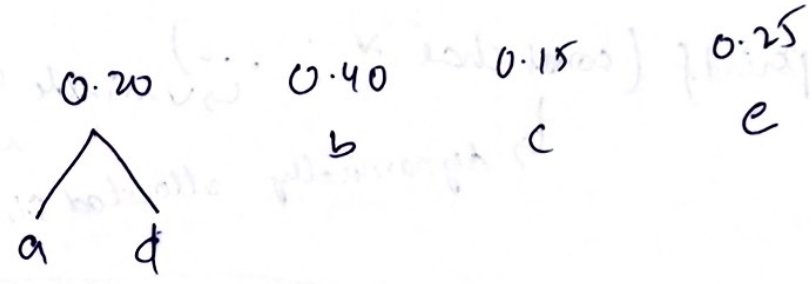
Advanced D.S

1. extended Binary Tree:
2. Quill tree
3. left
4. Huffman Coding

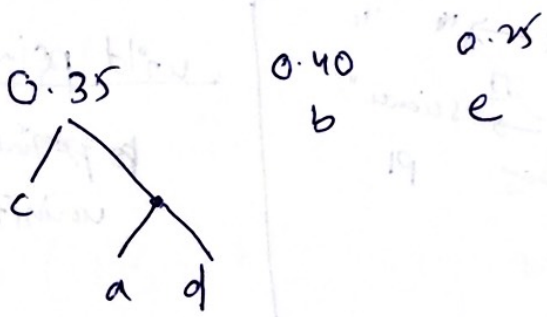
0.12	0.40	0.15	0.08	0.25
a	b	c	d	e

↳ Take minimum value elements & merge them

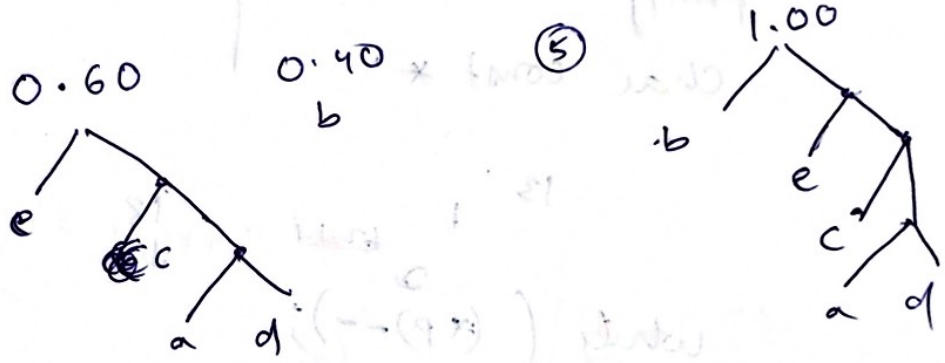
2.



3



4.

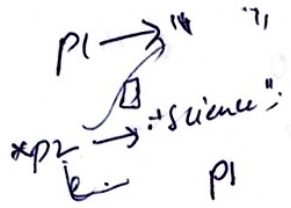


Query operations

1. Dynamic transitive closure
   (directed graph G) Reachable (x, y)

16/7/15

printf (const char *, ...)
↳ variable list (for) variable no of args
↳ dynamically allocated string



wild pointer:

~~pointer~~ pc
uninitialised pointer

~~printf~~

char const *

13
1
~~bool~~
2
while ((*p)--);

13
↓

conditional statement error (compile time)

→ heap storage - static & ~~global~~ global variables.

→ void main() {

char *str = "good";

char *str2;

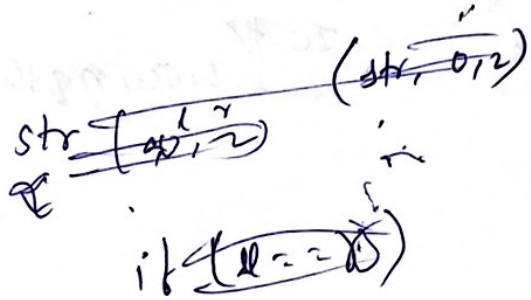
strcpy (str2, str);

printf ("%.s", str2);

}

error : (for str2 no memory allocated)

hello ,



permute (str, 0, 2)

abc, acb, bac, bca, cba, cab

```
{
for (i = l ; i <= r ; i++)

for (i=0 ; i< size ; i++) {
    for (j=i+1; j < size ; ) {
      if (arr[j]==arr[i]){
    for (k=j ; k<size ; k++) {
       arr[k] = arr[k+1];
    } size --;
    } else j++;
} }
```

Removes consecutive duplicates

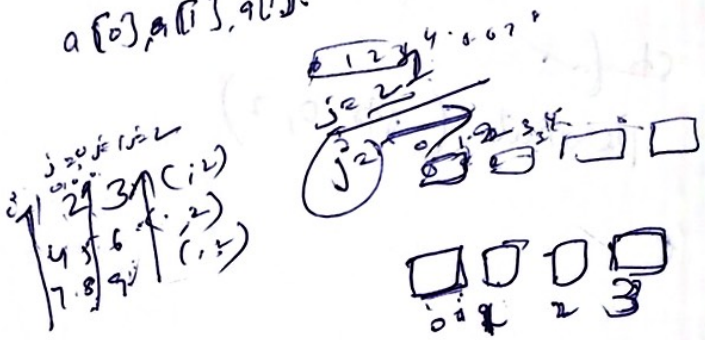
Deshaw

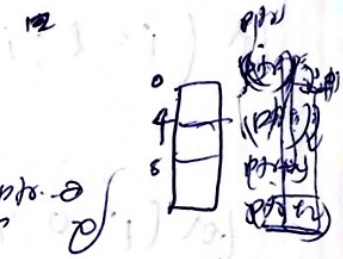
20 Aptitude

20 technical qstns

MFC

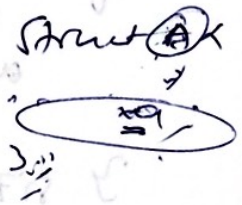
a[0], a[1], a[2].





max(1,2)

2, 2, 2



16/7/15

Pointers

int a; → variable

int *b; → ptr

int **c; → ptr to a ptr

→ value in pointer is constant

`const char *a;`

↓ Here address can be changed and point to a diff address

```
char r = 't';
char *a = &r;
*a = 't2'; → error
```

`char * const a;`

↳ Here address cannot be changed but value pointing to can be changed

```
char r = 't';
char *a = &r;
char r2 = 'x';
a = &r2; → error
*a = 'r2'; → ✓
```

→ Eg: Infinite loop, for whatever may be the input:

```
char ch='A';
while(ch)
{
    ch=getch();
}
```

a = 20

&a =

→ int T;

main() {

struct T {

char c; }

printf("%d", sizeof(T));

}

	C	C++
sizeof(T)	2	1
sizeof(struct T)	2	2
sizeof('a')	2	1

Output

C

2

(It does not consider 'T' as an independent data type & takes value of int T;

C++ (It considers 'T' as an independent data type)

1

Insert into table (col1, col2, ... col17) select

(, , , ,) from ~~table~~ table where

col = " ";

→

```
CreateObject("SAPI.SpVoice")
Speak "Ram in VIT"
".vbs"
```

17/7/15

```
struct employe
{
    int empid;
    char name[20];
    float salary;
}e1;
```

```
union employee
{
    int empid;
    char name[20];
    float salary;
}e1;
```

e1.13 ⇒ first member will be assigned with value

Memory allocations

malloc();
calloc();
realloc();
free();

Storage classes in 'c'

Register
Static
Auto
extern

2) Choose correct one

i) short const register i=10;

ii) static volatile const int i=10;

iii) unsigned auto long register i=10;

iv) signed extern float i=10.0;

3) sizeof(3.14) — sizeof(3.14f), sizeof(3.14l);
↓ 8 ↓ 4 ↓ 10

6) main()
{
int i=-1, j=-1, k=0, l=2, m;

m = i++ && j++ && k++ || l++;

printf cout << i << j << k << l << m;

}

i=0 ; k=1
j=0 ; l=3 ; m=1

7

0000 0000 0000 0001 − 1

1111 1111 1111 1110

+1

1111 1111 1111 1111 −1

1111 1111 1111 0000 −1 << 4

F F F 0

8)

```
if(a)
{
}
else
    break; → 'break' cannot be placed in if else
}
```

9

a = 2

a = a+2 << 1

ASCII Value

a = '97'

10

'\n' → ASCII value 10

11

13

15

1111

1110 → 14

p = 1

q = 8

i = 2

i = 1

*p = 2

*q = 8

(13)
(B)

returns length of string

printf(" ");

dersana.jayasankar@faceacademy.in

Time Complexity

1) Asymptotic Notations

only one consider the no of loops running an

(

Database

DDL - autosave

DML - commit to save

TCL - Rollback
- commit
- Savepoint

DCL - grant, Revoke

DQL:

~~select first_name~~

select upper (first_name)

22/7/15

Database Systems

Data - Fact recorded

Ex: Text, images, numbers, videos, speech

Database - collection of related data

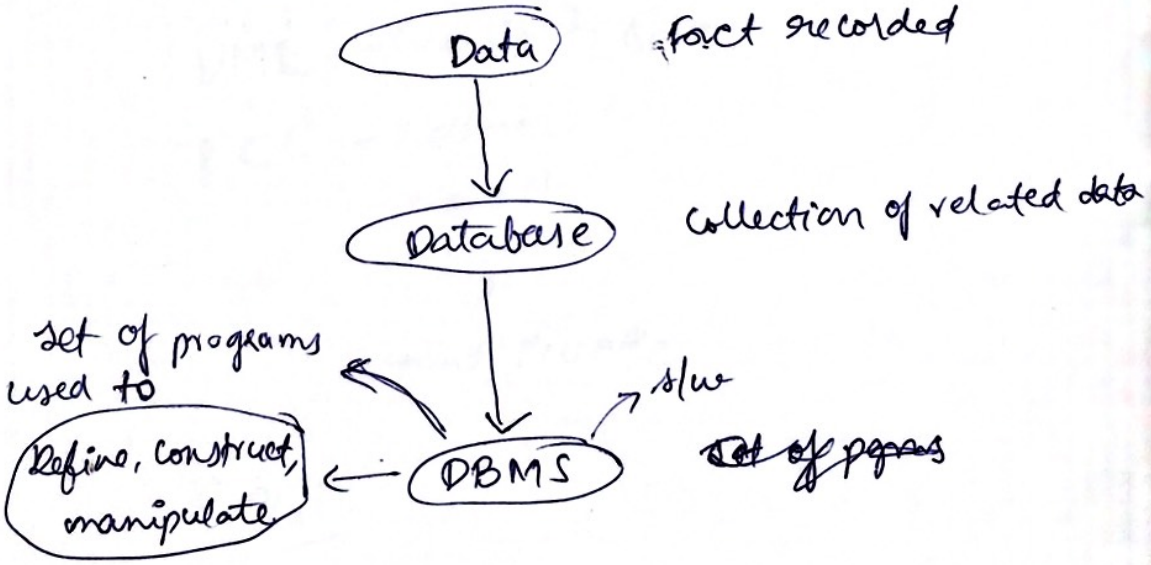
Text & number → Traditional database

videos & audios → Multimedia DB

geographical Img, videos → Geographical Info System

stocks, alert when out of stock → Realtime DB

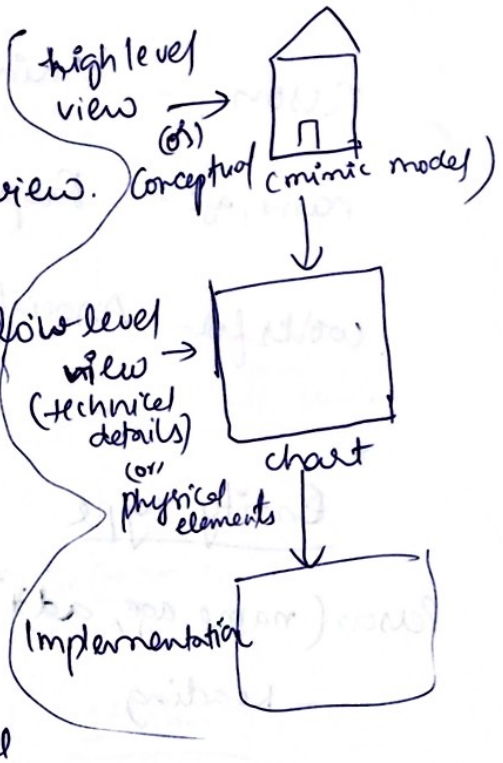
historical DB → Data warehouse



Database + DBMS → Database Systems

Ex: Book, Pen

* Naive Users cannot understand low level view (or) technical view. So high level design is made.



DB Design

1) High Level (or) Conceptual
   - E-R modelling
2) Representational (or) Modelling (or) Relational Model:
   Relations (tables)
3) Low level (or) Physical Data elements:
   → Implementation
   → Accessing

E-R Model

- Entity (Nouns) → Any object in db
- Attributes → (verbs describing entities)
  ↳ properties of entities.
- Relationship (Verb) → Describe the relationship b/w entities

Person - Thing - Entity - Noun

name, age - Properties Attributes - verb

works for - Association Relationship - verb

Entity Type - Entity

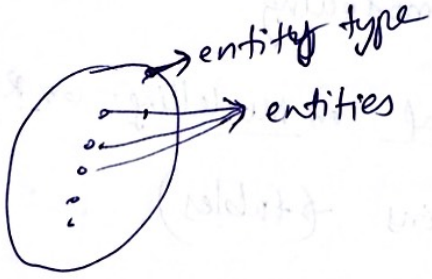
Person (name, age, add) —— (Raju, 26, Hyd)

- Reading - Instance of entity type

Intension - Extension

-

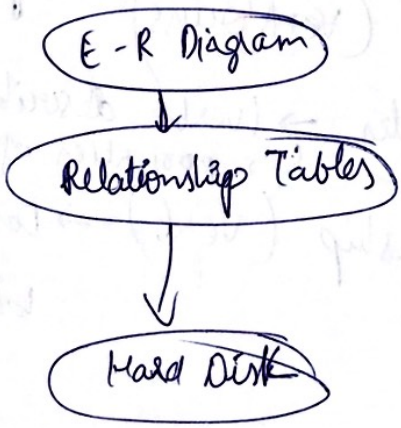
-



→ At any instance the no of elements (or) all elements in an entity type is called "State of Entity" (or) "State of Database"

11'ly for

Relationship Type - Relations



Attributes

1. Composite vs Simple Attributes

Simple → First Name

Composite → Name → FN (first), MN (middle), LN

2. Single valued vs Multi valued attributes

Single valued → age (only one)

Multi valued → Ph.no, Add → more than one ph.no's

3. Stored vs Derived attributes

Stored → dob

Derived → date of birth ⇒ age

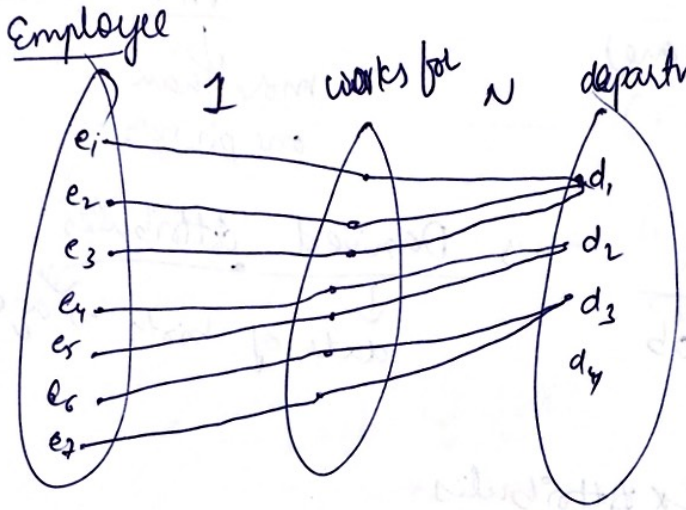
4. Complex Attributes

↓

composite + Multi-Valued

Relationships

Req: Every emp works for exactly a department and a dept can have many emps: New dept need not have any emp.



Degree: No. of entities participating in a relationship

In above ex: degree: 2 (emp, dept)

Cardinality Ratio:

max. no of relations, an entity can participate

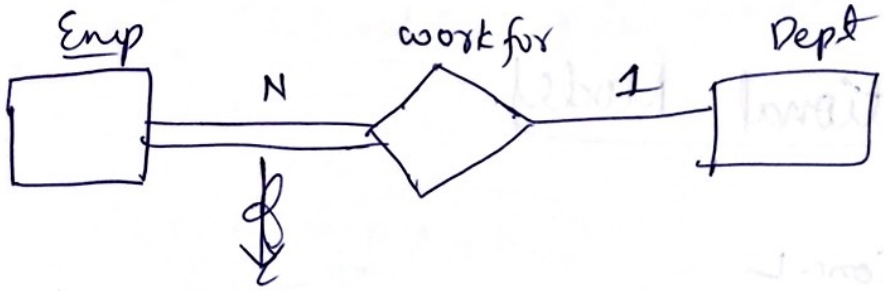
Ex:- employee : 1
dept : N

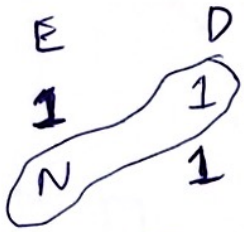
Structural constraints

Participation (or) Existence:

min no. of relationships an entity can participate

Ex:- emp: 1
dept: 0





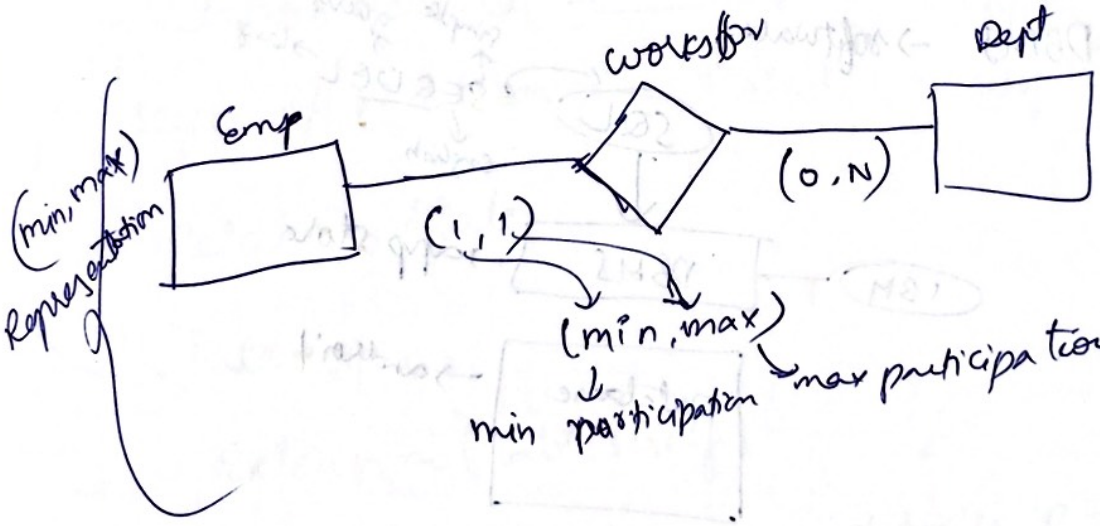
N : 1

——— → Partial participation

⬭ → total participation

↓

⇒ all entities are participating in relation then, it is total participation



Relational Model

Conc. L

Rep. L

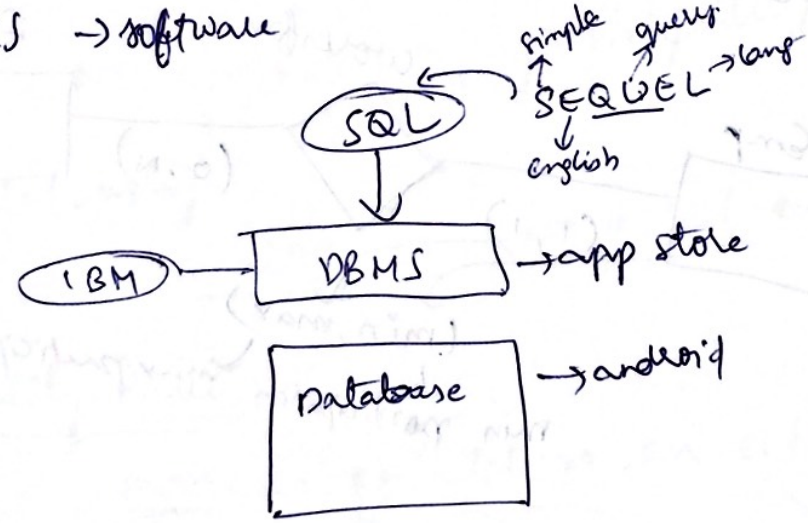
Low Level

DBMS

Database (model)

Relational model — Database Model

DBMS → software



hierarchical, Network databases

↳ legacies

Relational calculus + Relational DB + Relational Algebra

⇒ RDB + RA + RS

DBMS — SQL

Terminology

Relation - Table

tuple - row

attribute - column

domain - {set of values}

Relation Schema → heading of table ⇒ $R(A_1, A_2, A_3, A_4)$

D_1, D_2, D_3, D_4 → set of values

degree of Relation

Relation state

Intension

Extension (Table itself)

~~Current Relation state~~ ⇒ no. of tuples present in table at that pt of time

$R(A_1, A_2, A_3, A_4)$

D_1, D_2, D_3, D_4

table — table name

$r(R) \subseteq D1 \times D2 \times D3 \times D4$

(ordered pair containing one value from all of the domains)

In mathematics, any subset of a cross product is Relation,

$r(R) \subseteq$ (table 'R') is a subset, so $r(R)$ (table) is called Relation.

→ maximum no. of tuple,

$$r(R) \subseteq |D_1| \times |D_2| \times |D_3| \times |D_4|)$$
$$10 \quad 10 \quad 10 \quad 10$$

10^5 (max)

but all possibilities may not be meaningful like, a student might not have all possible ph. no's

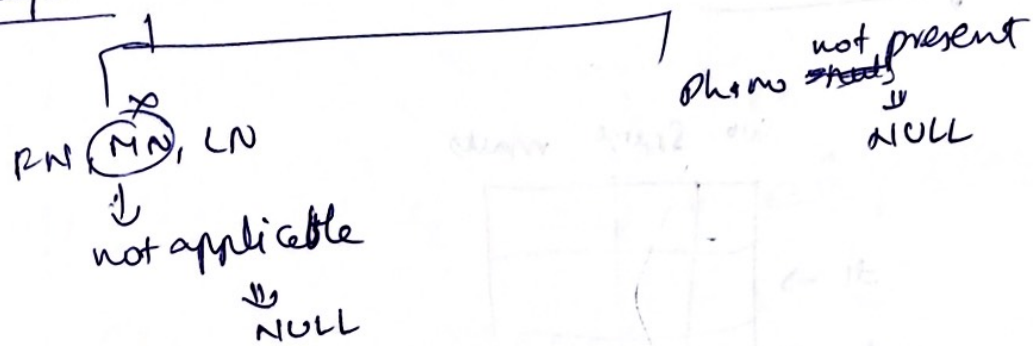
Tuple, Tuple Values & Null

Relation == Set

* A set cannot have two same values,

⇒ A relation cannot have all the values in tuple same.
↳ two different

Exceptions



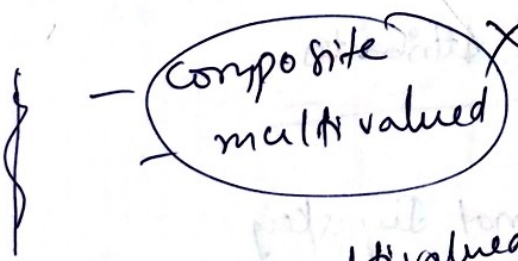
Relational Constraints :-

1. Domain Constraints → 2 values range 3
2. Key Constraints → uniqueness
3. Entity Integrity constraints
4. Referential Integrity Constraints.

1) Domain

Atomic
↳ smallest individual unit

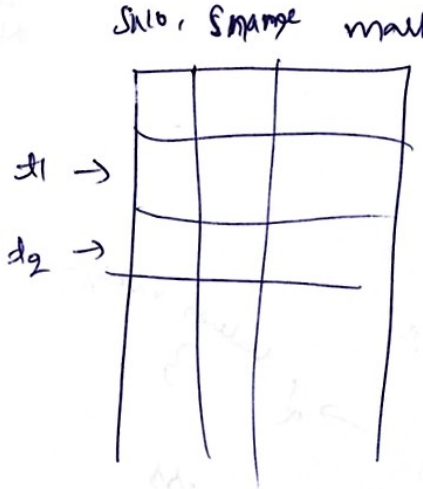
- flat file system



⇒ multivalued ⇒ new relation

composite ⇒ break

2) Key Constraints

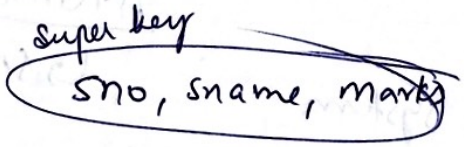


No two tuples in a relation will have all the values in tuples same.

$$t_1 \neq t_2$$

Super key :-

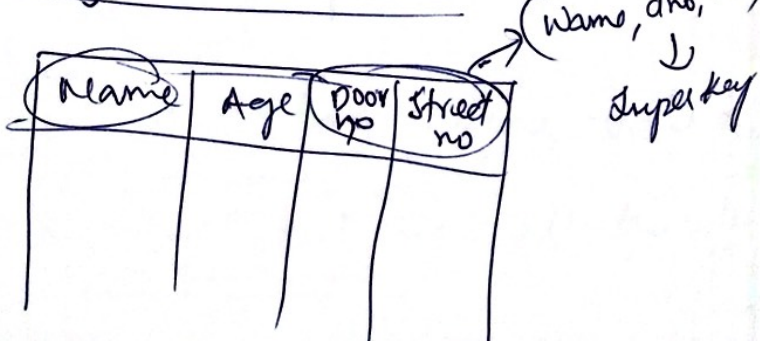
the set of attributes in which no two tuples will have all the attributes same



Super key $\subseteq$ Attributes

Key = minimal Superkey

Ex. for key having two attributes



default Key ⇒ (Set of all attributes)
⇓
(Super key)

Any Superset of a key is a Super key

(SNO, Sname) → Super key

→ Key
R(A1, A2, A3, A4)

for A1 ; no of superkeys

A1, A2, A3, A4
2 × 2 × 2

∴ 2^3 ⇒ 8 S.K's

→ More than one keys in a table :-

Ex :- ①

mobile

owner	sim no	MEI-no

② Ex

minimal superkeys (or) key (or) Candidate key

owner name	color	Reg. no	engine. no

Candidate keys { # keys of table }

Primary key { one of candidate key }

A1, A2, A3, A4 → Super key (uniquely identifies)
*

A2, A3, A4 - S.K ⇒ (If I remove A3 if we don't it cannot identify uniquely then it is key)
*

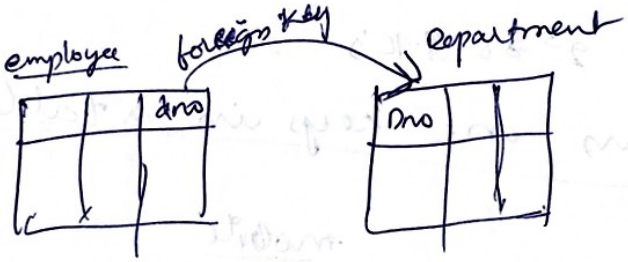
A3, A4 - S.K

Either candidate key 1 (or) Candidate key 2 can be primary key

Entity Integrity
Referential Integrity
} Constraints

According to entity integrity
→ P. key cannot be null

Referential. I.C
↳ foreign key cannot be null



~~P.K + null~~
F.K can be null
↳ can be duplicate
↳ more than one is allowed

Semantic
↳ Rules
Ex:- emp sal < 1,00,000

Actions upon constraint violations

i) insert
ii) Delete
iii) Update

	insert	delete
Domain .c	✓	✗
Key .c	✓	✗
Entity .c	✓	✗
Referential .c	✓	✓ (f.K – R.K)
action	Rejection	

P.K null ←

Three Actions

dept

dno	dname
1	
2	
3	

d-no

d-no
1
2
3

3-actions

1) Rejection

Cascade delete
(delete all tuples in table referring to particular id)

Ex:- emp-dependents of emp-id

emp-id
1
1
1

Set NULL

Set NULL for the particular value

→ dept-id of employees

Modify : delete + insert

Normalisation

1) Consider,

emp

eid	e_n	did

dept

did	d_no	S_id

all tables in a universal table

eid	e_n	d_id	d_n	superVisor_id

↓

Problems in universal table:

1) Redundancy

2) Anomalies

- Insertion

in universal table, at one place,

did	d_name
1	CSE

another place

did	d_name
1	ECE

while inserting we may go wrong, and creates erroneous data, this causes insertion anomalies

Deletion Anomalies :

Ex:- If you want to delete all emp of dept_no '1'. you should delete emp_details in U.table & replace null values @11 delete all tuples, where you cannot have dept info of department itself.

3) Update (Modify) :-

→ if you want to change dept_name entire U.table you have to change

Procedure of Dividing Tables according to rules is called "Normalisation"

→ we use "Functional Dependencies" and "Candidate keys" for Normalisation.

Functional Dependencies

If key is given in a table, then we can extract the remaining attributes in the respective tuple

table T

A	B	C
1		
2	a	b
3		
2	a	b

Ex:-

~~given~~

'A' determines (B, C) (or)

$A \rightarrow (B, C)$ (B, C) functionally dependent on A

(2) (a, b)

Def:-

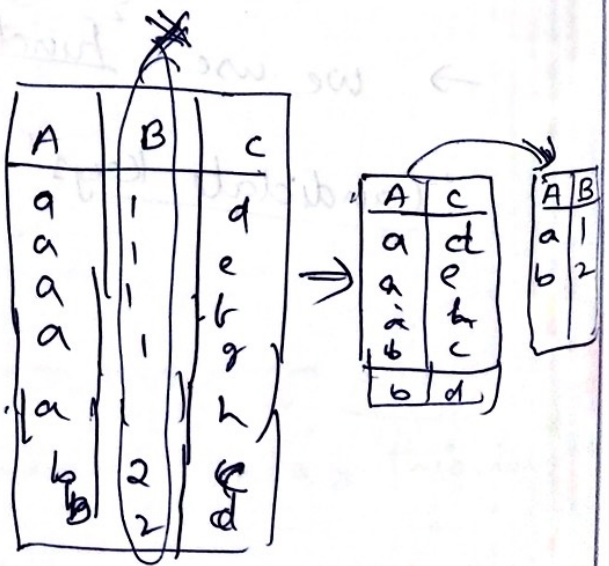
tuple t_1 tuple t_2

$t_1(A) == t_2(A)$

then $t_1(B, C) == t_2(B, C)$

* If there are lots of repetitions in tables & few attributes which are functionally dependent, then replace the attributes with a new table

Ex:-



By this, we eliminate the redundancy in attributes (table).

$A \rightarrow B$

('A' determines 'B')

('B' functionally dependent on 'A')

(BCNF)

(A B) A, B = collection of attributes

$X \rightarrow Y$ (functional dependency)

3 types

Trivial

$A \rightarrow A$

$AB \rightarrow A$

$X \supseteq Y$

Non-Trivial

$X \cap Y = \phi$

$A \rightarrow B$

$A \rightarrow BC$

$AB \rightarrow CD$

Semi-non Trivial

$X \cap Y \neq \phi$

$AB \rightarrow BC$

Rule out the functional dependencies based on the tables:

(may be) $Eid \rightarrow ename$

$ename \rightarrow eid$

Eid	Name
1	a
2	b
3	b

A	B	C
1	1	4
1	2	4
2	1	3
2	2	3
2	4	3

✗ $A \rightarrow B$

✗ $B \rightarrow C$

✗ $B \rightarrow A$

✗ $C \rightarrow B$

✓ (may be) $C \rightarrow A$

✓ (may be) $A \rightarrow C$

Formal Definition :-

	A	B
t_1		
t_2		
	if t_1 & t_2 agree here	then they must agree here
	if t_1 & t_2 disagree here	they may agree or disagree here

Various Usages of FD's → (during Requirements analysis)

1) Identifying additional (extra) FD's

2) Identifying Keys

3 Identifying equivalences of FD's → $A \rightarrow BC$ or $A \rightarrow B$, $A \rightarrow C$

4 Finding minimal FD set

Operations of ~~FD's~~ DB using FD's

Two methods

(i) Inference Rules ✗ error prone

ii) Closure set of attributes ✓ easy

Inference Rules:

a) Reflexive : $A \to B$ if $B \subseteq A$

b) Transitive : $A \to B$ and $B \to C$ then $A \to C$

c) Decomposition : $A \to BC$ then $A \to B$ & $A \to C$

d) Augmentation : if $A \to B$ then $AC \to BC$

e) Union : if $A \to B$ & $A \to C$ then $A \to BC$

f) Composition Rule : if $A \to B$ & $C \to D$ then $AC \to BD$

Closure Set of Attributes

Func. Dependencies :-

$A \to B$

$B \to D$

$C \to DE$

$CD \to AB$

closure set :-

It contains all the attributes which can be derived from the given attribute.

Ex :-

FD's :- $A \rightarrow B$

$B \rightarrow D$

$C \rightarrow DE$

$CD \rightarrow AB$

$A^+ = \{B, D, A\}$; $B^+ = \{B, D\}$

$C^+ = \{C, D, E, A, B\}$; $D^+ = \{D\}$

$E^+ = \{E\}$; $(CD)^+ = \{A, B, C, D, E\}$

Determining Candidate Keys

if $R(AB)$ then, no of candidate keys possible

$= \{A, B, AB\}$

$R(ABC)$

then, $\{A, B, C, AB, BC, AC, ABC\}$

$\Rightarrow (2^n - 1)$ candidate keys

Ex:- R(A B C D)

$\Rightarrow 2^4 - 1 = 15$ cand. keys may be possible

$A \to B$
$B \to C$
$C \to D$
$D \to A$

If 'A' is candidate key then key containing 'A' is 'super key'

Cand key of length '4' — (A B C D)

len '3' — (A B C) (B C D) (C D A) (A B D)

len '2' — (A B) (B C) (C D) (D A) (A C) (B D) ⇒ Super keys

len '1' — (A) (B) (C) (D). } candidate key

(A) $A^+ = \{A, B, C, D\}$

$B^+ = \{B C A D\}$

$C^+ = \{A B C D\}$

$D^+ = \{A, B, C, D\}$

Equivalence of FD's

$F : \{A \to C, AC \to D, E \to AD, E \to H\}$

$G : \{A \to CD, E \to AH\}$

check i) F covering G ⇒ F ⊇ G

and

ii) G ⊇ F

i) $F \supseteq G$

$A^+ = \{C, D, A\}$ ⇒ $F \supseteq G$

$E^+ = \{A, D, H\}$

ii) G ≥ F

$A^+ = ACD$

$E^+ = \{A, H, C, D\}$

⇒ G ≥ F true

⇒ both are equivalent

Lossless Decomposition

→ Normalisation mostly implies dividing a large table into small one's

→ If we again combine these small tables, we should not get any additional data.

→ we have to take care while splitting the table.

A	B	C
a_1	b_1	c_1
a_2	b_1	c_1
a_1	b_2	c_2

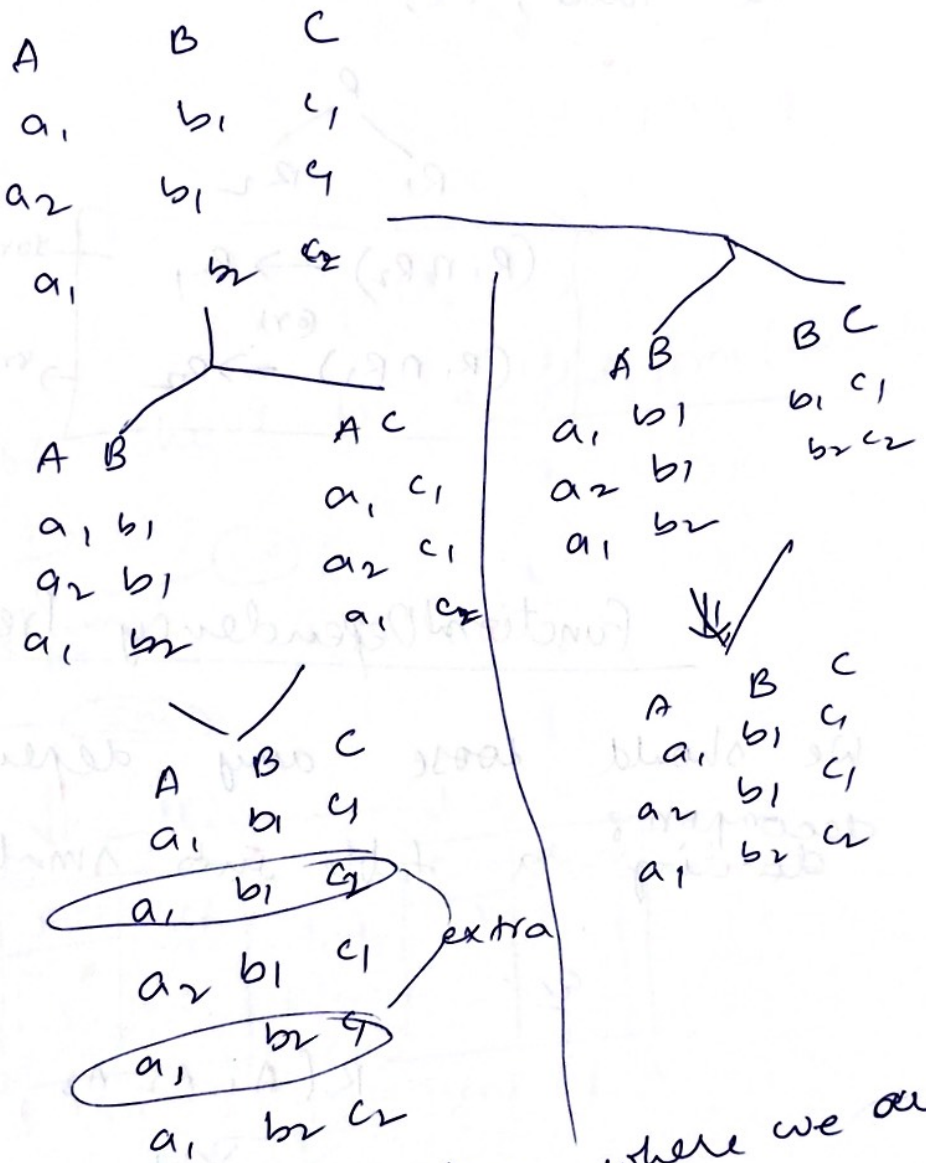
A
a_1
a_2

B	C
b_1	c_1
b_2	c_2

→ Ex:-

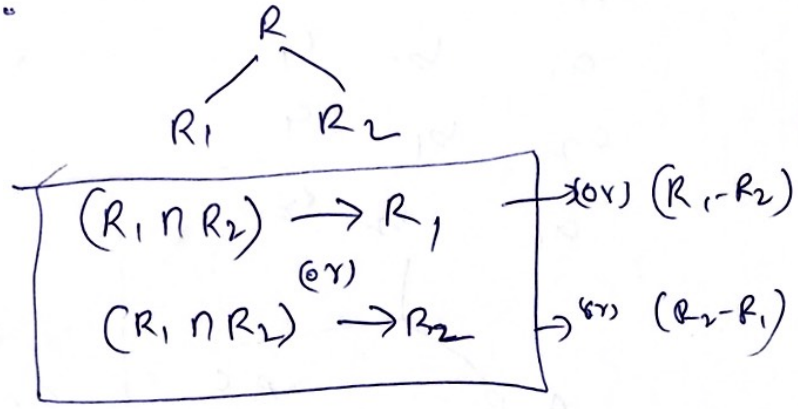
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_1	c_1
a_2	b_2	c_2 → extra

→ when we are dividing, we should take care that some attributes of tables should be common.



→ This type of decomposition, where we are gaining additional data which is not real is "Lossy Decomposition"

→ The Decomposition will not be lossy if the common attribute is a key in any one of the table, i.e,



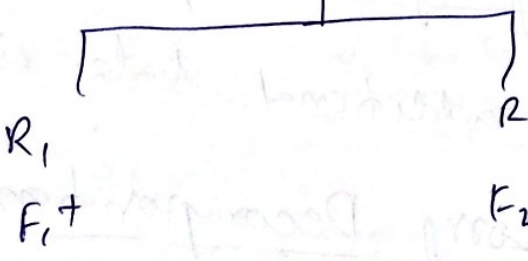
Functional Dependency Preserving

We should loose any dependencies after ~~dividing~~ decomposing a table into smaller tables

Ex:-

$R(A_1\ A_2\ A_3 \cdots A_n)$

$F \rightarrow F^+$ (total func. D's)



$$(F_1^+ \cup F_2^+)^+ = F^+$$

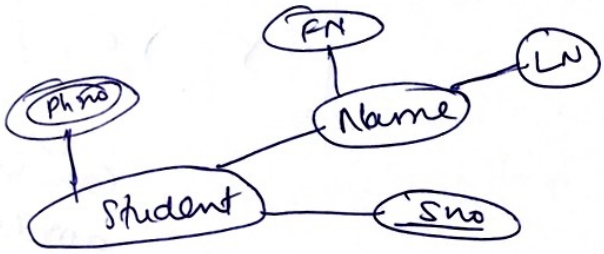
First Normal Form:

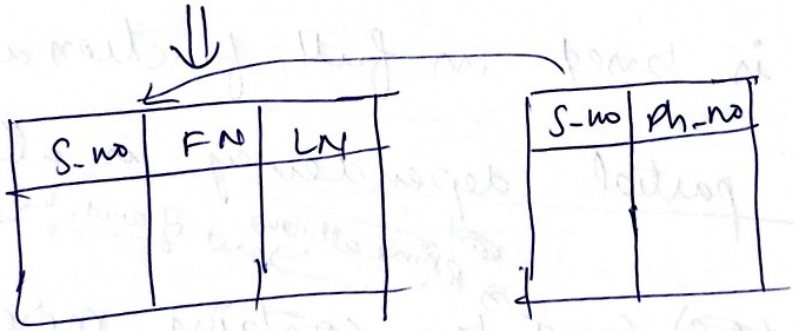
Aim: BCNF (0% Redundancy)

1NF
2NF
3NF
BCNF

1NF:-

The table should be in flat form (heirarchy)





→ By default

→ (No multivalued, No composite) Attributes

Second Normal Form:-

FD's

$AB \rightarrow C$

$B \rightarrow C$

$\rightarrow A^+ = \{A\}$

$B^+ = B, C$

$C^+ = C$

$(AB)^+ = \{A, B, C\}$ ← cand. key $\Rightarrow$ (AB) unique

$(BC)^+ = \{B, C\}$

$\{CA\}^+ = \{C, A\}$

A	B	C
1	a	C_1
2	a	C_4
1	b	C_2
2	b	C_2
1	c	C_3
2	c	C_3
3	c	C_3
4	c	C_3
5	c	C_3

Can be avoided

→ 2 NF is based on full functional dependency

No partial dependency is allowed

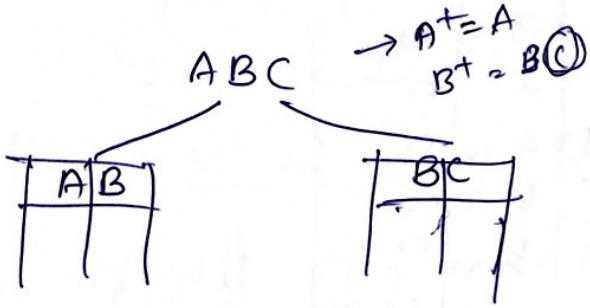
Prime attribute → part of cand. key

→ if (AB) cand. key contains more than one attribute, then part of it (or) no single attribute in cand. key should not determine anything

Ex:- (AB) → C (AB → cand. key)

B → C → not allowed

→ So, Remove part of attribute which is determining and place it separately



$A^+ = A$
$B^+ = B$
$AB^+ = AB$

$B^+ = BC$
$C^+ = C$

3NF :-

No transitive Dependencies

Transitive dependency :-

Non-prime attributes transitively depending on the key

Prime Attribute :-

Part of candidate-key

$A \to B,\ B \to C \Rightarrow A \to C$

R(ABC)
$A \to B$
$B \to C$

$A^+ = ABC$
$B^+ = BC$
$C^+ = C$

Transitive dependency

LHS = Superkey
RHS = Key

A	B	C
1	a	c_1
2	a	c_1
3	a	c_1
4	a	c_1
5	b	c_2
6	b	c_2
7	b	c_2

A	B
1	a
2	a
3	a
4	a
5	b
6	b
7	b

$A \rightarrow B$

B	C
a	c_1
b	c_2

$B \rightarrow C$

common attribute

(Loss less decomposition)

3 NF may preserve f. dependencies

BCNF (Boyce Codd)

A Relational Schema 'R' is in BCNF, if whenever a non-trivial FD $X \rightarrow A$, holds in 'R', then 'X' is super key of R, i.e, determinants of all FD's must be super keys

X	Y
a	c
a	c
a	c
b	d
b	d
b	d

wherever you see, $X \rightarrow Y$, form a new table where.. X is key, Y is attribute

R (A, B, C)

FD (A → B, B → C, C → A) } BCNF

CD: $A^+ = ABC$

$B^+ = ABC$

$C^+ = ABC$

Ex:-

R(ABC)

F: { AB → C, C → B }

~~A+ = {A, B, C}~~ $A^+ = \{A\}$

$B^+ = B$

$C^+ = B, C$

$AB^+ = A, B, C$

$Bc^+ = B, C$

$Ac^+ = A, B, C$

A, B, C → prime attributes

2/10/15

finding Candidate Keys in Relation,

R (A B C D E F) with functional dependencies

F.D = { AB → C, C → D, D → EB, E → F, F → A }

Sol:- with '1' Key Attribute

A+ = {A}

B+ = {B}

✓ C+ = { C, D, E, B, F, A }

✓ D+ = { D, E, B, F, A, C }

E+ = { E, F, A }

F+ = { F, A }

with '2' Key Attributes

✓ (AB)+ = { A, B, C, D, E, F }

(AE)+ = { A, E, F }

(AF)+ = { A, E }

✓ (BE)+ = { B, E, F, A, C, D }

✓ (BF)+ = { B, F, A, C, D, E }

(EF)+ = { E, F, A }

with '3' attributes

✓ (BEF)+ =

(AEF)+ =

✓ (ABF)+ =

✓ (ABE)+ =

with '4' attributes

(ABEF)+

↓

*

=> '8' Candidate Keys

2Q Candidate Keys of Sub Relations

R(ABCD) : { AB → CD, D → A }

what are the candidate keys of sub relation

R(BCD)

R(BCD)'

Sol:- B+ = {B}

C+ = {C}

D+ = { D, A }

BC+ = { B, C }

CD+ = { C, D, A }

BD+ = { B, D, A, C } → is candidate key

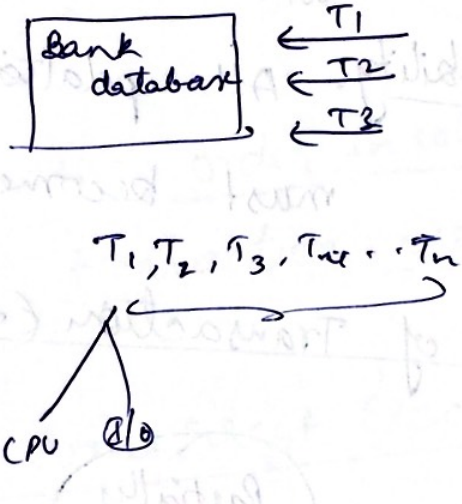
Transaction Management & Concurrency Control

Transaction:-

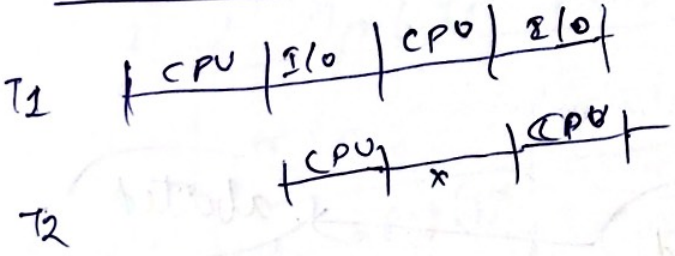
A transaction is a collection of operations that forms a single logical unit of work.

Transaction

R(A) → I/O
A = A - 50 ; → CPU
W(A) → I/O
R(B) → I/O
B = B + 50 ; → CPU
W(B)



Transaction



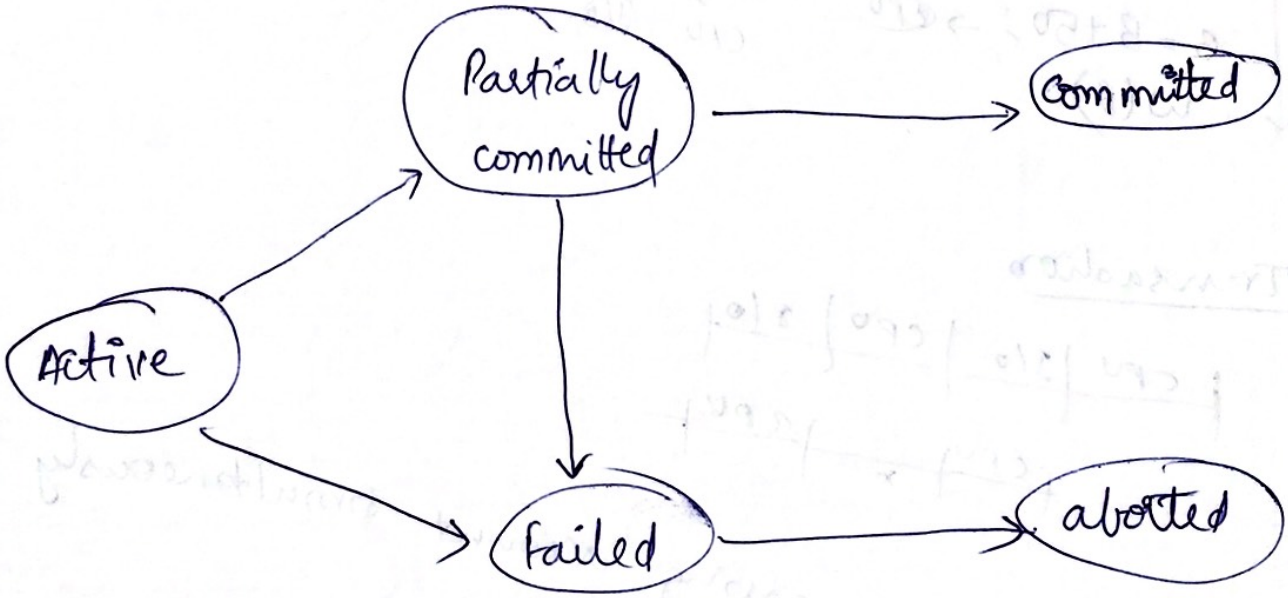
→ Problem of accessing account simultaneously

Similar to Synchronisation techniques in O.S we have Transaction Management techniques in DBMS

Transaction Properties

1. **Atomicity** :- All or none → Transaction Manager
2. **Consistency** :- correctness → User/Application Programmer
3. **Isolation** : Each Tx should be executed without knowing what is happening with other transaction → Concurrency control manager
4. **Durability**: All updations by a Tx must become permanent → Recovery

States of Transaction (or) Transaction States



Concurrent Execution :-

Interleaving execution of the operations of a transaction.

why?

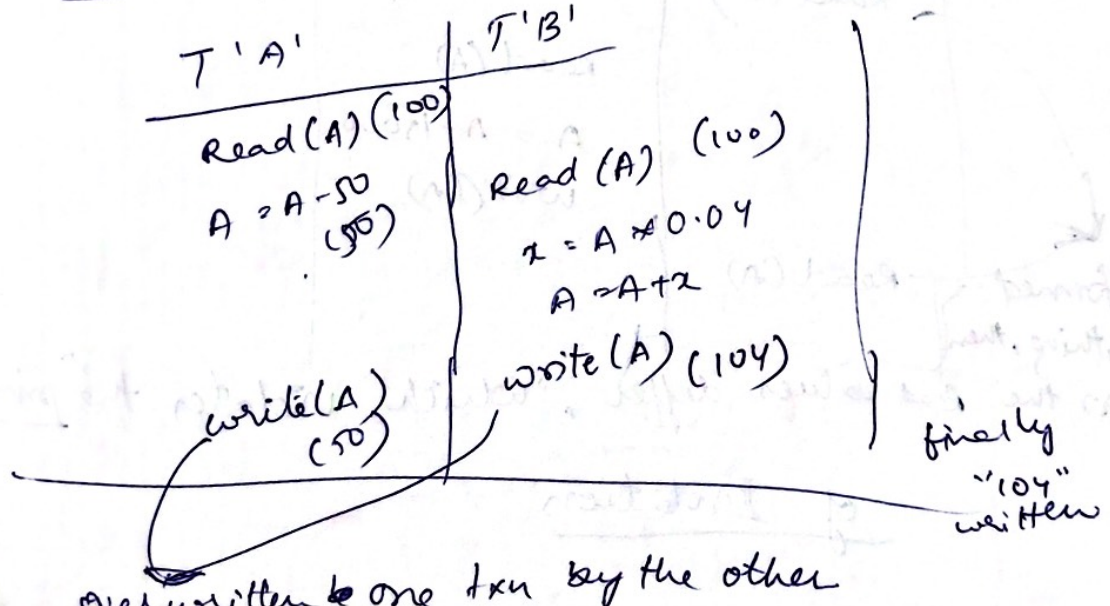
1. for avoiding longer waiting time
2. Simultaneous & effective use of I/O & CPU
2. the processor & disk utilisation increases

Schedule :-

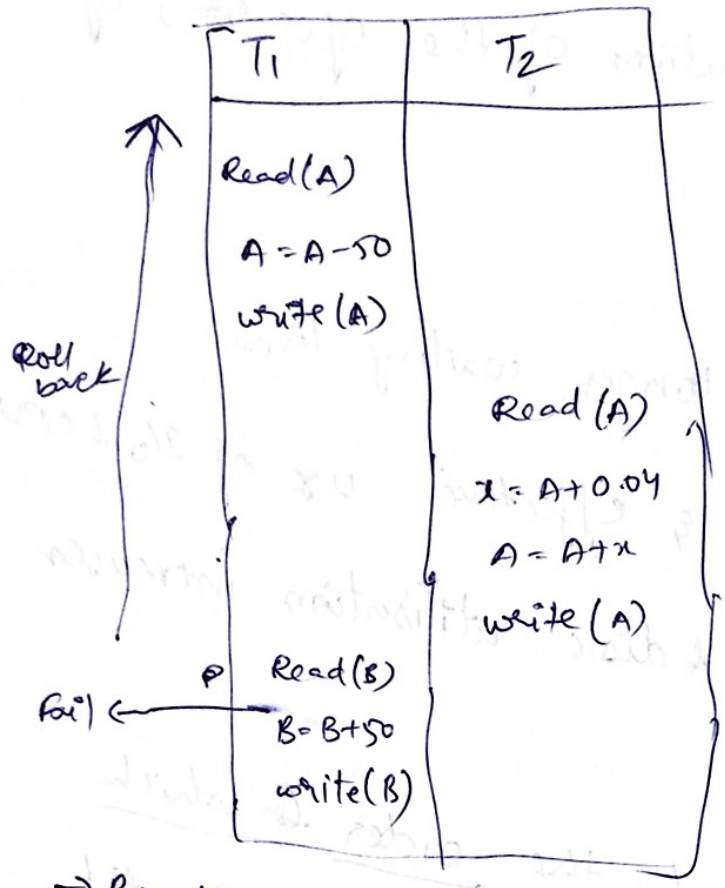
It represents the order in which instructions of a transaction executed

Problems due to concurrent Execution of the transactions

1. Lost Update Problem (W-W conflict)

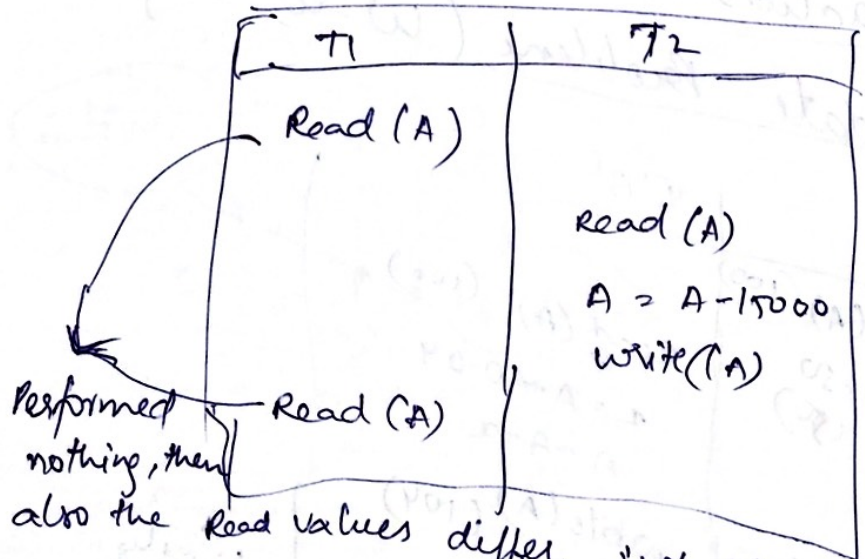


2. Dirty Read Operation (W-R conflict)



→ Reading an un-committed data is called dirty read operation.

3. Unrepeatable Reads (or) Non-Repeatable Reads (R-W)



also the Read values differ, which violates the principle of Isolation.

Phantom Tuple (Phantom Phenomenon):-

→ Tuple is added by transactions

T1	T2
eno ename salary 1 A 5000 3 C 4000 select * from emp where salary >3000	insert into Emp Values(4,D,3500)
eno ename salary 1 A 5000 3 C 4000 4 D 3500 select * from emp where salary >3000	

→ Reading Tuples by one transaction at different times gives different Results

4) Incorrect Summary Problem:-

T1	T2
	sum = 0 Read (x) sum = sum + x ;
Read (x) x = x + 500 ; write (x) ;	
	Read (x) ; sum = sum + x Read (y) sum = sum + y ;
Read (y) y = y + 500 write (y) ;	

we will get wrong summary, if some other transaction is interfering and changing the values in between, when we are summing up the values

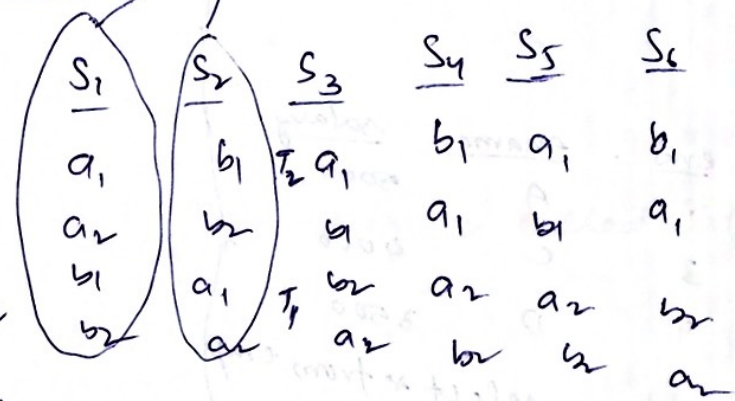
Characterizing Schedules:-

1) Serial Schedule:-

Ex:-

T_1	T_2
a_1	b_1
a_2	b_2

↓ Interleaving should be like after 'a' only 'a2' (after "b" only "b2")



If there are 'm' transactions and 'n' operations in each transaction, then

T_1 , T_2 , T_3 ... T_m

n_1 n_2 n_3 ... n_m ,

the no. of possible schedules are

$$\frac{(n_1+n_2+n_3+\dots n_m)!}{n_1! * n_2! * n_3 \dots n_m!}$$

Serial Schedules are like T_1 T_2 (or) T_2 T_1 $\Rightarrow 2!$

$\Rightarrow \underline{\underline{n!}}$

*) Serial Schedules $\Rightarrow$ Consistent $\Rightarrow$ but must performance

, we have to allow those schedules, which are logically equivalent to Serial Schedules.

Types of Schedules :-

① Serial Schedule:

Serial Schedules, that does not interleave the actions of any operations of diff transaction.

$T_1 \rightarrow T_2$ (or) $T_2 \rightarrow T_1$

T_1	T_2
R(A)	
A=A+100	
W(A)	
	R(A)
	A=A+100
	W(A)

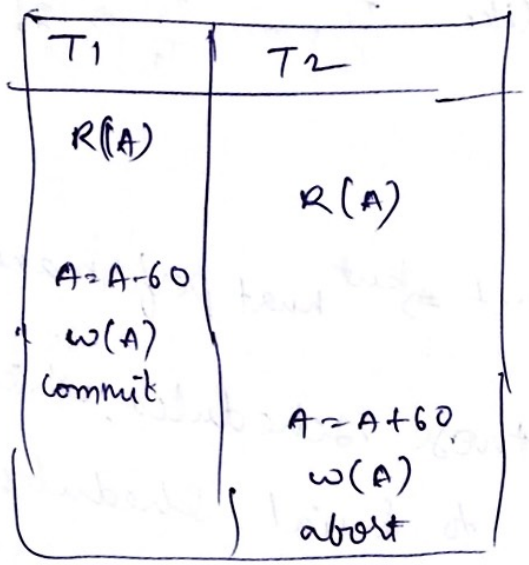
consistent state of operation

T_1	T_2
R(A)	
A=A+100	
	R(A)
	a=a+100
W(A)	
	W(A)

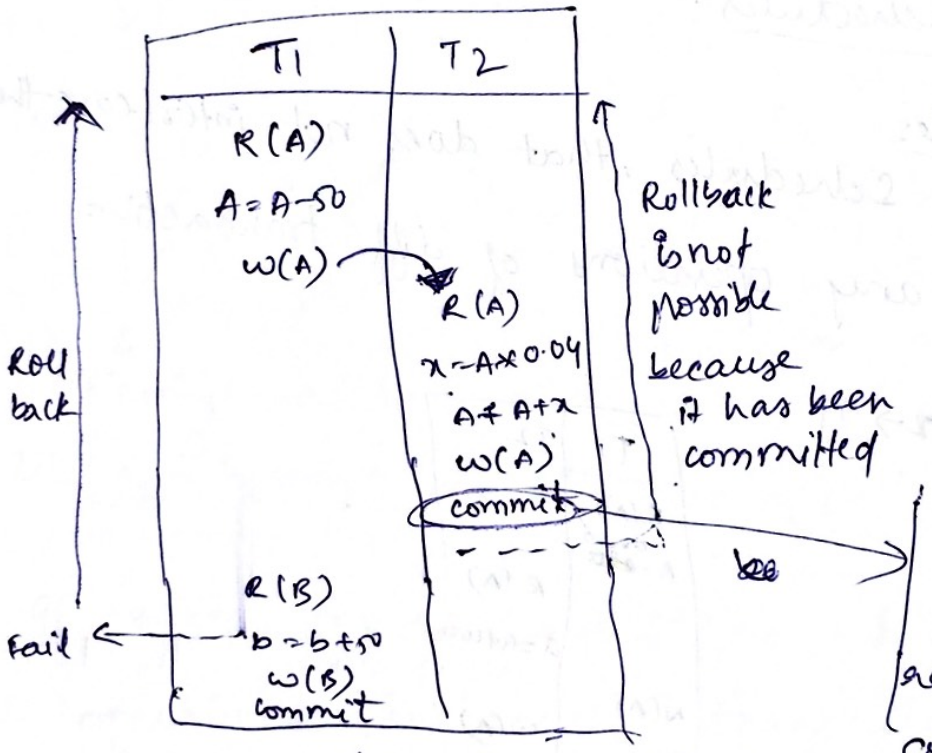
inconsistent state of operation

② Complete Schedule :-

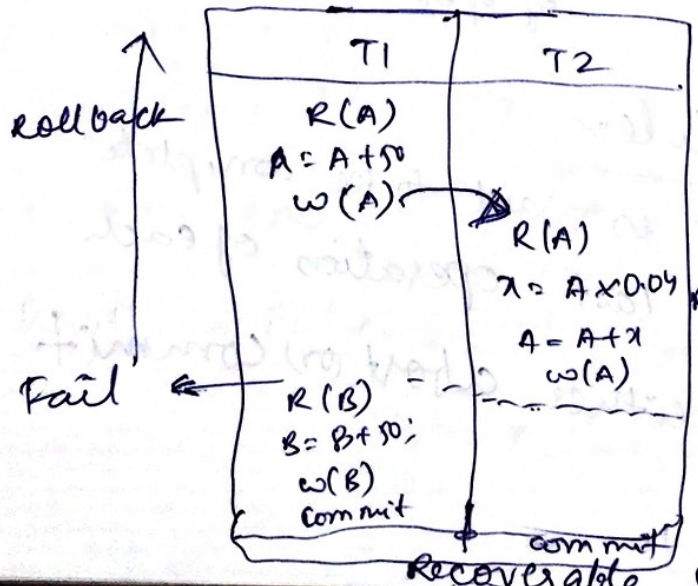
A schedule is said to be complete schedule, if the last operation of each transaction is either abort or commit



2) Recoverable Schedule:



Non-Recoverable Schedule

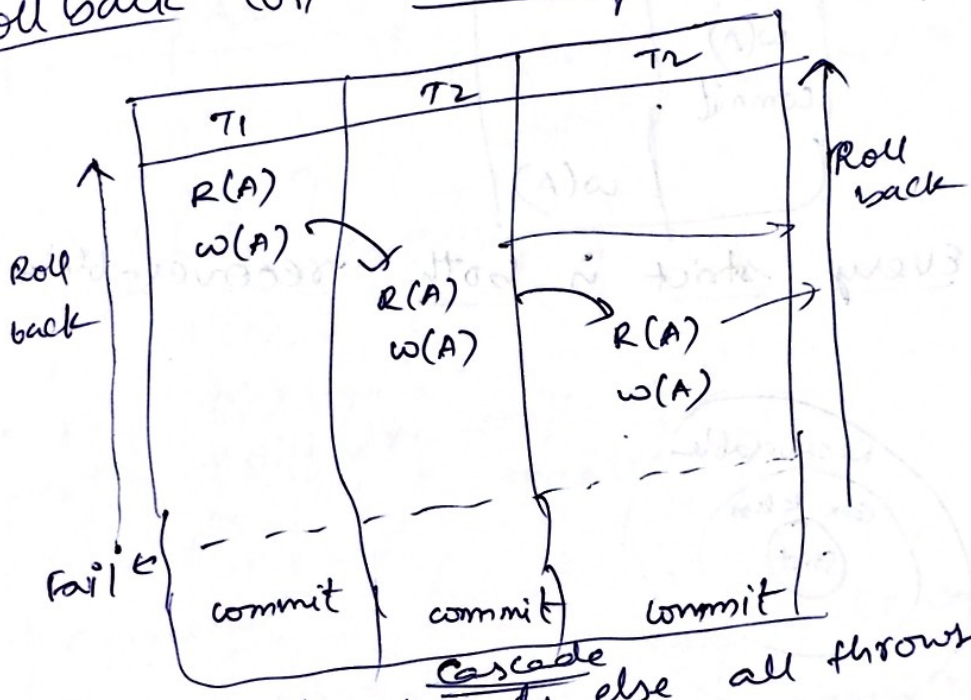


Recoverable Schedule

Ex: offering poisoned food, taking only after the person who offers consumes.

4) Cascadeless Schedule:-

If one transaction failures causes multiple transactions to roll back, called cascading rollback or cascading aborts

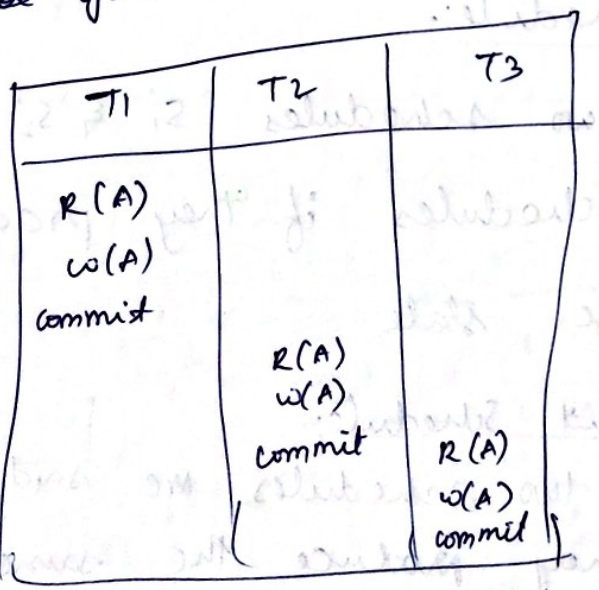


Cascade

→ If one eats all eats else all throws one after the other

Cascade less:-

→ you don't take it unless the other eats it



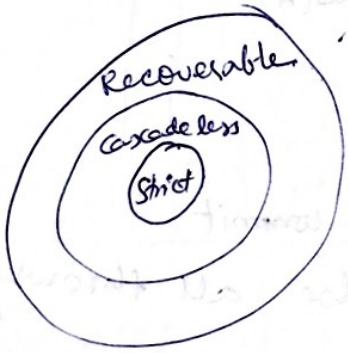
Cascadeless Schedule

Strict Schedule:-

→ It will not either read (or) write until the other commits unlike Cascadeless which only does not read until other commits

T_1	T_2
R(A)	
w(A)	
commit	
	w(A)

→ Every strict is both recoverable & Cascadeless



Result Equivalent Schedule:-

Equivalent Schedule:-

The two schedules 'S1' & 'S2' are said to be equivalent schedules if they produce the same final database state.

Result Equivalent Schedule:-

The two schedules are said to be result equivalent if they produce the same final database state for the same initial values of data.

ex:

S1	S2	
R(A)	R(A)	100
A=A+10	A=A*1.1	110
W(A)	W(A)	110

Conflict Equivalent & Conflict Serializable:

Conflict Operations

⇒ diff txn's on same data item with atleast one write

T_i T_j

R(A) ... W(A) } Conflicting operations
W(A) ... R(A) }
W(A) ... W(A) }
R(A) ... R(A) } non-conflicting operations
operating on diff data

Conflict equivalent:

Two schedules are said to be conflict equivalent if all conflicting operations in both the schedules must be executed in same order

S1

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	

S2

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	

Q Test which of the following schedules are conflict serializable

S_1 : $R_1(A)$ $R_2(B)$ $W_1(A)$ $W_2(B)$

S_2 : $R_2(B)$ $R_1(A)$ $W_2(B)$ $W_1(A)$

S_1

T1	T2
R(A)	
	R(B)
W(A)	
	W(B)

S_2

T1	T2
	R(B)
R(A)	
	W(B)
W(A)	

$S_1 \overset{\text{Conflict}}{\equiv} S_2$

2Q

S_1 : $R_1(A)$ $W_1(A)$ $R_2(B)$ $W_2(B)$ $R_1(B)$

S_2 : $R_1(A)$ $W_1(A)$ $R_1(B)$ $R_2(B)$ $W_2(B)$

S_1

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	

S_2

T1	T2
R(A)	
W(A)	
R(B)	
	R(B)
	W(B)

here write is occuring first

here read occuring first

$\Rightarrow$ so $S_1 \neq S_2$ are not conflict equivalent

S_1

T1	T2
	R(A)
R(A)	
	W(B)
W(B)	

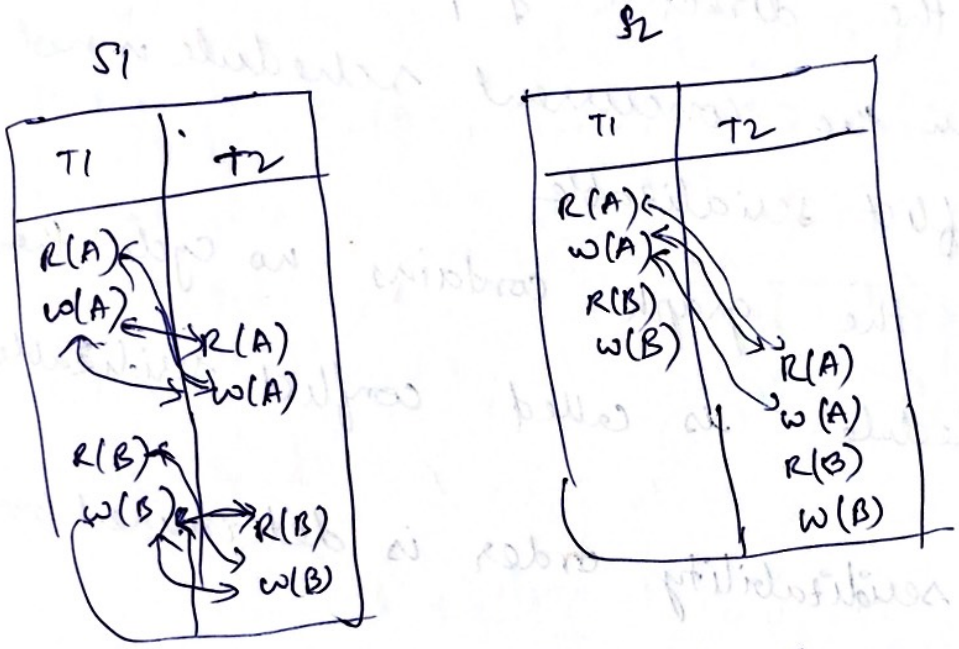
S_2

T1	T2
R(A)	
	R(A)
	W(B)
W(B)	

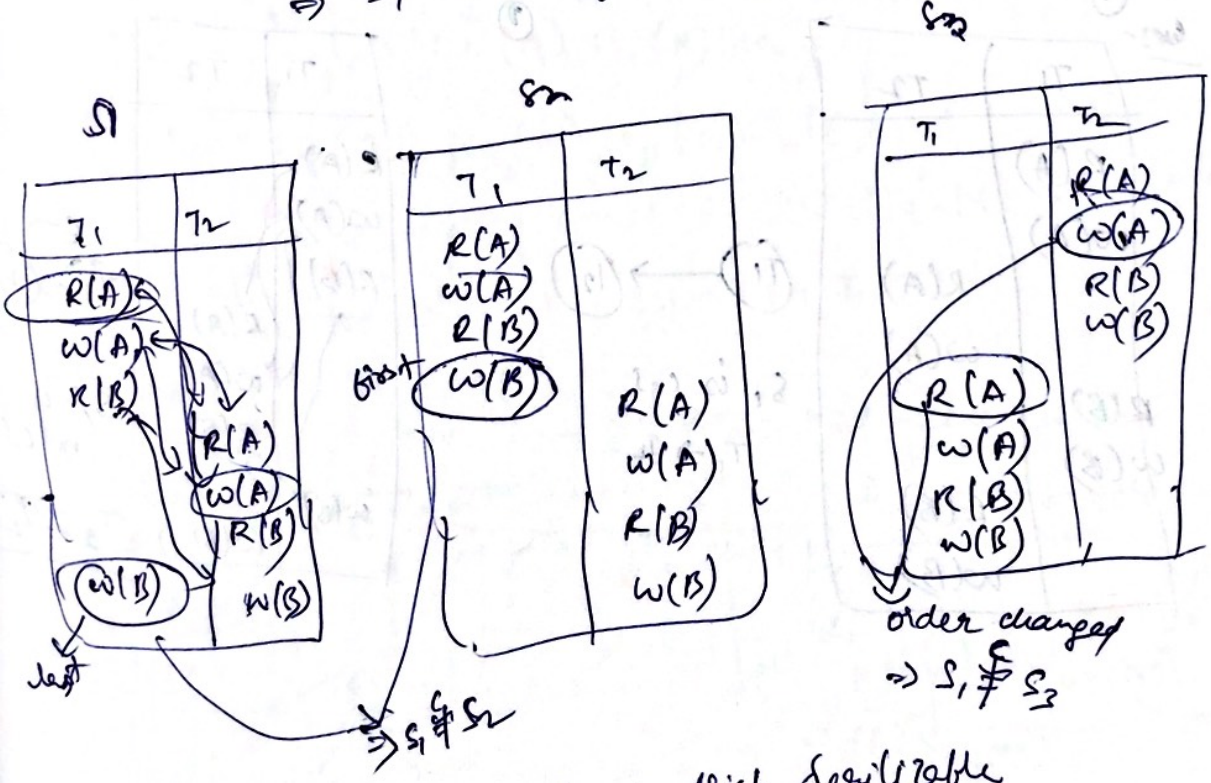
Conflict Serializable :-

A schedule is said to be conflict serializable if it is conflict equivalent to a serial schedule

ex: Given schedule, (i) check whether it is conflict serializable (or) not.



⇒ S_1 is conflict serializable



⇒ $S_1 \not\equiv S_2$

⇒ S_1 is not conflict serializable

Test for Conflict Serializability using precedence graph.

1. Construct a directed graph where each vertex corresponds to a transaction and each directed edge represents a conflicting operation
2. If the directed graph contains cycles, then the concurrent schedule is not conflict serializable
3. If the graph contains no cycle then the schedule is called conflict serializable

The serializability order is determined based on directed graph.

① Ex:-

T1	T2
R(A)	
w(A)	
	R(A)
	w(A)
R(B)	
w(B)	
	R(B)
	w(B)

S_1 is C.S

$T_1 \rightarrow T_2$

②

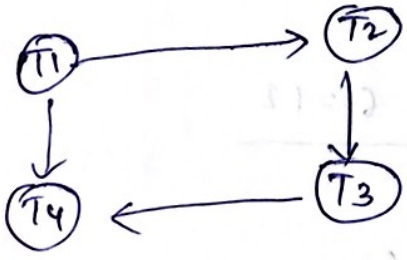
T1	T2
R(A)	
w(A)	
R(B)	
	R(A)
	w(A)
	R(B)
w(B)	
	w(B)

S_1 is not C.S

$T_2 \not\rightarrow T_2$

Find the serializability order of the concurrent schedule given below

$R_2(x)\ W_3(x)\ W_4(Y)\ R_2(Y)\ W_2(Z)\ R_4(Y)$



Serial Schedule

Eq $(T_1\ T_2\ T_3\ T_4)$

Q

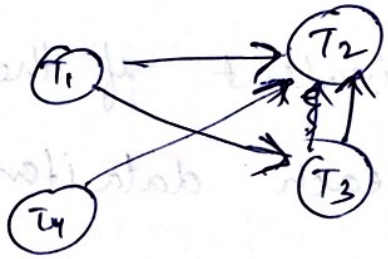
$R_4(A)\ ,\ R_2(A)\ ,\ R_3(A)\ ,\ W_1(B)\ ,\ W_2(A)\ ,\ R_3(B)\ ,\ W_2(B)$

a) not C-S

b) $T_3\ T_4\ T_1\ T_2$

c) $T_1\ T_4\ T_3\ T_2$

d) $T_2\ T_3\ T_1\ T_4$



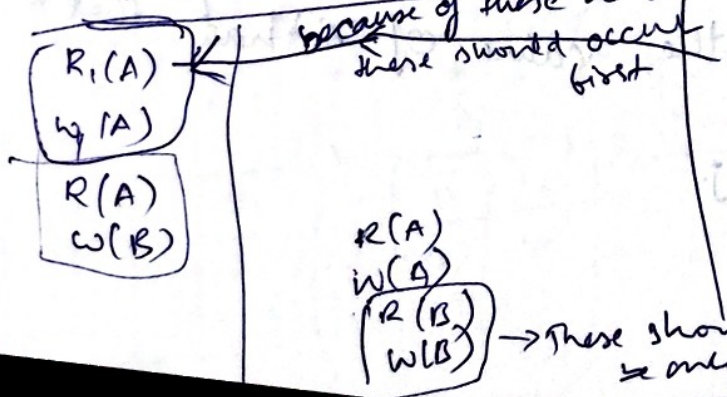
Two transactions, T_1 & T_2 are given as follows.

T_1 : $R_1(A)\ W_1(A)\ R_1(B)\ W_1(B)$

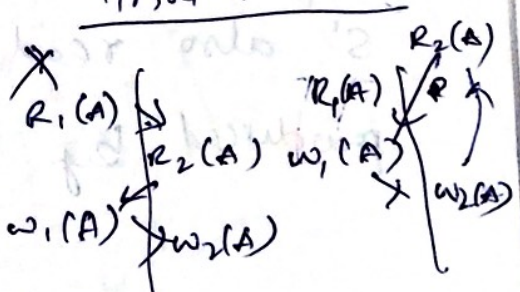
T_2 : $R_2(A)\ W_2(A)\ R_2(B)\ W_2(B)$.

Find the total no of conflict serializable schedules that can be formed by T_1 & T_2?

Ans: $T_1 \rightarrow T_2$ (or $T_2 \rightarrow T_1$) are serial schedules



* Trial & error



$$\Rightarrow \frac{4!}{2! \, 2!} = 6$$

$T_2 \rightarrow T_1$ also '6' possibilities

$\Rightarrow$ Total = $6 + 6 = 12$

View Equivalent Schedule:

Two schedules S & S' are said to be view equivalent if the following 3 condn's are met for each data item (say A)

i) For each data item 'A', if transaction T_i reads the initial value of 'A' in schedule 'S' then transaction T_i must in schedule 'S' also read the initial value of (A)

ii) If transaction, T_i executes read - R(A) in schedule 'S', and that was produced by Transaction, T_j (if any), then transaction T_i must in schedule 'S' also read the value of 'A' that was produced by T_j.

iii) For each data item the transaction (if any) that performs final write of 'A' operation in 'S' must also perform the final write of 'A' operation in "S'".

Note:- (Summary):-
All write - read sequences to be maintained in the same order in both 'S' & "S'".

(x):

S1

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

S2

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Rules :	A	B
i) first read	T_1, T_1	T_1, T_1
ii) last write	T_1, T_2	T_2, T_2

When a ~~view equivalent schedule~~ when a Schedule 'S' is view equivalent to its serial schedule then it is "View Serializable"

Ex:

S1

T_1	T_2
R(A)	
	W(A)
W(A)	

Serial "2"

S2

T_1	T_2
R(A)	
W(A)	
	W(A)

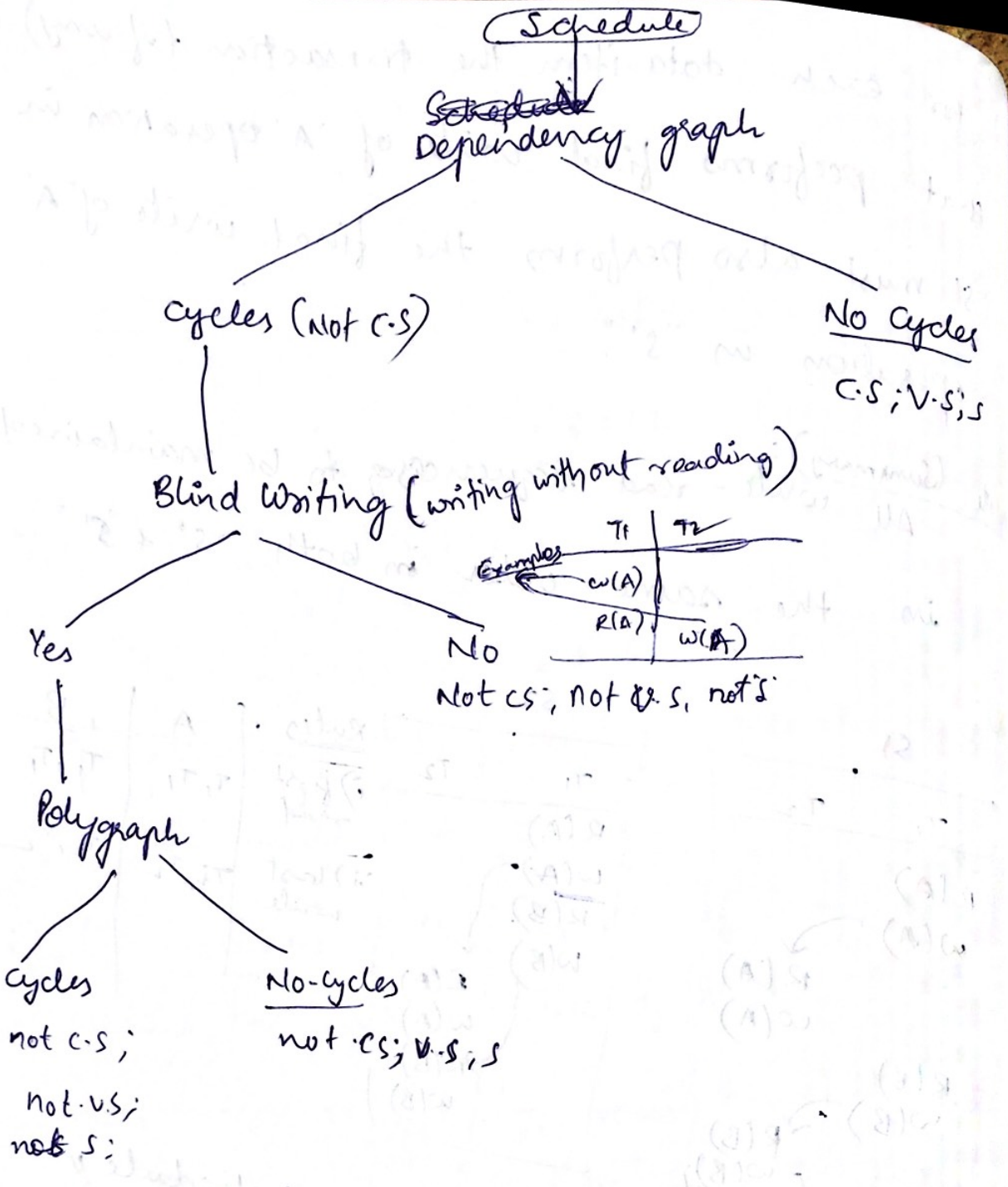
Rule	A
i)	T_1, T_1
ii)	T_1, T_2 ✗

$\Rightarrow S_1 \neq S_2$

Serial "1"

T_1	T_2

✗



Serializable Schedule:- A Schedule is serialisable if it is either conflict serialisable (or) view serialisable (or) both.

→ concurrent schedules = 6 = $\frac{4!}{2!2!}$ = $\frac{24}{4}$ = 6

Non-serial schedules = 6 - 2 = 4

Note: The no of concurrent schedules that can be formed over no of transactions having $n_1, n_2, n_3 \ldots n_m$ operations respectively are

$$= \left[\frac{(n_1 + n_2 + n_3 + \ldots n_m)!}{n_1! * n_2! * n_3! \ldots n_m!} \right]$$

The no of non-serial schedules are,

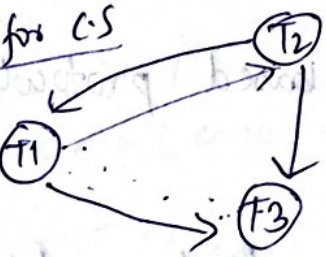
$$\left[\frac{(n_1 + n_2 + n_3 + \ldots n_m)!}{n_1! * n_2! * n_3! \ldots n_m!} \right] - n!$$

Examples of
i) Un-repeatable Read pblm
ii) Lost Update Pblm

Examples of polygraphs for View serialisability

$S = r_1(A)\, w_2(A)\, r_3(A)\, w_1(A)\, w_3(A)$

i) checking for C.S



⇒ cycle ⇒ not C.S
then check for blind writes

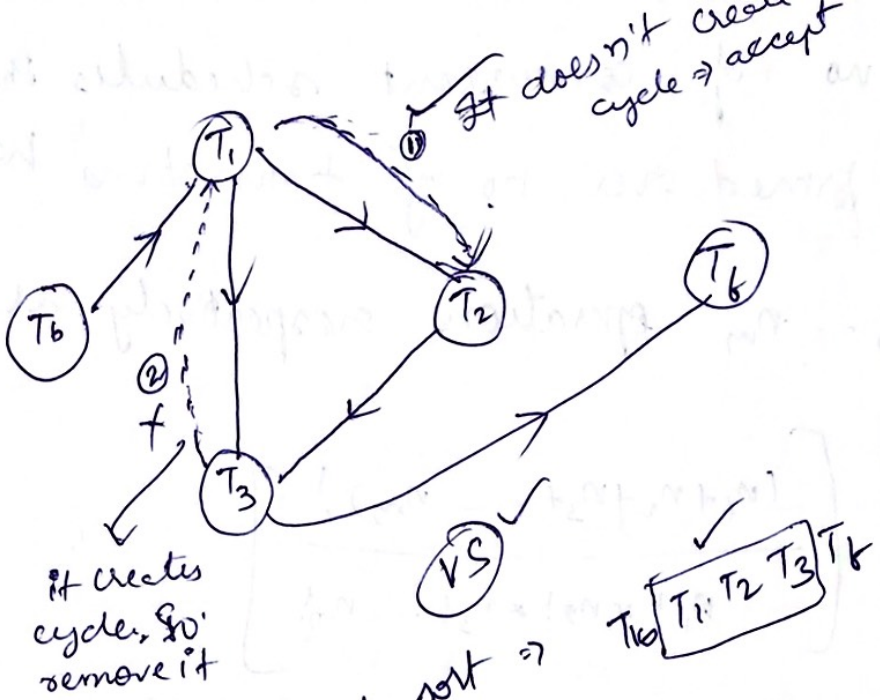
Yes, there is a blind write $w_2(A)$. So, check for V.S

ii) Checking for V.S:-

T_0	T_1	T_2	T_3	T_f
$w(A)$				
	$r(A)$			
		$w(A)$		
	$w(A)$		$r(A)$	
			$w(A)$	$r(A)$

conflicting position should be placed properly

going to read finally from all the data items



while topological sort ⇒

Concurrency Control:-

1) Lock based Protocols:-

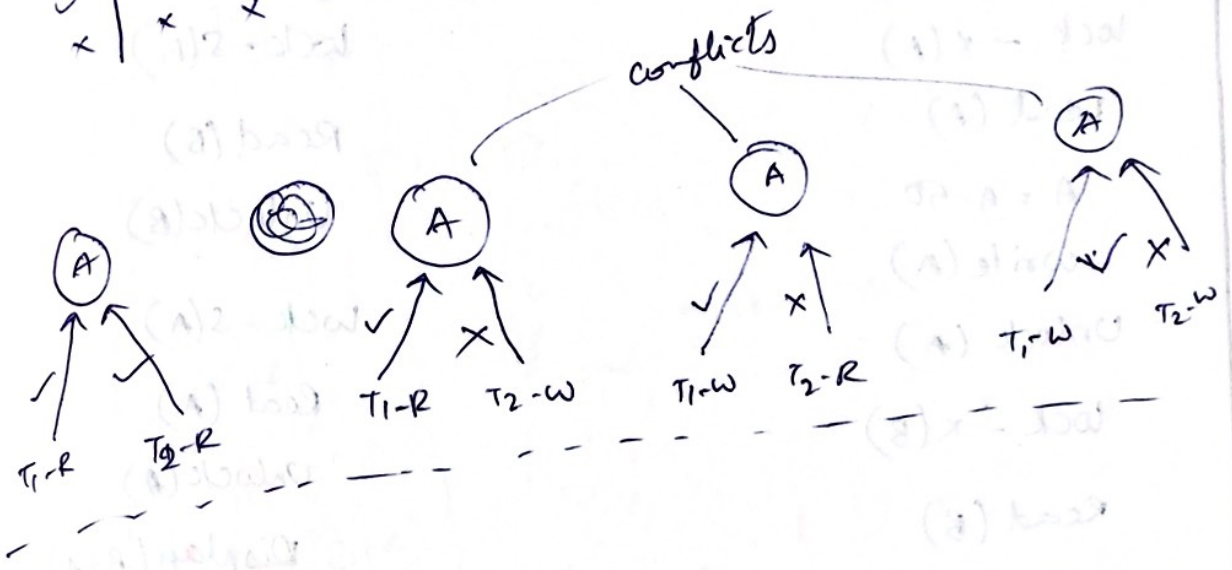
which requires that all data items must be accessed in a mutually exclusive manner. i.e, when one transaction is accessing the data item, no other transactions simultaneously update the data item

To implement lock based protocols we need two locks

1) Shared lock : only Read (lock-S)
2) Exclusive lock : both Read & write (lock-X)

	S	X
S	✓	X
X	X	X

} compatibility b/w lock models



→ ~~Conflict~~ concurrency control manager in DBMS takes care of concurrency with some ~~rules~~ rules kept in mind.

→ Serial schedules are logical schedules

→ ~~Bef~~ & Before generating & executing schedule itself, we have to check the serializability

→ Rules to be followed before generating schedules with CCM (conc. control Manager).

Three protocols

1. lock based protocol
2. graph based " "
3. timestamp based " "

T_1 : A $\xrightarrow{50}$ B

```
lock - X(A)
Read (A)
A = A - 50
write (A)
Unlock (A)
lock - X(B)
Read (B)
B = B + 50
write (B)
Unlock (B)
```

T2 : Display (B+A)

```
lock - S(B)
Read (B)
Unlock (B)
lock - S(A)
Read (A)
Unlock (A)
Display (B+A)
```

Problems in Interleaving

Inconsistent Result

(1)

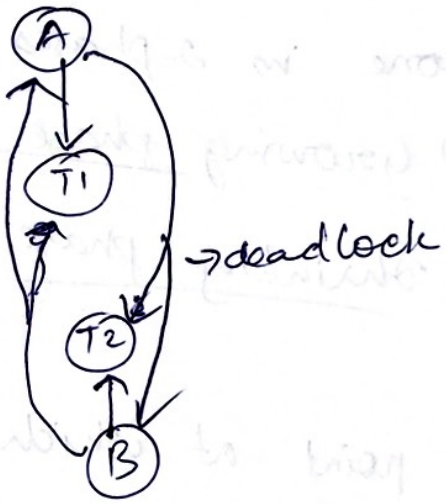
T1	T2
lock-X(A)	
Read(A)	
A = A-50	
write(A)	
Unlock(A)	
	lock-S(B)
	Read(B)
	Unlock(B)
	lock-S(A)
	Read(A)
	Unlock(A)
	display(B+A)
lock-X(B)	
Read(B)	
B = B+50	
write(B)	
Unlock(B)	

	A	B	
before T_i	100	200	= 300
After T_i	50	250	= 300

but Op = 250 ⇒ error

② Deadlock

T1	T2
lock-X(A)	
R(A)	
A=A-50	
W(A)	
	lock-S(B)
	R(B)
	lock-S(A)
	R(A)
	display(B+A)
	commit
	unlock(B)
	unlock(A)
lock-X(B)	
R(B)	
B=B+50	
W(B)	
commit	
unlock(B)	
unlock(A)	



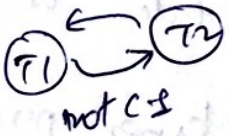
③ Simple locking cannot always provide Serialisation

T1	T2
L(A)	
U(A)	
	L(B)
	U(B)
L(B)	
U(B)	

T2 → T1

④

T1	T2
L(A)	
U(A)	
	L(A)
	U(A)
	L(B)
	U(B)
L(B)	
U(B)	



not CS

2-Phase Locking Protocol:

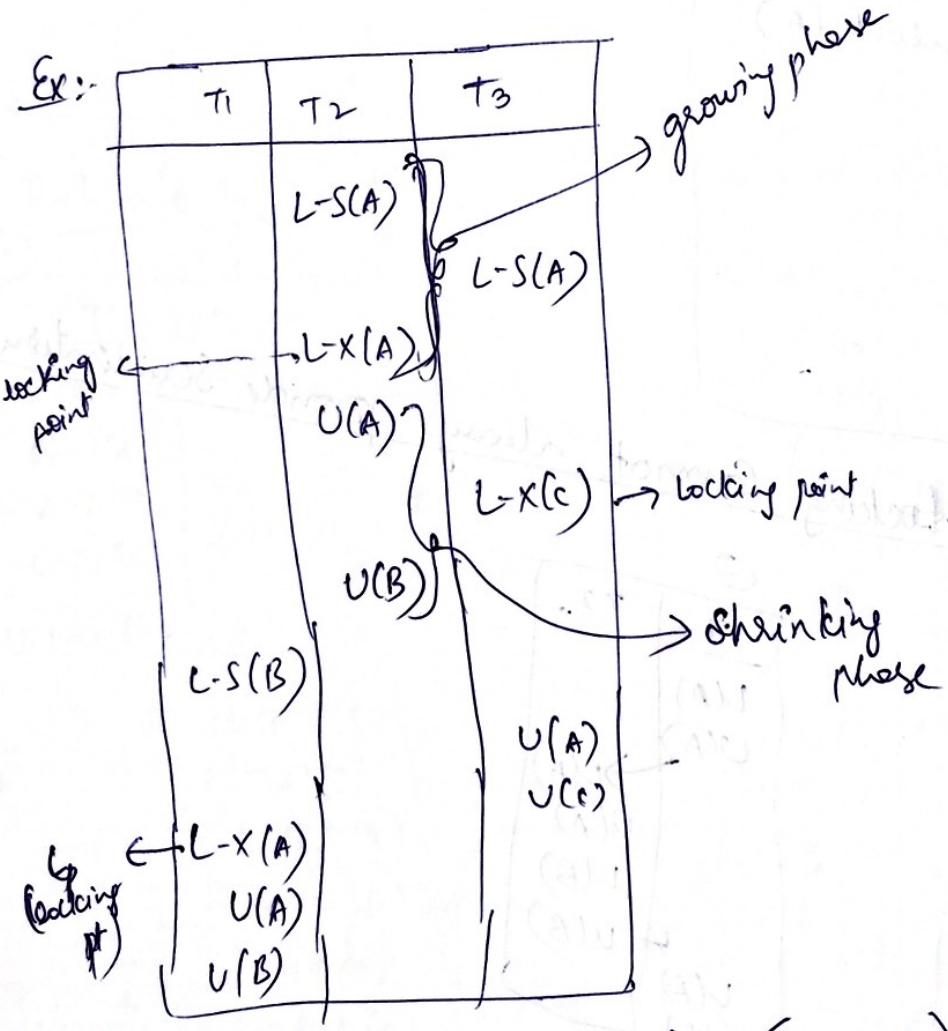
which requires both locks & unlocks being done in 2-phases

1) Growing phase – "obtains locks"
2) Shrinking phase – "Release locks"

Lock Point:

The point at which the transaction has obtained its final lock is called a "Lock point"

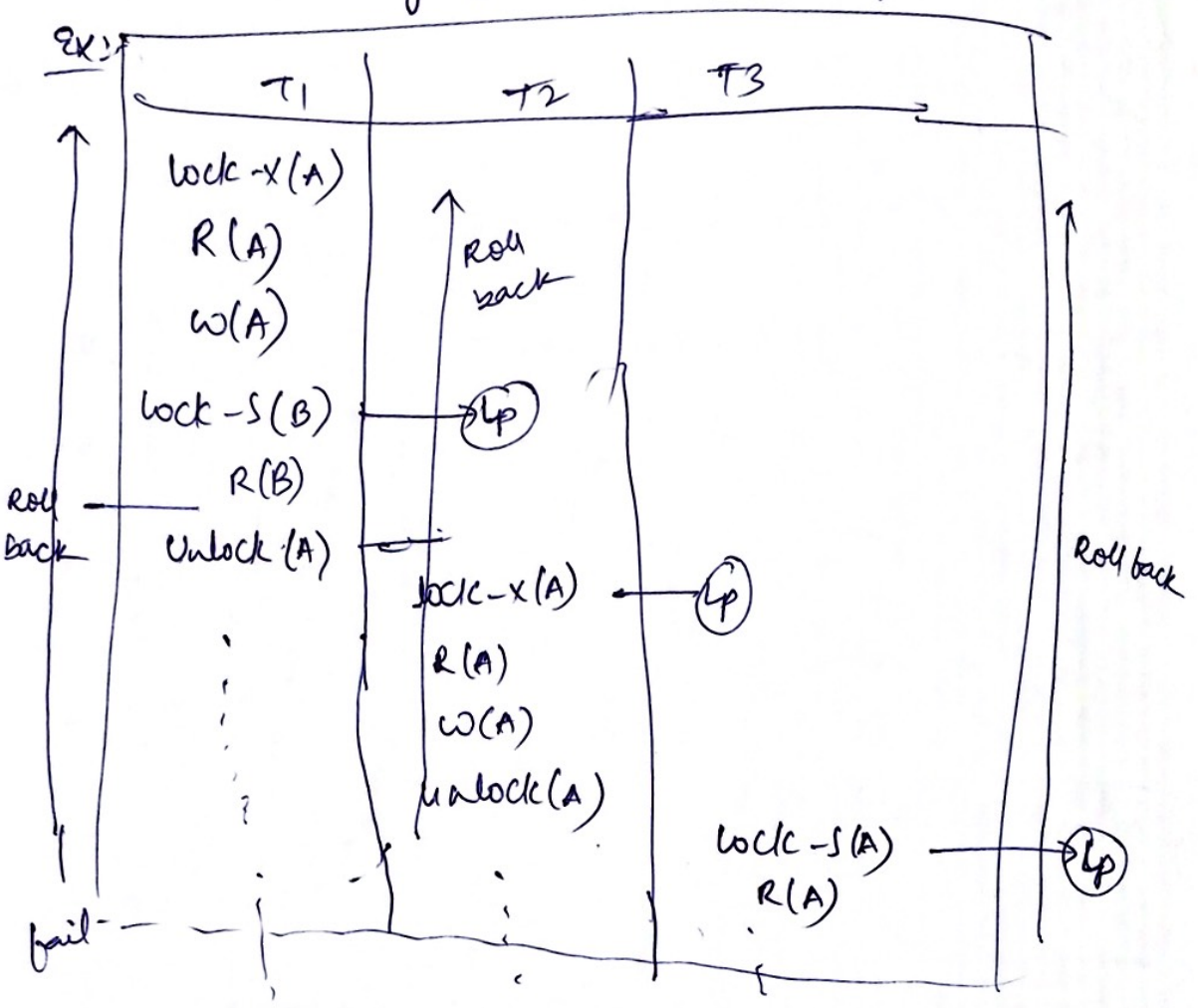
Ex:-



⇒ Serial Schedule is $(T_2\ T_3\ T_1)$

⇒ Two phase locking protocol ensures conflict serializability & Serializability order is determined based on their lock

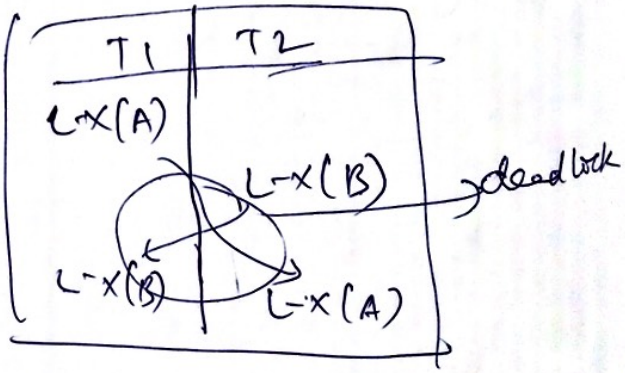
→ 2-phase locking protocol fails during Rollbacks



→ Cascading rollbacks may occur in 2-phase locking protocol

* Deadlocks may also during 2-phase locking protocol

Ex:-



→ Cascading Rollbacks can be avoided by a modification of 2-phase locking protocol

1) Strict 2-phase locking protocol (strict 2PL):-

which requires that in addition to locking being 2-phase all exclusive mode locks taken by a transaction must be held until the transaction commits

2) Rigorous - 2-phase locking protocol (Rigorous 2PL):-

which requires that in addition to locking being 2-phase all locks must be held until the transaction commits.

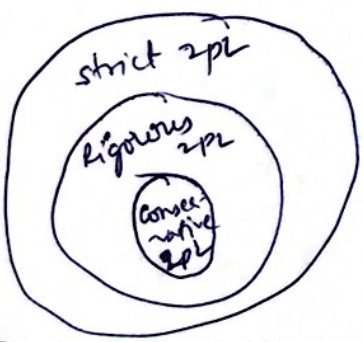
∴ Avoids cascading Rollbacks but deadlocks can occur

3) Conservative 2PL:-

which requires the transaction to update all the locks before it starts & release all the locks after it commits

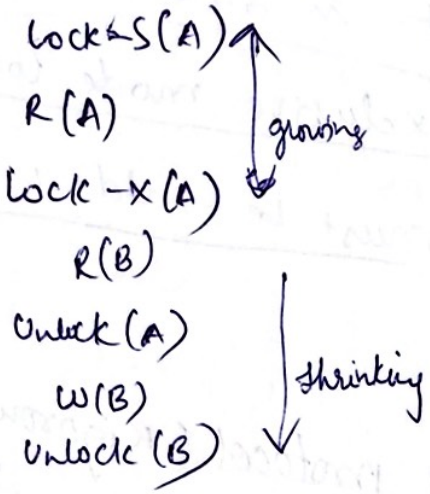
Avoids cascading Rollbacks & deadlocks

Disadv
→ holds lock for long time
→ suppose to know all requirements



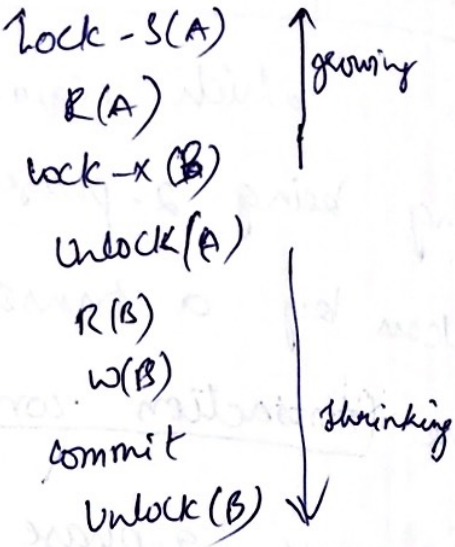
Examples of 2PL

1)



Sol:- 2PL but not Strict, Rigorous (or) Conservative

2)



2PL, strict, but not Rc & Con.

3)

Lock-S(A)
R(A)
Lock-X(B)
R(B)
W(B)
commit
Unlock(A)
Unlock(B)

Strict, Rigorous 2PL

4)

Lock-S(A)
Lock-X(B)
R(B)
W(B)
R(A)
commit
Unlock(A)
Unlock(B)

Strict, Rigorous, conservative 2PL

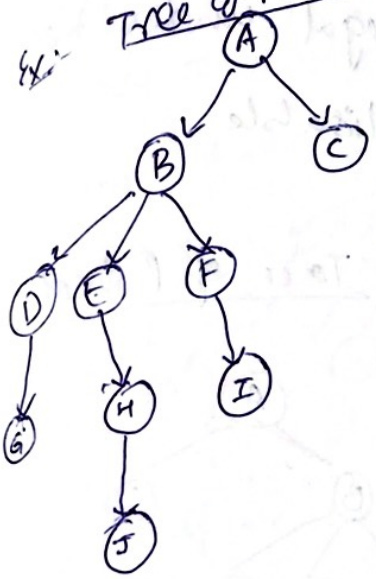
* In growing phase you can upgrade a shared lock to Exclusive lock (or) In shrinking phase you can degrade a lock from exclusive to shared lock

Graph based Protocol

→ To overcome the disadvantages of locking protocols we have developed graph based protocol.

→ we use trees in GBP

Ex: Tree of Protocol



1) Locks must be done in order of the tree
Ex:- for A, B, H
A → B → H ✓

2) In tree no cycles will be formed

Rules:

In tree protocol the only lock instruction allowed is "lock-X" (exclusive) and each transaction T_i can lock a data item atmost once and must observe the following rules

1) The first lock by T_i may be on any data item
2) Subsequently a data item can be locked by T_i only if the parent of the data item is currently locked by T_i.
3. Data items can be unlocked at any time

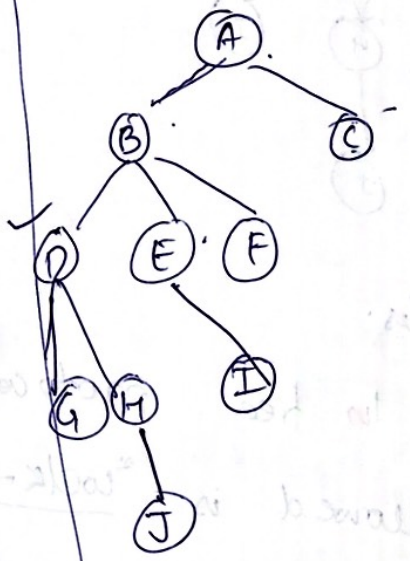
4. A data items that has been locked and unlocked by T_i cannot subsequently relocked by T_i.

NOTE:-

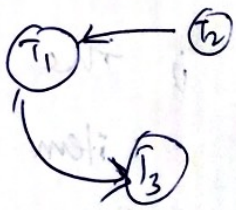
All schedules that are legal under tree protocol are conflict serialisable

T_1	T_2	T_3
lock-X(B)		
	lock-X(B)	
	lock-X(H)	
	unlock(D)	
lock-X(E)		
lock-X(D)		
unlock(B)		
unlock(E)		
		lock-X(B)
		lock-X(E)
	unlock(H)	
lock-X(G)		
unlock(D)		
		unlock(E)
		unlock(B)
unlock(G)		

Tree of Protocol



Conflict Serializable



$T_2 - T_1 - T_3$

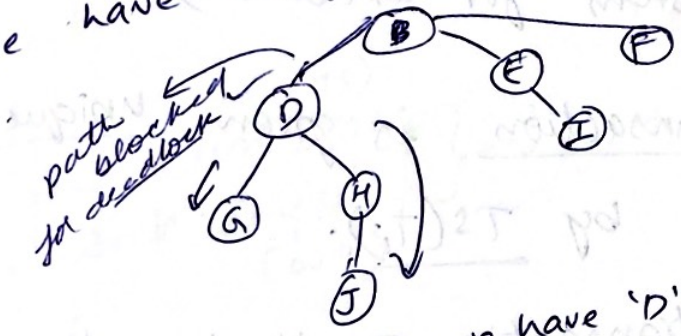
Advantages:

1. Unlocking will be done much earlier, as we are not going to wait until commit is done

2. There are no deadlocks:-

because, deadlock occurs if we wait for a resource which is held by other, but here we have required resource as we have prior

Ex:



if we want 'H' we have 'D' (parent) & no one can have 'H' as we lock the way at 'D' by having it

3. Ensures conflict serializability

Disadvantages:

1. Requires prior knowledge about the orders of the data items.

2. Unnecessary locking overheads i.e, In some cases lock the data items that it does not access

Timestamp Ordering Protocol

Timestamp based protocol:

which requires that ordering among the transactions is determined in advance based on thier timestamps (i.e, the time when ever it enters into the system for executions)

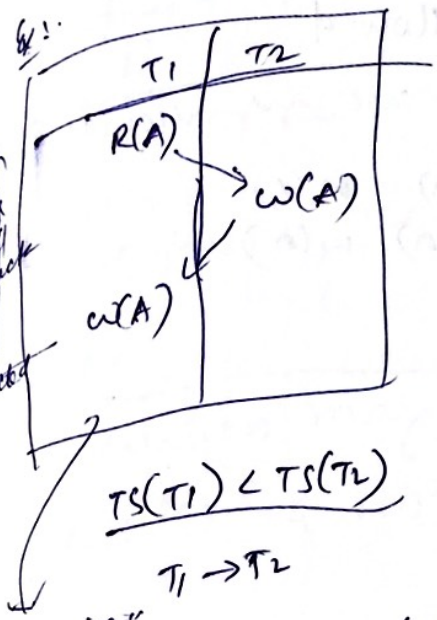
Each transaction is given unique fixed time-stamp denoted by $TS(T_i)$.

If any transaction T_j that enters after T_i, the relation between their timestamps be $TS(T_i) < TS(T_j)$ which means that the producing schedule must be equivalent to a serial schedule $T_i \rightarrow T_j$

Timestamp Ordering Protocol:-

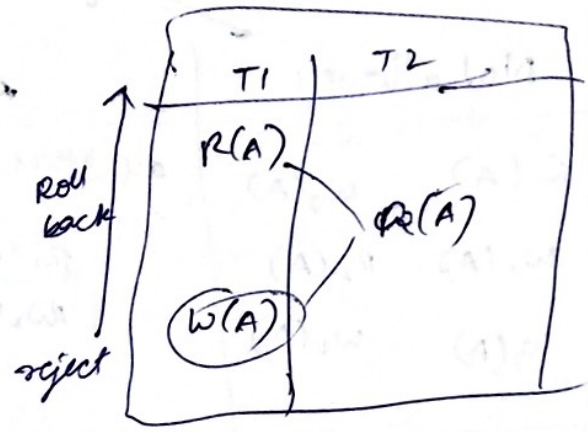
In timestamp ordering protocol ensures that ~~they~~ any conflicting read & write operations are executed in timestamp order, if not such an operation is rejected and the transaction will be rolled back. The rolled back transaction will be restarted with a new timestamp.

Ex:



$TS(T_1) < TS(T_2)$

$T_1 \rightarrow T_2$

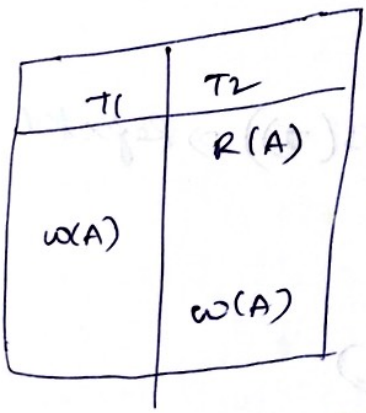


$TS(T_1) < TS(T_2)$

Read Timestamp (RTS) } are used to implement
write Timestamp (WTS) }

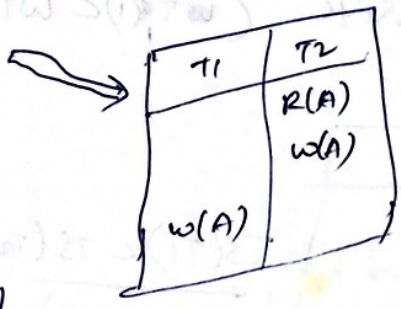
Thomas Write Rule:-

which requires to ignore all obsolete write operations

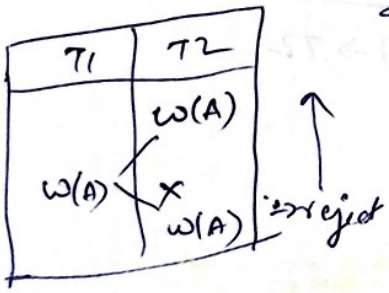


$TS(T_2) < TS(T_1)$

$T_2 \rightarrow T_1$



* It is saying that late writes can be ignored



→ Reject and Rollback

$T_2 < T_1$

	Not allowed	Allowed
TSP	$R_1(A)$ $W_2(A)$ $W_1(A)$ $R_2(A)$ $W_1(A)$ $W_2(A)$	all other cases are allowed Ex :- $R_2(A)$ $W_1(A)$ $W_2(A)$ $R_1(A)$
TWR	$R_1(A)$ $W_2(A)$ $W_1(A)$ $R_2(A)$	

Implementation

for each data item, (ex: A) it will maintain the values RTS, WTS & check when a new txn comes with the values,

$T_1 = 1$

$T_2 = 2$

	A
RTS	2
WTS	2

if T_1 tries to write $(WTS(1) < WTS(2)) \Rightarrow$ Rejected

Ex:

T1	T2
R(B)	
	R(B) B = B-50 W(B)
R(A)	
	R(A)
display(A+B)	
	A = A+50 W(A)
	Display (A+B)

$TS(T1) < TS(T2)$

$\Rightarrow T1 \rightarrow T2$

Advantages of TSP:

1. This protocol ensures serialisability
2. Ensures freedom from deadlock

Disadv:

Starvation may occur due to continuously getting aborted & restarting the Txn.

Organisation of Records in a file:-

Ordered file Organisation :-

All the records of file are ordered based on some search key value

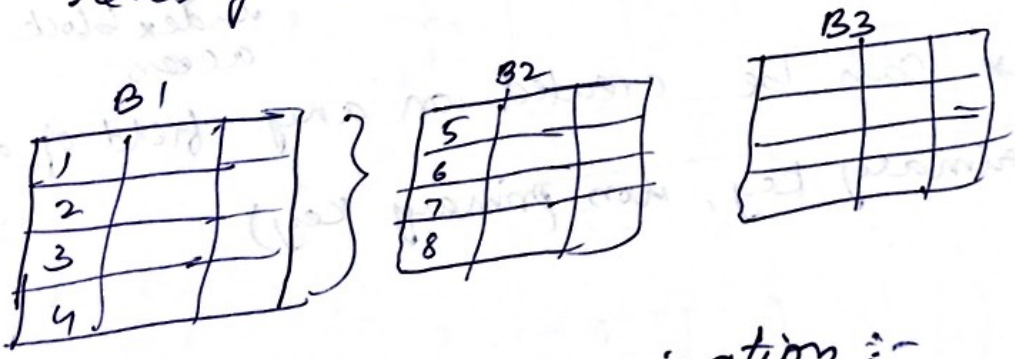
Searching: Binary Search

B- data blocks

To access a record, the average no of block access = $\boxed{\log \frac{B}{2}}$ blocks

Adv: Searching is efficient

Disadv:- Insertion is expensive due to re-organisation of the entire file.



Un-Ordered File Organisation :-

All the records of file are inserted at wherever the place is available usally at the end of the file

Searching: 1. Linear search

B - data block

to access a record, the average no of block access = $\frac{B}{2}$ blocks

Adv:- Insertion is efficient

Disadv:- Searching is inefficient compared to ordered file organisation.

Indexing

Index:-

1. Indexes are used to improve search efficiency
2. Index is a record consists of two fields

Key	Block positions

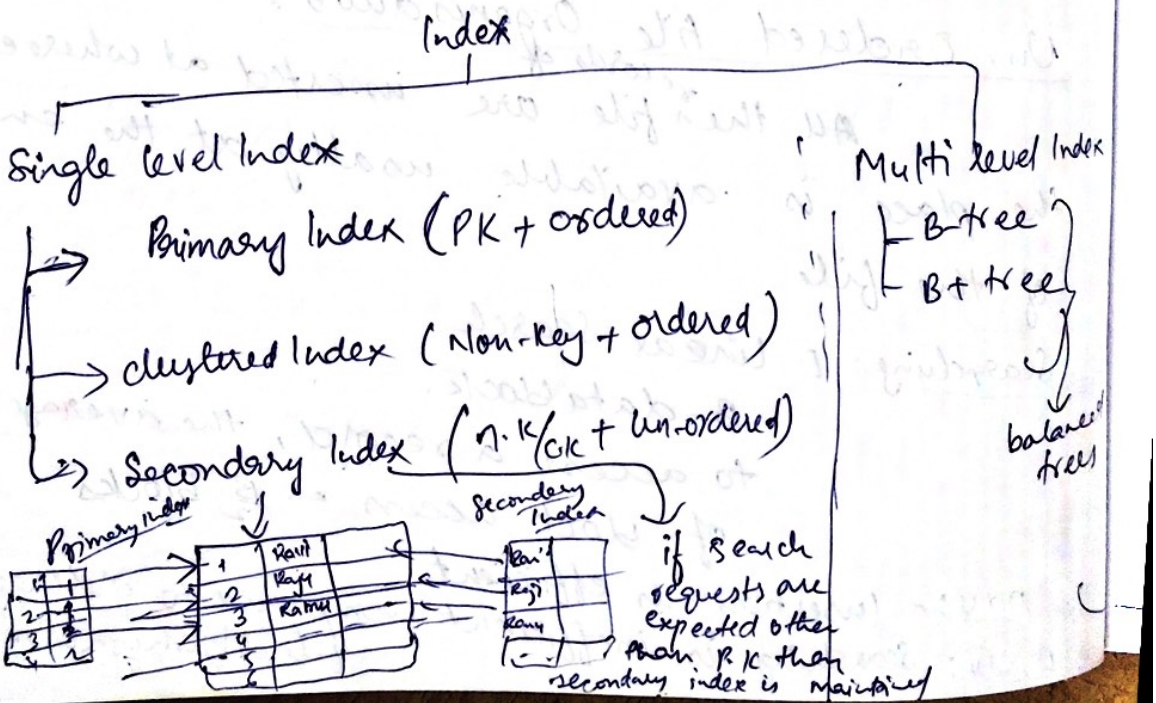
↳ pointer to a block where key is available

3. Index is an ordered file
4. Searching :- Binary Search
5. To access a record using the index,

the avg no of block accesses = $\log_2 B_i + 1$

- $\log_2 B_i$: index block access
- $+1$: data block access

6. Index can be created on any field of a relation (primary key, non primary key)

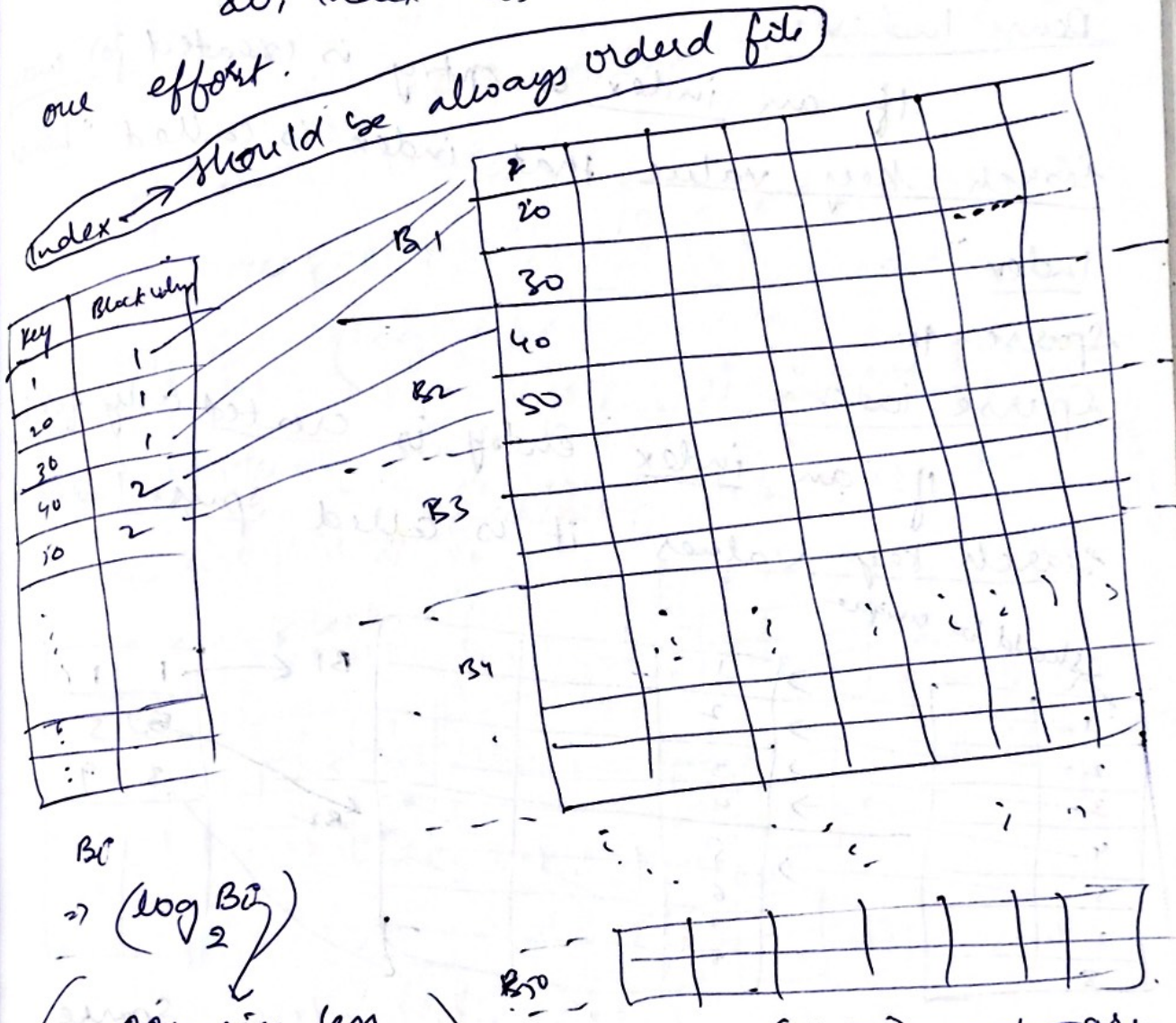


Self Explanation

consider you have a large file with many blocks & attributes. & searching all the blocks takes huge cost.

So, Index is maintained which reduces one effort.

Index → should be always ordered file



B_r

$\Rightarrow (\log_2 B_r)$

(occupies less no of blocks due to only 2 attributes in file)

B_d ++ searches are required

$\log_2 B_d$ searches are required

$\Rightarrow$ Requires $(\log_2 B_r + 1) \Rightarrow$ no of searches

$\log_2 B_r$ → index block access

1 → data block access

Classifications of Indexes:-

1. Dense Index
2. Sparse Index

Data should be ordered for Sparse Index

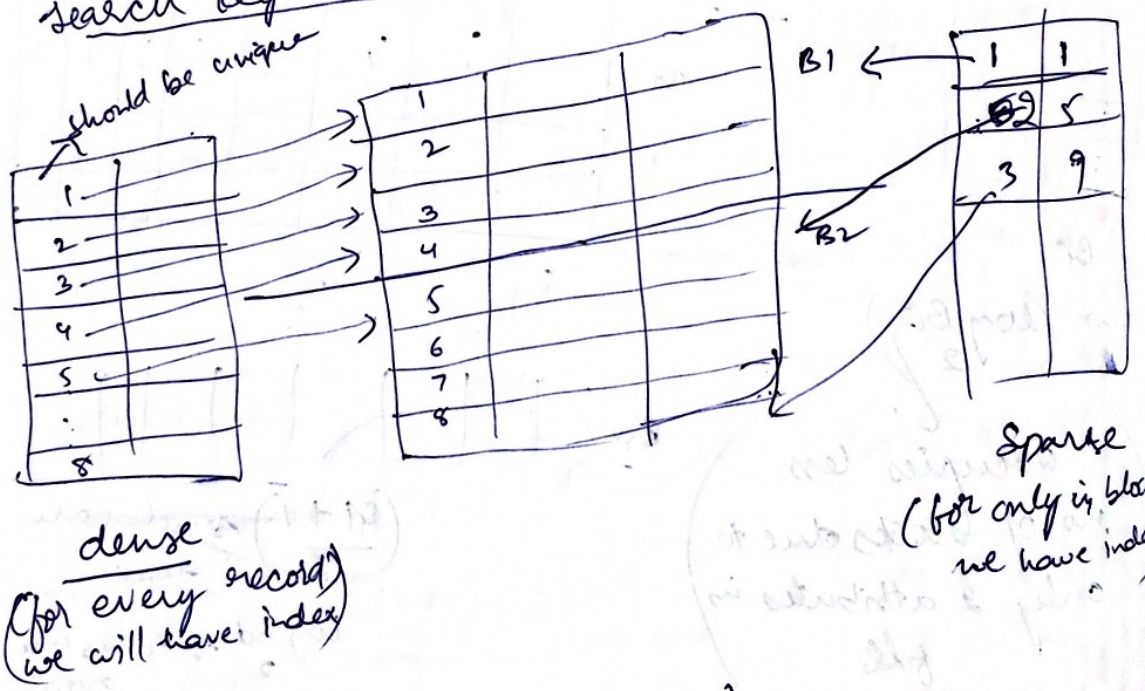
Dense Index:-

"If an index entry is created for every search key value that index is called "Dense Index"

Sparse Index:-

If an index entry is created only for some search key values it is called Sparse Index

should be unique



dense
(for every record we will have index)

Sparse
(for only 3 blocks we have index)

Primary Index (P.K + Ordered):-

A primary index is an ordered file whose records are of fixed length with two fields, the first field is same as P.K of data file and the second field is "

pointer to data block where the key is available

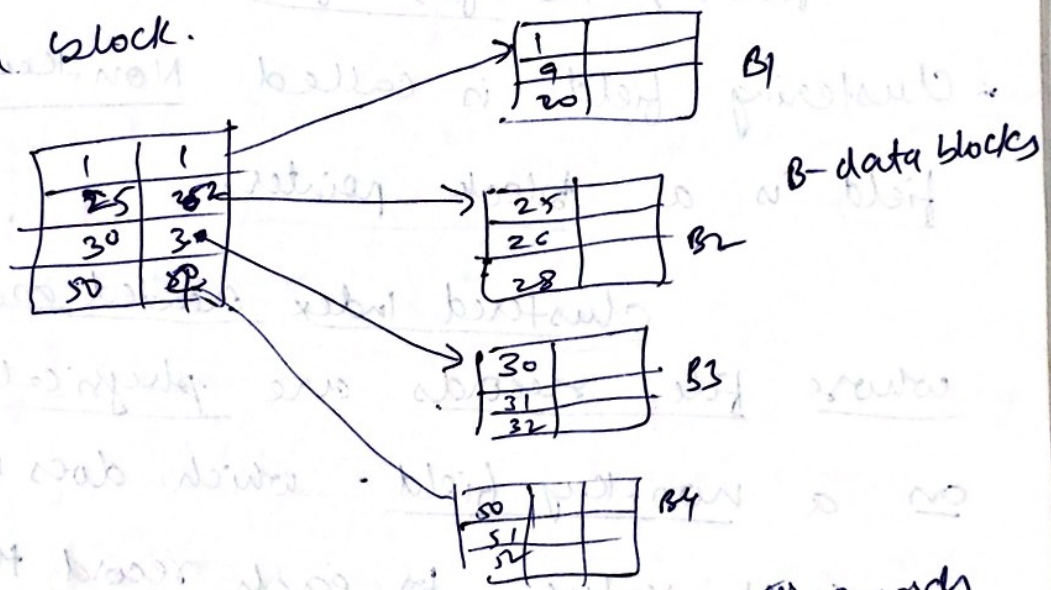
2. Index entry is created for first record of each block called "block Anchor" (Sparse Index)

3. The no of index entries is equal to the no of data blocks

4. The average no of block access required will be $\left(\log_2 b_i + 1\right)$ → index blocks

5. The type of index is called Sparse index because it is indexing only first record of each block.

6.



Ex: (a) Records = 30000 ; block size = 1024 bytes ; unspanned records size = 100 bytes ; Suppose primary index created of Key field of size 9 bytes & block pointer of 6 bytes ⇒ key + blk ptr = 6+9 = 15 bytes = each record

Find the avg no of blk accesses to search for a record using with & without indexing?

Sol:- Records = 30000 ⇒ blocks = $\frac{30000}{10}$ = 3000

blks without indexing = $\lceil \log_2 3000 \rceil$ = 12

Index records / block = $\lfloor \frac{1024}{15} \rfloor$ = 68

Index records = 3000

⇒ no of blocks of index = $\frac{3000}{68}$ = 45

⇒ Avg no of blk access = $\lceil \log_2 45 \rceil + 1$

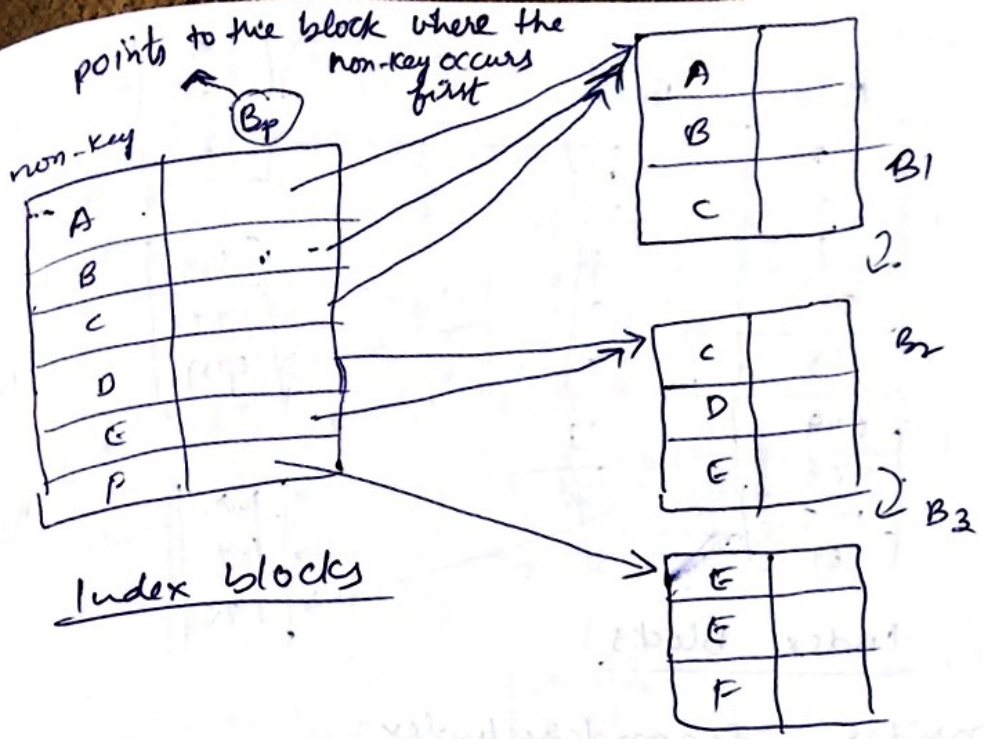
(with indexing) = 7

Clustered Index:- (Non Key + Ordered):-

clustered Index is an ordered file with two fields, the first field is same as the clustering field is called Non-key & the second field is a block pointer.

clustered index created on data file whose file records are physically ordered on a non-key field which does not have a distinct value for each record that field called clustering field"

* Index entry is created for each distinct value of a clustering field
* the block pointer points to the first block in which the key is available.



*) Type of Index is Dense / Sparse

→ The average no. of block access using index

$$= \left(\log_2 B_i'' + 1\right)$$

Secondary Index ((n.k) (or) Candidate Key + Unordered) :-

1. Secondary index provides a secondary means of accessing a file for which some primary access already exist

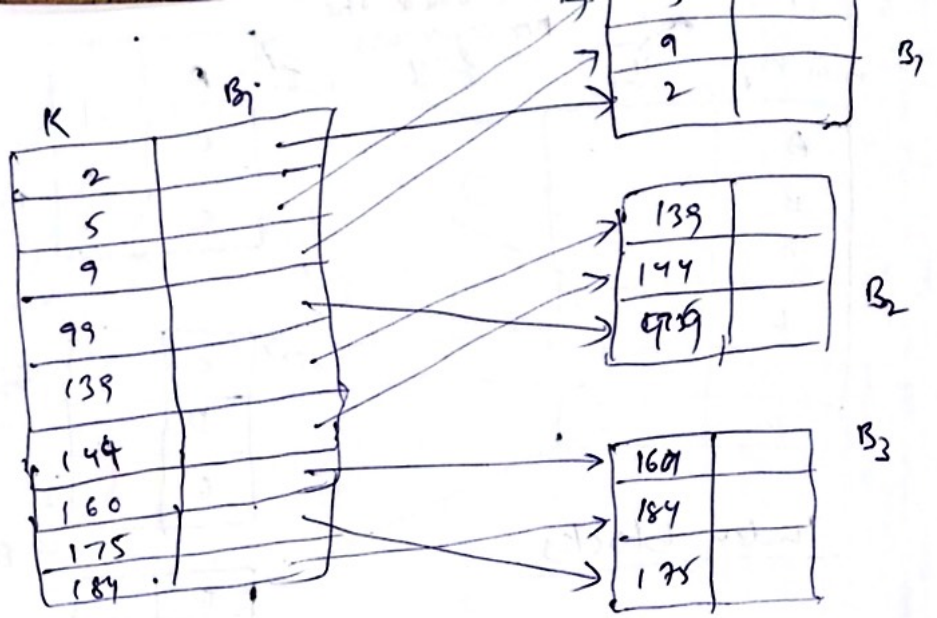
2. Secondary index may be on a non-key (or) a candidate key.

3. Index entry is created for each record in a data file.

4. No of index entries - No. of records

5. Type of secondary index - dense

Ex:-



Index blocks

Q. Consider secondary index: no. of records = 30,000; block size = 1024 bytes; Records unspanned; Record size = 100 bytes; Index P.K = 5 bytes blk ptr = 9 bytes; find the avg no of blk access with & without indexing

Sol:

~~without indexing~~ = ~~to be previously calculated~~

without indexing ⇒ linear search ~~no of~~ all the blks should be accessed

⇒ $\frac{30000}{10}$ = 3000 blocks (worst case)

Avg case = $\frac{3000+1}{2} = \frac{3001}{2}$ (worst case → best case)

with indexing

⇒ index is ordered

no of records/block = $\frac{1024}{15} = 68$ records/block

dense index ⇒ no. of blks = $\frac{30000}{68} = 442$ blks

⇒ Avg no of blk accesses = $\left(\log_2 442 + 1\right) = 9+1 = 10$ blk

⇒ with indexing 10 block access

data records = $16,384 = 2^{14}$; Record size = 32 bytes = 2^5
unspanned. Block size = 1024 bytes;
key field size = 6 B; Blk ptr size = 10 B.
If secondary index is built on the key field of the file and multi-level index scheme is used the no. of 1st level and 2nd level blocks in multilevel index respectively are?

Sol: no. of records/block in data file = $\frac{2^{10}}{2^5} = 2^5 = 32$

$\Rightarrow$ no. of blocks = $\frac{2^{14}}{2^5} = 2^9 = 512$ blks

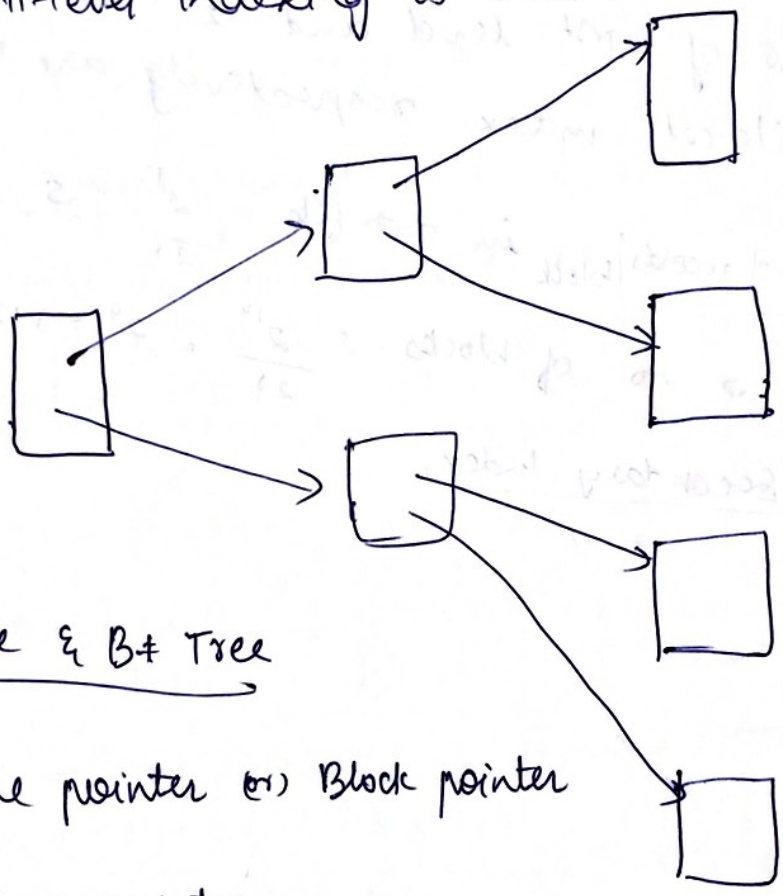
Since secondary index,

$\Rightarrow$ n.

Introduction to B trees and B+ Trees

→ Generalisation of multi-level indexing

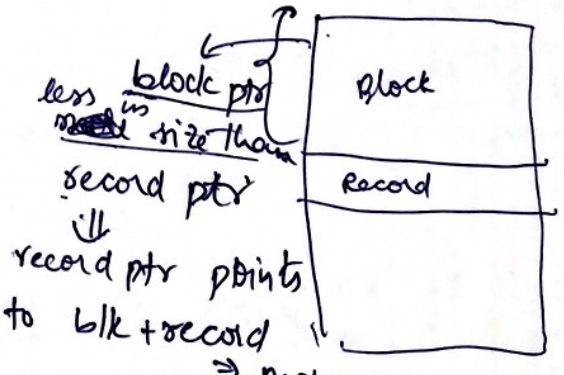
→ Dynamic (they can grow & shrink) whereas multi-level indexing is static



B-Tree & B+ Tree

i) Node pointer (or) Block pointer

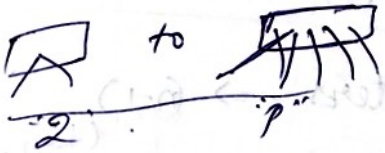
ii) Record pointer

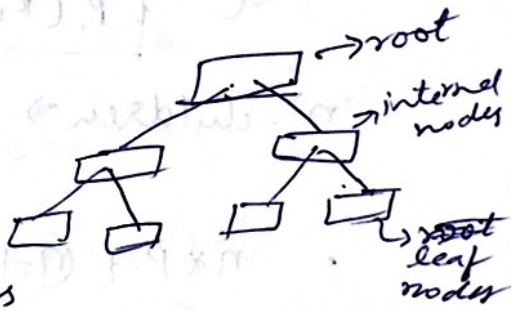


Ex:- if
Blk ptr = 6B
Record ptr = 7B

Properties of B-Trees:-

i) **Root:-**
should have children b/w "2" and "p" children

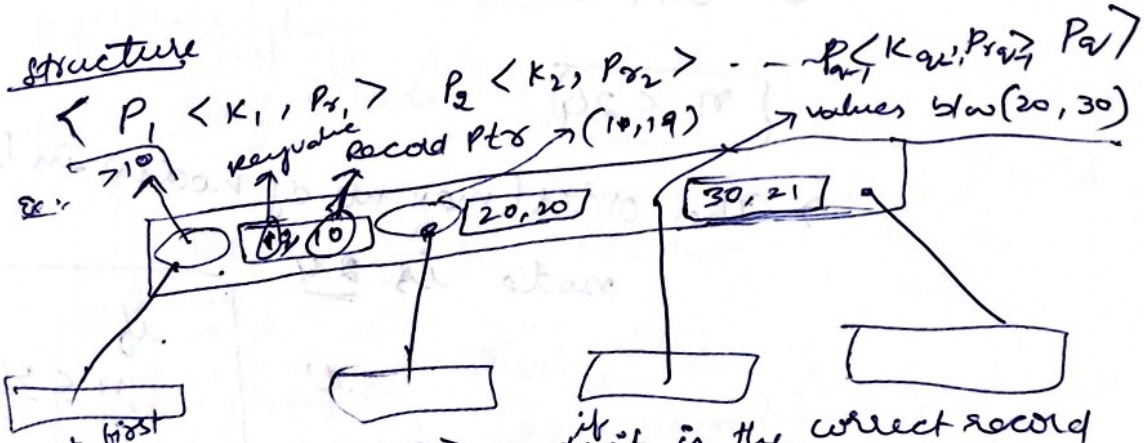




2) **Internal Nodes:-**

* I.N can store upto "p-1" keys
2. have b/w $\lceil \frac{p}{2} \rceil$ and "p" children

Ex:- structure



1) At first checks $< K_1, P_1 >$ and if it is the correct record then traverses to the Record pointed by Record ptr
else checks if

Leaf Node:-

- can store between $\lceil (P-1)/2 \rceil$ and "P-1" keys
- all leaves are at same depth

Q In a B tree, suppose search key is 9 bytes long, disk block size is 512 Bytes, record pointer is 7B, block pointer is 6B then calculate the order of B-tree node.

Sol:- Blk $\{ P_1(K_1, P_1), P_2(K_2, P_2) - - - \ldots P_{n-1}(K_{n-1}, P_{n-1}) P_n \}$

'n' - children $\Rightarrow$ n - Blk pointers $\Rightarrow$ (n-1) (Key, Record Ptr)

- $n * P + (n-1)(K + RPtr) \leq$ Block size
- $= n*6 + (n-1)(9+7) \leq 512$
- $= -6n + 16n - 16 \leq 512 \Rightarrow 22n \leq 528$

$n \leq 24$

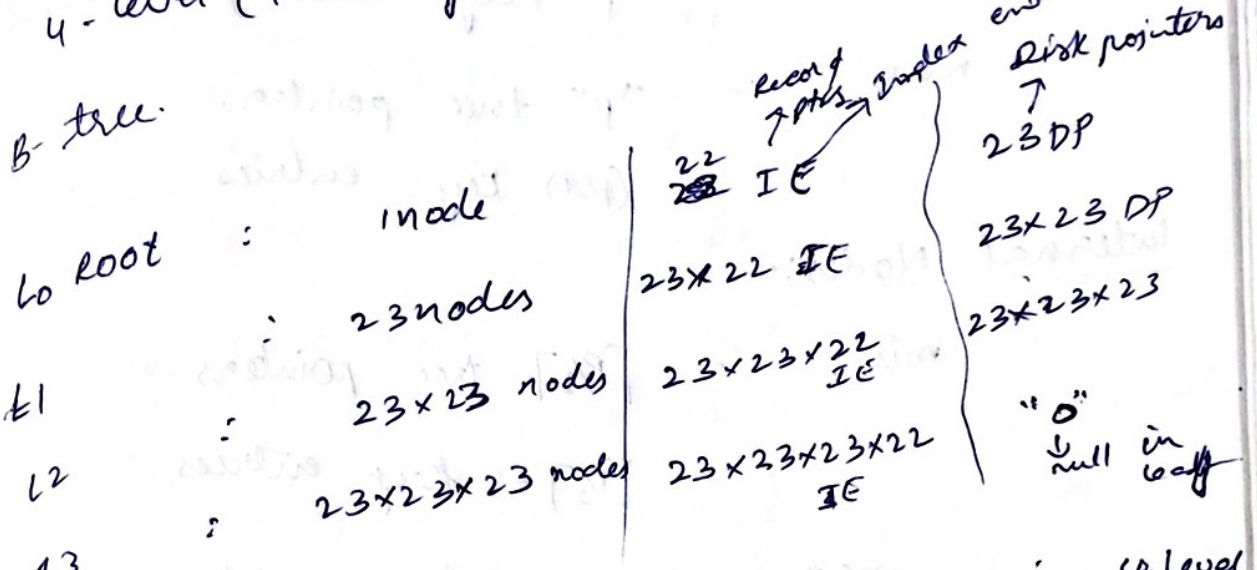
$\Rightarrow$ The order (max no of nodes) in B-Tree node is 24

if $n \leq \frac{530}{23}$

≈ 23.04

$\Rightarrow \lfloor \cdot \rfloor \Rightarrow 23$ (floor)

Suppose that the order of a B-tree is 23. Then how many index records will be stored in a 4-level (including root as 1 level) across the B-tree.



⇒ The max no. of index entries in 4-level B tree where order = 23 ⇒

$$(22 + 23\times22 + 23\times23\times22 + 23^3\times22)$$

$$= 22(1+23+23^2+23^3)$$

$$= 22(23^4-1) = 6156480$$

→ B+ trees have index pointers only in the leaf Nodes.

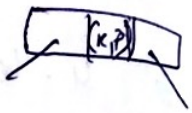
Overflow & Underflow In B Trees

- Insert - Overflow (node full)
- Delete - Underflow (node doesn't have what required to delete)

order - B-tree - "p"

Root:-

min - 2 tree pointers
1 key entry



max - "p" tree pointers
(p-1) key entries

Internal Nodes:-

min - $\lceil p/2 \rceil$ tree pointers
$\lceil p/2 \rceil - 1$ key entries

max - p - tree pointers
(p-1) - key entries

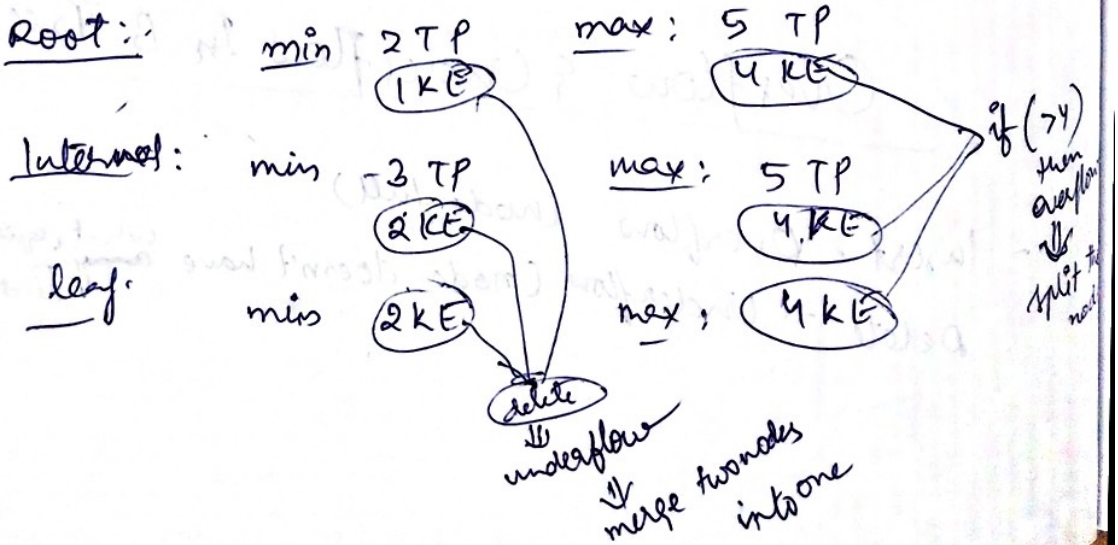
Leaf Nodes:-

min - $\lceil p/2 \rceil - 1$ key entries

max - (p-1)

Ex:-

Order of B-tree is "5": Calculate min & max in each type of node of tree.



Insert the following keys in B-tree of order-4

keys: 2, 5, 10, 1, 6, 9, 4, 3, 12, 18, 20, 25

sol:

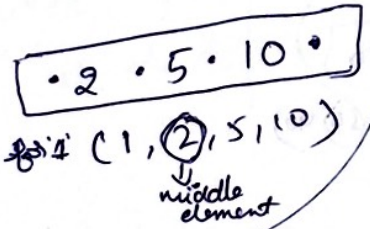
Root:
min: 2 TP, 1 KE
max: 4 TP, 3 KE

Internal nodes
min: 2 TP, 1 KE
max: 4 TP, 3 KE

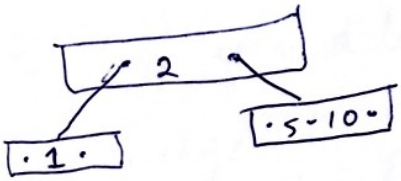
Leaf Nodes
min: 1 KE
max: 3 KE

sol: ● condns:- If overflow occurs break the node into two & take the middle key to the top
If underflow occurs combine the two nodes & make it to one & top

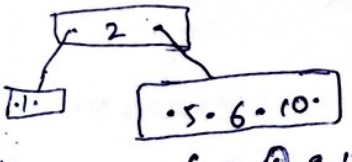
steps:- ①



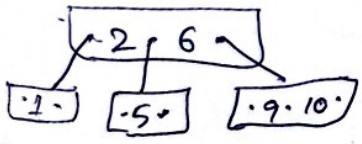
②



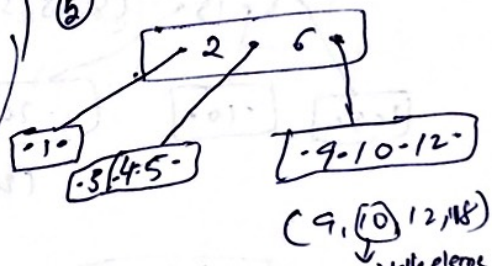
③



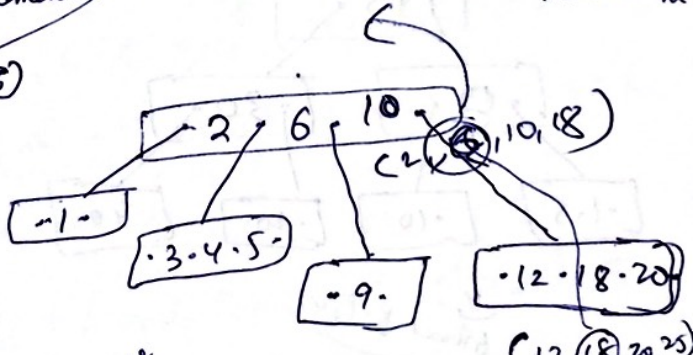
④



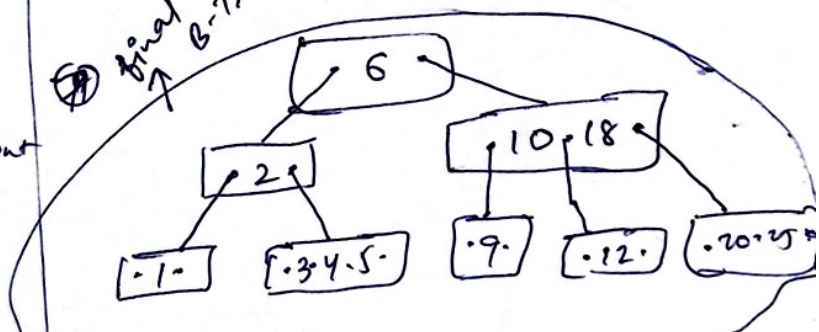
⑤



⑥



⑦ final B-Tree



Q Construct a B-Tree of order '3', for the keys

20, 15, 10, 5, 8, 30, 1, 40.

max: 3TP
2KE

Root: min: 2TP
1KE

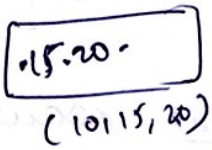
max: 3TP
2KE

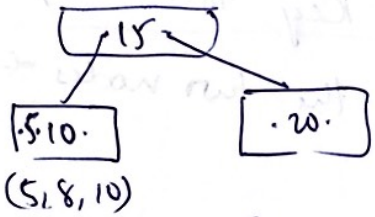
Internal node: min: 2TP
$\lceil \frac{p}{2} \rceil - 1$ 1KE

2KE

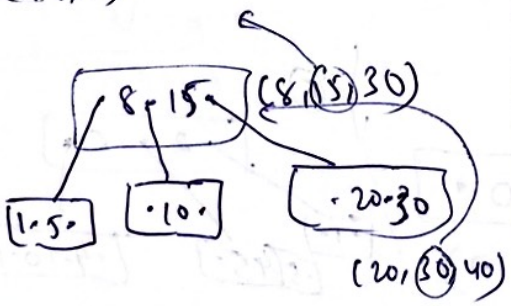
Leaf Node: min: 1KE

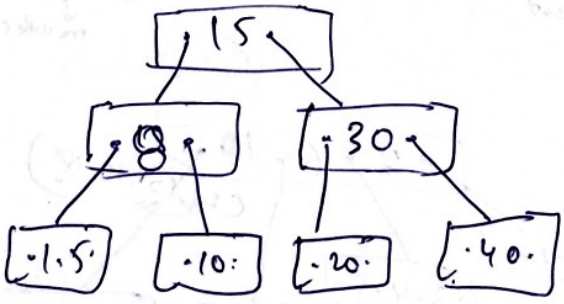
①





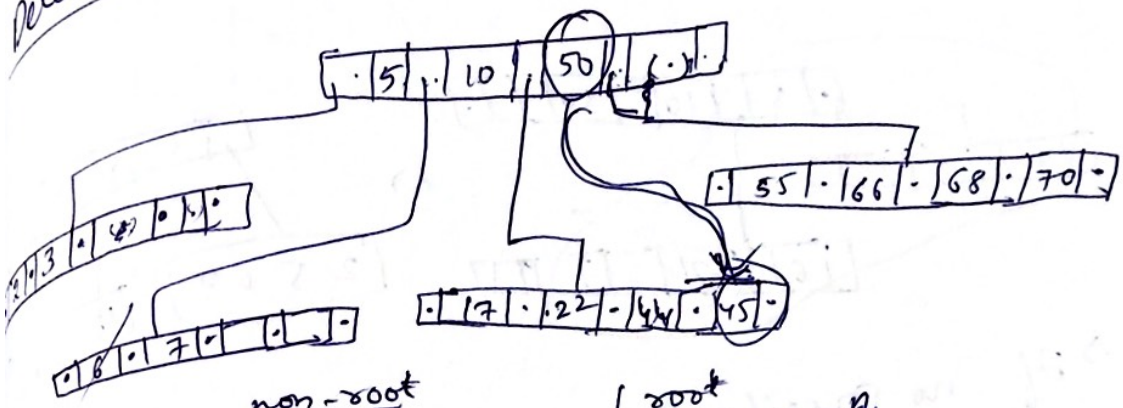
②





Final B-Tree

Deletion from a B-tree



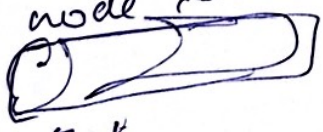
p = 5 ;

non-root
max keys = 4
min keys = 2

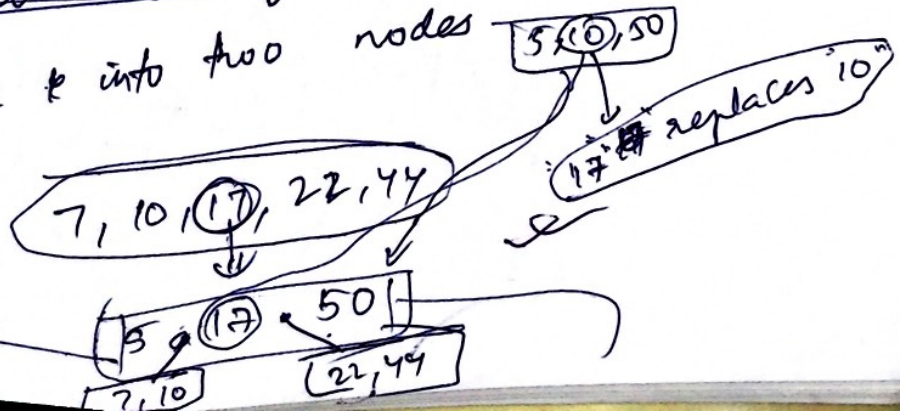
root
max keys = 4
min keys = 1

1) In B-trees deletion should occur at the leaf level
2. If not we will make it to occur at the leaf level & delete it and check for the underflow
3. like for '50', if we want to delete, we take either inorder predecessor (or) in-order successor replace it and delete it & check for underflow.

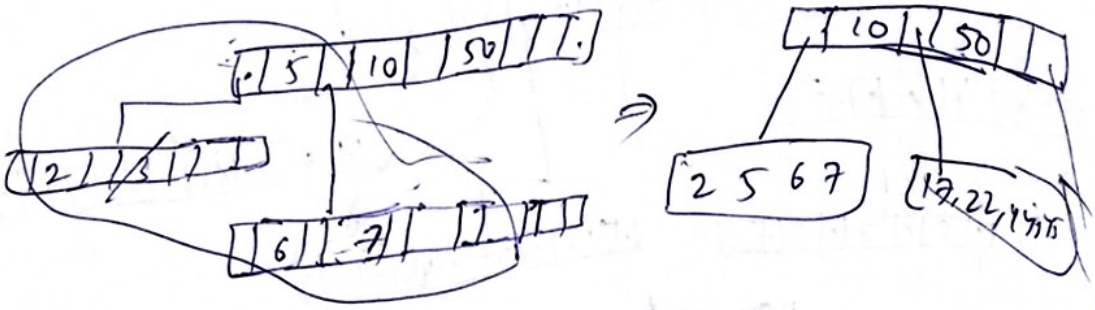
ex:- if you delete "6", we get underflow, so, we merge the node which has underflow with



most populous sibling (only on both sides) & re-arrange & into two nodes

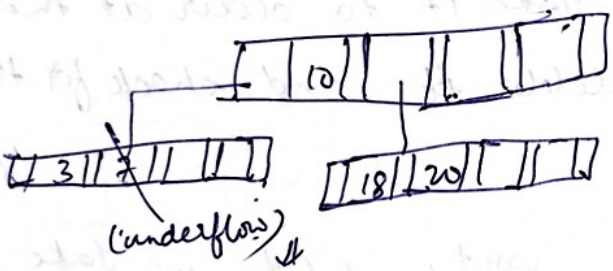


2) when you are deleting '3'

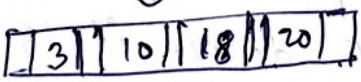


⇒ If no overflow occurs, then remove key in parent & merge into a single node

Ex:



(3, 10, 18, 20) ⇒ underflow in root ⇒ no need of another levels



B+ Trees

File Organisation

Database
|
Files
|
Records
|
Fields

Block

Db is a collection of files
files " " " of Records
Records " " " of attributes

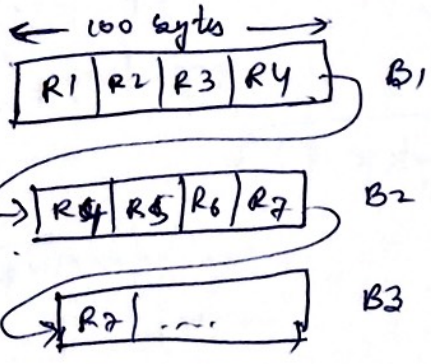
Blocking factor:

Average no of records, that can be kept in a block (per block)

Strategies to store file into blocks: (of records)

1) Spanned Strategy:

It allows partial part of record can be stored in a block



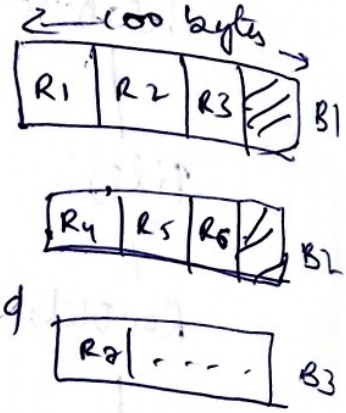
adv: no wastage of mem

disadv: Block accen increases

Suitable: variable length record

Unspanned strategy :-

entire record in a single block



disadv: wastage of mem

adv: reduces block access

suitable :- for fixed length record

Organisation of records in a file :-

Ordered file Organisation

Continuation after txn & concurrency ctrl