Part C

1. Problem statement for library management system.
Design UML Class diagram
Explain its components.

⇒ **Problem statement :-**

→ Purpose : Monitoring and controlling transactions in a library.

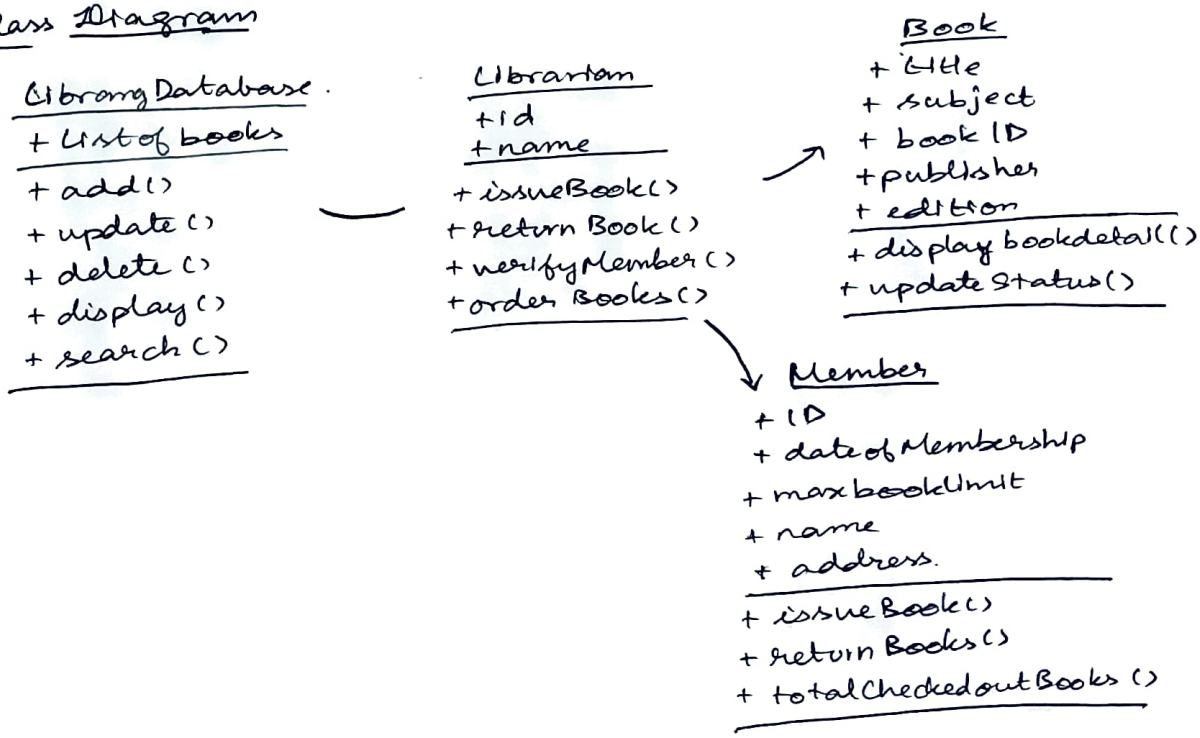
→ Need to maintain :- Details of new books & books lent.

→ Basic operations :- Adding new member
new books
searching books & members
facility to borrow & return books.

→ Well thought out, attractive interface
with strong searching, inserting & reporting capabilities.

→ Report generation facility help keep record of
books borrowed.

Class Diagram



Components.

→ Upper section ⇒ Library Database, Librarian, Book, Member.

→ Middle section ⇒ + List of Books, ID, name, title, etc.

→ Lower section ⇒ add(), update(), delete(), etc.

[Write a story explaining the correlation
of the three sections].

3. Program for Computers (Refer Qbank for Full questions).

⇒

```
#include<iostream>
using namespace std;
class computers
{ public: int hrs, mins, secs, no;
 public: void getDetail()
      { cout<<"Enter total no. of computers:";
        cin>>no;
        cout<<"Enter the usage in hrs:";
        cin>>hrs; }.
 public: void calculatesecondsperDay()
      { int secspd;
        secspd = hrs*3600;
        cout<<"Usage in seconds per day:";
        cout<<secspd<<"secs"; }
 public: void calculateminutesperWeek()
      { int minpw;
        minpw = hrs*60*7;
        cout<<"Usage in minutes per week:";
        cout<<minpw<<"mins"; }
 public: void calculatehourperMonth()
      { int hrspm;
        hrspm = hrs*30;
        cout<<"Usage in hours per month:";
        cout<<hrspm<<"hrs"; }
 public: void calculateperYear()
      { float dayspy;
        dayspy = (hrs*365)/24.00;
        cout<<"Usage in days per year:";
        cout<<dayspy<<"days"; }};

int main()
 { computers CM;
  CM.getDetail();
  CM.calculatesecondperDay();
  CM.calculateminutesperWeek();
  CM.calculatehourperMonth();
  CM.calculateperYear();
  return 0; }.
```

4. Example for cast.
Explain OOPS features with suitable examples.

=> Cast :- special operator that forces one data type to be converted into another.

example:-

Implicit casting =>
```
short a = 2000;
Int b;
b = a;
```

features of OOPS.

→ Class :- • User-defined data type, main building block of OOP.
• It holds data members & member functions in single unit.
• Blueprint of an object

ex:
```
public class Student
{ Int age;
  Int marks;
  Int rollno;
  {
    }}
```

→ Objects :- • It is an instance of a class.
• Memory isn't allocated without object.
• Helps access data members & member functions.

ex:
```
Student obj = new Student();
Student obj1 = new Student();
Student obj2 = new Student();
```

→ Inheritance :- • Ability to inherit properties of one class to another.
• Makes code reusable.
• Easy to add new features/methods.
• Provides overriding feature.

Base class
↓
sub-class.

ex:
```
Class Student
  {{
    }};
class Name : public Student
{
  };
```

→ Polymorphism :- • One name, many forms.
• Can be classified as static & dynamic

ex: + → performs addition.
+ → performs concatenation. (string).

Data abstraction:-
- Hiding background details, providing essential details.
- Avoids code duplication & increases reusability.

ex:)
```
class A
{ private:
    int x;
  void set(int x)
   { a = x; }
  void display()
   { cout<<a; }};
int main()
{ A obj;
 obj.set(10);
 obj.display();
 return 0; }
```

Encapsulation:-
- Binding data into single unit.
- Restricts properties/methods from outside access.
- Uses access modifiers ⇒ private, public, protected.

ex:)
```
Class A
{ private:        ——→ In class
    int x;
  public:        ——→ any class
    int y;
  protected:     ——→ hierarchial class.
    int z; }
```

6. Explain in detail about ~~destructor~~ constructor & demonstrate with example.

→ **Constructor** :- Special type of member function of a class which initializes objects of a class.
- It is automatically called when object is created.
- Doesn't have any return type.

Constructor example

```
class Student
{ public:
    Student()  ————————> constructor.
  { cout<< "Student"; }};
int main()
  Student obj; ——————> object will call the constructor.
   return 0; }.
```

There are three types of constructors.

(i) **Default constructor** :- • Doesn't take any argument.
• No parameters.
ex:) Student(). [for class Student()].

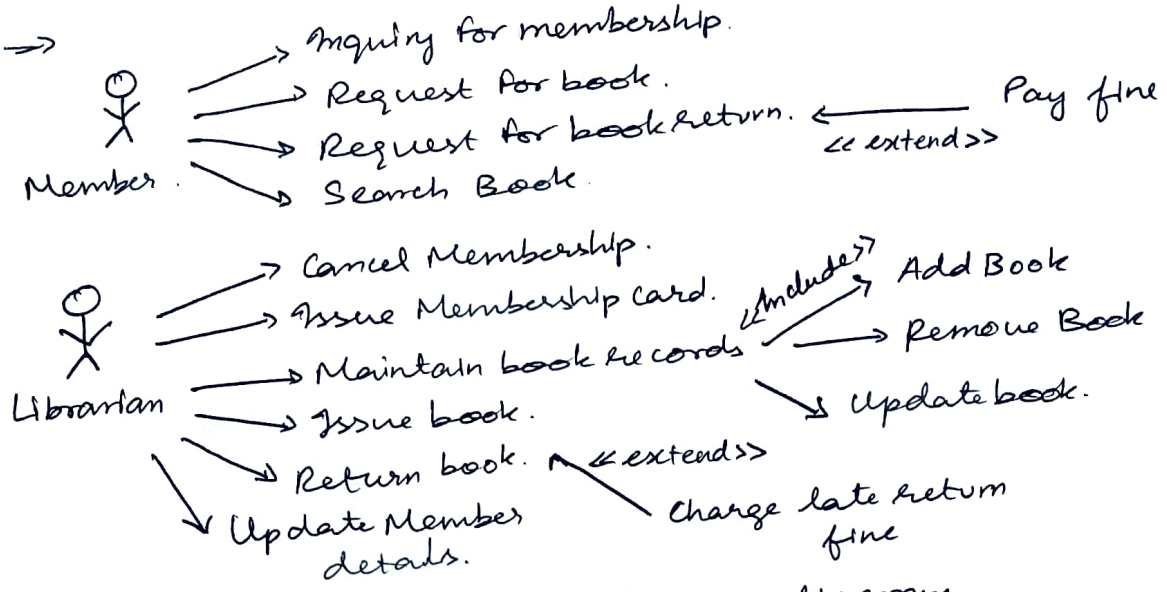
(ii) **Parameterized constructor** :- • Arguments are passed in constructor.
• These arguments help initialize an object when it is created.
ex:) Student(int x, int y) [for class Student()]

→ It is used to overload constructors.
Types :- Implicit → Example e = Example(0,50);
& Explicit → Example e (0,50);

(iii) **Copy constructor** :- • Initializes an object using another object of same class.

syntax) Classname(const classname & objectname)
{ ... }.

10. Library use case diagram.



11. Benefits and concepts of Use case diagram.

=> Use case diagram
→ Summarizes details of system's users (actors) and their interactions with system.

Helps in
→ Scenarios of interactions.
→ Goals system helps actor achieve.
→ Scope of system.

Components

=> Actors => Users that interact with system.
→ Can be person or organization.
→ They must be external objects that produce / consume data

=> System => Specific sequence of actions / interaction between actor and system.
→ Also called scenario.

=> Goal => End result of most use cases.

Benefits

=> They are traceable.
=> Easily understandable.
=> Serve as basis for estimating, scheduling & validating effort.
=> Capture additional behaviours that help improve robustness of system.

13. Various Relationships in Use case Diagram.

→ Five relationship types.

(i) → Association between actor & use case.
(ii) → Generalization of an actor
(iii) → Extend between 2 use cases
(iv) → Include between 2 use cases
(v) → Generalization of a use case.

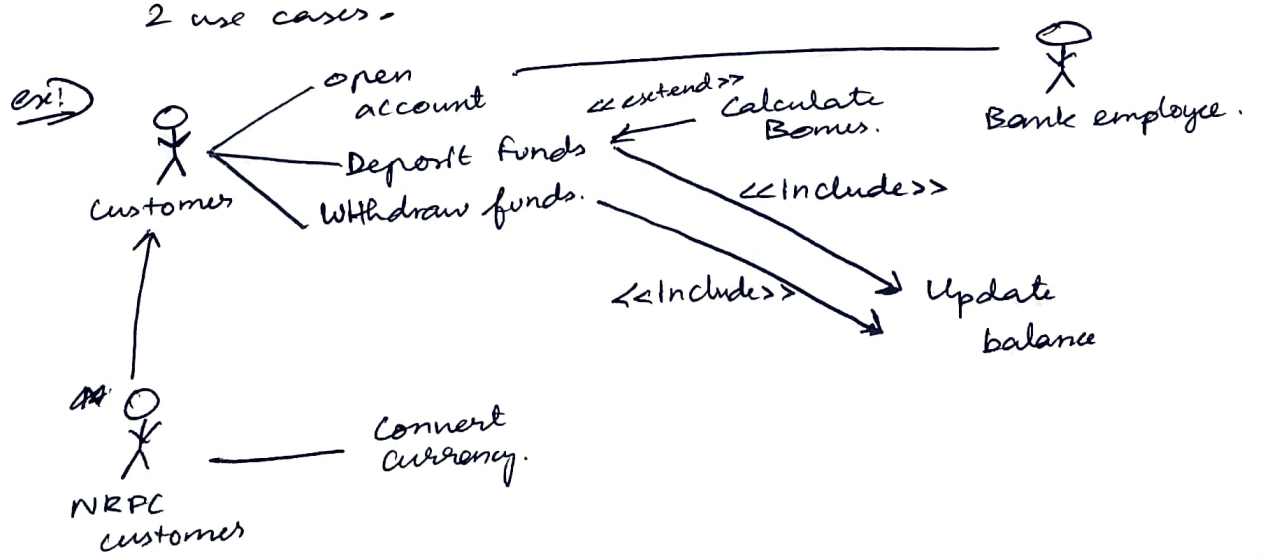
(i)
→ Actor must be associated with at least one use case.
→ Actor can be associated with multiple use cases
→ Multiple actors can be associated with a single use case.

(ii)
→ One actor can inherit role of other actor.
→ Descendant inherits all the use cases of ancestor.

(iii) → It extends base use case & adds more functionality to system.
→ It is dependent on the extended (base) use case.
→ It is usually optional.
→ Must be meaningful on its own.

(iv) → Shows that the behaviour of the included use case is part of the including (base) use case.
→ Mainly used to reuse common actions across multiple use cases.
→ Also, used to simplify complex behaviours.
→ Base use case is incomplete without included use case.
→ Included use case is mandatory.

(v) → Behaviour of ancestor is inherited by descendant.
→ Used when there is common behavior between 2 use cases.



14. Various UML diagrams & purposes.

→ Class Diagram → Shows classes in a system, attributes & operations of each class & the relationship between classes.

→ Component Diagram → Structural relationship of components of a software system

→ Deployment Diagram → Shows hardware of system & software in hardware.

→ Object Diagram → Use real world examples, shows relationship between objects.

→ Package Diagram → Dependencies between different packages in a system

→ Profile Diagram → Helps extend & customize UML by adding new building blocks.

→ Composite structure Diagram → Shows internal structure of a class

→ Use case Diagram → Graphic overview of actors, and their needs as functions & their interactions

→ Activity Diagram → Workflow in a graphic way.

→ State Machine Diagram → Describe behaviour of objects that in accordance to current state.

→ Sequence Diagram → Shows object interactions & the order those interactions occur.

→ Interaction Overview Diagram → shows a series of interaction diagrams.

→ Timing Diagram → Behaviour of objects in a given time frame.

→ Communication Diagram → Similar to sequence diagram, focus on messages passed between objects.