**Ahmednagar Jilha Maratha Vidya prasarak Samaja's**

**NEW ARTS, COMMERCE AND SCIENCE COLLEGE**
**AHMEDNAGAR**

**DEPARTMENT OF COMPUTER APPLICATIONS**

# F.Y. / S. Y. / T.Y. B. C. A. (Science)

**Roll No.** ______

# LAB Practical Workbook

Savitribai Phule Pune University

# Section-I & II

S. Y. B. C. A. (Science)

SEMESTER III BCA – 305 & BCA - 306

# Lab Course – I & II
# Data Structure and ARDBMS

Workbook

Name: ----------------------------------------

Roll No.:---------------------------------------

Academic Year: ---------------------------

## Editors:

**Section-I: Data Structure**

| | |
|---|---|
| Mr. Satyavan M. Kunjir | Dr. D. Y. Patil ACS College, Pimpri, Pune |
| Mrs. Madhuri A. Darekar | Dr. D. Y. Patil ACS College, Pimpri, Pune |

**Section-II: Advanced Database Management System**

| | |
|---|---|
| Ms. Sujata P. Patil | Dr. D. Y. Patil ACS College, Pimpri, Pune |
| Ms. Vidya H. Bankar | Dr. D. Y. Patil ACS College, Pimpri, Pune |

## Reviewed By:

| | |
|---|---|
| Dr. Ranjit D. Patil | Dr. D. Y. Patil ACS College, Pimpri, Pune |

## Introduction

1. **About the Workbook:**

   This workbook is intended to be used by SYBCA (Science) students for the Data Structure (DS) & Advanced Relational Database Management System (ARDBMS) Assignments in Semester–III. This workbook is designed by considering all the practical concepts / topics mentioned in syllabus.

2. **The objectives of this Workbook are:**

   1) Defining the scope of the course.
   2) To bring the uniformity in the practical conduction and implementation in all colleges affiliated to SPPU.
   3) To have continuous assessment of the course and students.
   4) Providing ready reference for the students during practical implementation.
   5) Provide more options to students so that they can have good practice before facing the examination.
   6) Catering to the demand of slow and fast learners and accordingly providing the practice assignments to them.

3. **How to use this Workbook:**

   The workbook is divided into two sections. Section-I is related to DS assignments and Section-II is related to ARDBMS assignments.
   The Section-I (DS) is divided into eight assignments. Each DS assignment has several SET. It is mandatory for students to complete all the SET in given slot.
   The Section-II (ARDBMS) is divided into six assignments. The assignments comprise of activities to be carried out on given databases. The students have to create database, insert appropriate records and then perform the activities specified in each of the assignments. A pool of databases will get created as student progresses through the assignments and these databases can be repeatedly used in subsequent assignments.
   Each ARDBMS assignment has several SET. It is mandatory for students to complete all the SETs in given slot.

4. **Instructions to the students:**

   Please read the following instructions carefully and follow them.
   - Students are expected to carry this workbook every time they come to the lab for practical.
   - Students should prepare for the assignment by reading the relevant

material which is mentioned in ready reference.

- Instructor will specify which problems to solve in the lab during the

  allotted slot and student should complete them and get verified by the instructor. However, student should spend additional hours in Lab and at home to cover all workbook assignments if needed.
- Students will be assessed for each assignment on a scale from 0 to 5

| | |
|---|---|
| Not done | 0 |
| Incomplete | 1 |
| Late Complete | 2 |
| Needs improvement | 3 |
| Complete | 4 |
| Well Done | 5 |

## 5.Instruction to the Instructors:

- Make sure that students should follow above instructions.
- Explain the assignment and related concepts using white board if required or by demonstrating the software.
- Give specific input to fill the blanks in queries which can vary from student to student.
- Evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
- The value should also be entered on assignment completion page of the respective Lab course.

## 6. Instructions to the Lab administrator:

You have to ensure appropriate hardware and software is made available to each student.

The operating system and software requirements on server side and also client side areas given below:

- Server and Client Side-(Operating System) Fedora Core Linux/Windows
- Database server – PostgreSQL
- Turbo C

**Table of Contents for Section-I**

## Assignment Completion Sheet

| | | | |
|---|---|---|---|
| Lab Course I | | | |
| Data Structure | | | |
| **Sr. No** | **Assignment Name** | **Marks (Out of 5)** | **Teachers Sign** |
| 1 | Sorting Techniques (Non Recursive) | | |
| 2 | Sorting Techniques (Recursive) | | |
| 3 | Searching Techniques | | |
| 4 | Linked List | | |
| 5 | Stack | | |
| 6 | Queue | | |
| 7 | Trees | | |
| 8 | Graph | | |
| Total (Out of 40 ) | | | |
| Total (Out of 10) | | | |

AHMEDNAGAR JILHA MARATHA VIDYA PRASARAK SAMAJ'S

**NEW ARTS, COMMERCE AND SCIENCE COLLEGE**

## Department of Computer Applications (BCA Science)

# Certificate

This is to certify that Mr. /Ms. ________________________________________

has successfully completed the course work for Lab Course I:BCA:305:Data Structure and has scored ______ Marks out of 30.

**Instructor** **H.O.D / Coordinator**

**Internal Examiner** **External Examiner**

# Assignment No : 1
# Sorting Techniques (Non Recursive)

**SORTING**

- Sorting means arranging a set of data in some given order or ordering a list of items.

  ‘Or’

  Sorting is a process of ordering a list of elements in either ascending or descending order.
- List is a collection of record each contains one or more fields. The field which contains unique value for each record is called key field.
- Definition:-

  Sorting is the operation of arranging the records of a table according to the key value of each record

  e.g. consider a telephone directory which consists of 4 field phone number, name, address, pin code .

  So a large data is maintained in the form of records. If we want to search a phone no and name it should be alphabetically sorted then we can search easily. It would be very difficult if records were unsorted.
- The sorting algorithm are divided into two categories

1) Internal sorting-

   Sorting is done on data which is sorted in main memory.
2) External sorting –

   Sorting is done on data which is stored on auxiliary storage device.

   e.g. hard disk, floppy, tape etc.

- **BUBBLE SORT**
  - This is one of the simplest and most popular sorting methods. The basic idea is to pass through the file sequentially several times.
  - In each pass we compare successive pairs of elements(x[i] with x[i+1]) and interchange the two if they are not in the required order.
  - One element is placed in its correct position in each pass.
  - In first pass, the largest element will sink to the bottom, second largest in the second pass and so on. Thus a total of n-1 passes are required to sort n keys
  - Efficiency of bubble sort

  Worst case = average case = best case complexity =O($n^2$)

- **INSERTION SORT**
  - Insertion sort inserts each item into its proper place in the final list
  - In this the first iteration starts with comparison of 1st element with 0th
  - In second iteration 2nd element is compared with the 0th and 1st element and so on.
  - In every iteration an element is compared with all elements

  - The basic idea of this method is to place an unsorted element into its correct position in a growing sorted list of data. We select one element from the unsorted data at a time and insert it into its correct position in the sorted set.
  - E.g. in order to arrange playing cards we pick one card at a time and insert this card hold in the hand.

- Efficiency of insertion sort
  Best case = O(n)
  Worst case = average case = O($n^2$)

**SET A:**

1) Write a C program to accept and sort n elements in ascending order by using bubble sort.
2) Write a C program to accept and sort n elements in ascending order by using insertion sort.

**SET B:**

1) Write a C program to read the data from the file "employee.txt" which contains empno and empname and sort the data on names alphabetically (use strcmp) using Bubble Sort.
2) Write a C program to read the data from the file "person.txt" which contains personno and personage and sort the data on age in ascending order using insertion Sort.

**SET C:**

1) Write a C program to sort a random array of n integers (value of n accepted from user) by using Bubble Sort algorithm in ascending order
2) Write a C program to sort a random array of n integers (value of n is accepted from user) by using Insertion Sort algorithm in ascending order .
3) Add the code to Print Time complexity for SET A programs.
4) Write a C program to sort the elements by initializing the array (e.g int A[5] = {10, 20, 35, 23, 12}) using bubble sort.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No : 2
# Sorting Techniques (Recursive)

**QUICK SORT**

- It is also called "partition Exchange sort. The strategy used here is "divide and conquer" i.e we successively partition the list in smaller lists and apply the same procedure to the sub-list. The procedure is as follows:-

**Procedure –**

- We will consider one element at a time (pivot) and place it in its correct position.
- The pivot is placed in a position such that all elements to the left of the pivot are less than the pivot and all elements to the right are greater.
- The array is partitioned into two parts:- left partition and right partition.
- The same method is applied for each of the partition.
- The process continues till no more partition can be made. We shall be considering the first element of the partition as the pivot element.

**Algorithm:**

```
Step 1: start.
Step 2: A is an array of n element.
Step 3:  lb=0           lb = lower bound
         ub = n-1       ub = upper bound.
Step 4: if(lb<ub)
         i.e. if the array can be partitioned
         j=partition(A,lb,ub)    // j is the pivot position

         quicksort(A,lb,j-1);
         quicksort(A,j+1,ub);
```

- Now, we must write the function to partition the array. There are many methods to do the partitioning depending upon which element is chosen as the pivot.
- We will be selecting the first element as the pivot element and do the partitioning accordingly.
- We shall choose the first element of the sub- array as the pivot and find its correct position in the sub- array.
- We will be using two variables down and up for moving down and up array.

**Algorithm for partitioning**

```
Step 1: down=lb+1
Step 2: up=ub
Step 3:  pivot=A[lb]
Step 4:  perform step 5 to 7 as long as down<up else go to step 8.
Step 5:  while (A[down]<=pivot && down<ub)          down++;
Step 6:  while (A[up]>pivot)                            up--;
Step 7:  if (down<up)      interchange A[down] and A[up]
Step 8:  interchange A[up] and pivot,      j=up,    i.e.  pivot position=up
Step 9:  return up
```

Step 10: stop.

- In this algorithm we want to find the position of pivot i.e. A[lb].
- We use two pointers up and down initialized to the first and last elements respectively.
- We repeatedly increase down as long as the element is < pivot.
- We repeatedly decrease up as long as the element is > pivot.
- If up and down cross each other i.e. up<=down, the correct position of the pivot is up and A[up and pivot are interchanged.
- If up and down do not cross A[up] and A[down] are interchanged and process is repeated till they do not cross or coincide.
- Efficiency of quick sort.
  Best case = average case = O(nlogn)
  Worst case = O($n^2$)

**MERGE SORT.**

- Merging is the process of combining two or more sorted data lists into a third list such that it is also sorted.
- Merge sort follows Divide and Conquer strategy.
  - Divide :- divide an n element sequence into n/2 subsequence.
  - Conquer :- sort the two sequences recursively.
  - Combine :- merge the two sorted sequence into a single sequence.
- In this two list are compared and the smallest element is stored in the third array.

**<u>Algorithm</u>**:-

```
Step 1: start
Step 2: initially the data is considered as a single array of n element .
Step 3: divide the array into n/2 sub-array each of length $2^i$ (I is 0 for 0th iteration). i.e. array is divided into n sub-arrays each of 1 element.
Step 4: merge two consecutive pairs of sub-arrays such that the resulting sub-array is also sorted.
Step 5: The sub-array having no pairs is carried a sit is
Step 6: step 3 and 4 are repeated till there is only one sub-array remaining of size n.
Step 7: stop.
```

**SET A:**

1) Write a C program to accept and sort n elements in ascending order by using Quick sort.
2) Write a C program to sort a random array of n integers (value of n accepted from user) by using Quick Sort algorithm in ascending order

**SET B:**

1) Write a C program to accept and sort n elements in ascending order by using Merge sort.
2) Write a C program to sort a random array of n integers (value of n is accepted from user) by using Merge Sort algorithm in ascending order.

**SET C :**

1) Add the code in SET A (Q1) and SET B (Q1) to Print Time complexity for Quick sort and Merge sort.
2) Write a C program to sort the elements by initializing the array (e.g. int A[5] = {11, 12,15, 16, 17}) using Merge sort .
3) Write a C program to sort the elements by initializing the array (e.g. int A[5] = {11, 12,15, 16, 17}) using Quick Sort .

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

## Assignment No : 3
## Searching Techniques

The commonly used searching methods used are linear search and Binary search

**SET A:**

1) Write a C program to accept n elements from user store it in an array. Accept a value from the user and use linear/Sequential search method to check whether the value is present in array or not. Display proper message.
2) Write a C program to accept n elements from user store it in an array. Accept a value from the user and use recursive binary search method to check whether the value is present in array or not. Display proper message.

**SET B:**

1) Add the code to print the number of Comparisons for SET A programs
2) Write a C program to accept n student names from user store it in an array. Accept a student name from the user and use linear/Sequential search method to check whether the student name is present in array or not. Display proper message.

**SET C:**

1) Write a C program to read the data from file 'cities.txt' containing names of 10 cities and their STD codes. Accept a name of the city from user and use linear search algorithm to check whether the name is present in the file and output the STD code, otherwise output “city not in the list”.
2) Write a C program to accept n elements from user store it in an array. Accept a value from the user and use Non- recursive binary search method to check whether the value is present in array or not. Display proper message.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No : 4
# Linked List

- **Linked list:-**

A linked list is an ordered collection of items which is dynamic in nature i.e. its size varies and each item is 'linked" or connected to another item. It is a linear collection of data elements called nodes.

- **LINKED LIST IMPLEMENTATION-:**

  A linked list may be implemented in two ways:
  1) Static representation
  2) Dynamic representation.

**1) Static representation:-**

An array is used to store the elements of the list. The elements may not be stored in a sequential order. The correct order can be stored in another array called "link"
The values in this array are pointers to elements in the disk array.

Data array

| | |
|---|---|
| 0 | Blue |
| 1 | Red |
| 2 | |
| 3 | Violet |
| 4 | Green |
| 5 | |
| 6 | Orange |

Link array

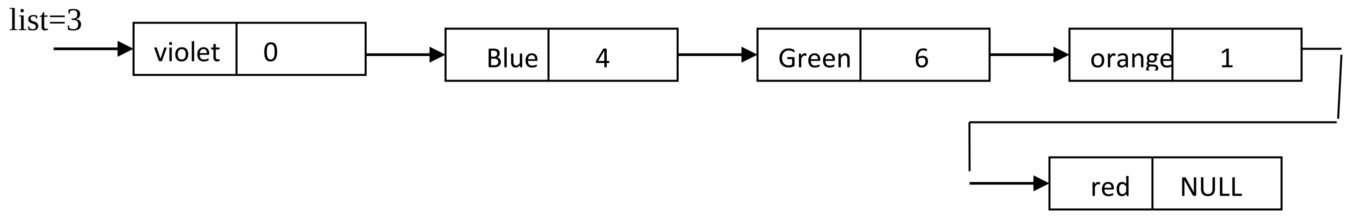
| | |
|---|---|
| 0 | 4 |
| 1 | -1 |
| 2 | |
| 3 | 0 |
| 4 | 6 |
| 5 | |
| 6 | 1 |

| | |
|---|---|
| Data[3] = violet | Link[3] = 0 |
| Data[0] = Blue | Link[0] = 4 |
| Data[4] = Green | Link[4] = 6 |
| Data[6] = Orange | Link[6] = 1 |
| Data[1] = Red | Link[1] = -1 list end |

2) **Dynamic Representation:-**

- The static representation uses arrays which is a static data structure and has its own limitations.
- A linked list is a dynamic data structure i.e. the size should increase and when elements are deleted, its size should decrease.
- This cannot be possible using an array which uses static memory allocation i.e. memory is allocated during compile time. Hence we have to use "dynamic memory allocation" where memory can be allocated and de-allocated during run-time.

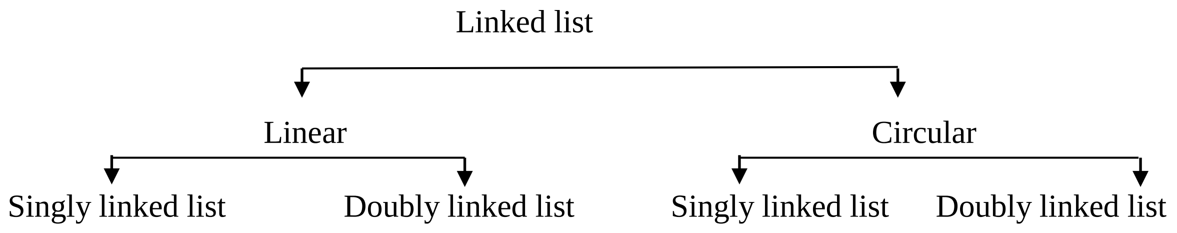
- Another way of storing a list in memory is by dynamically allocating memory for each node and linking them by means of pointers since each node will be at random memory location. We will need a pointer to store the address of the first node.



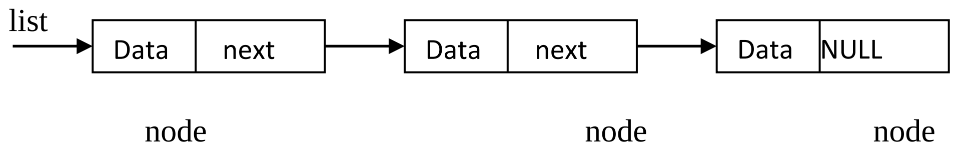
List is an external pointer which stores the address of the first node of the list.

- **TYPES OF LINKED LIST**

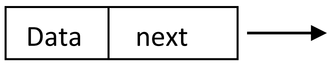
1) Singly Linked list
2) Circular linked list
3) Doubly linked list
4) Circular doubly linked list

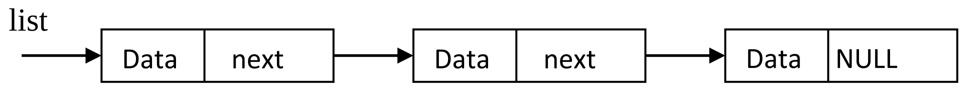


1) **Linear linked list**
   In this list the elements are organized in a linear fashion and list terminates at some point i.e. the last node contains a NULL pointer.



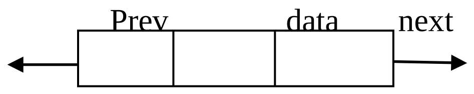
- Singly linked list-
  Each node in this list contains only one pointer which points to the next node.

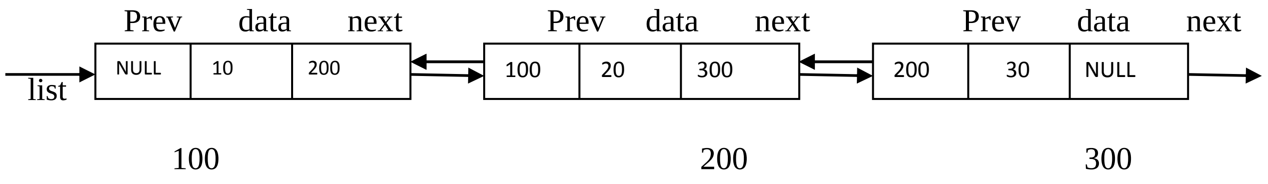




➢ Doubly linked list

Each node in this contains two pointers, one pointing to the previous node and the other pointing to the next node. This list is used when traversing in both directions is required.

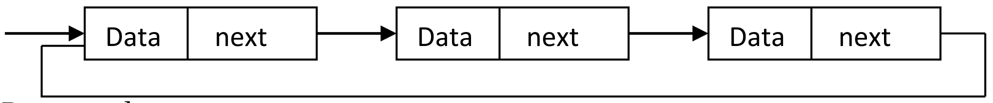


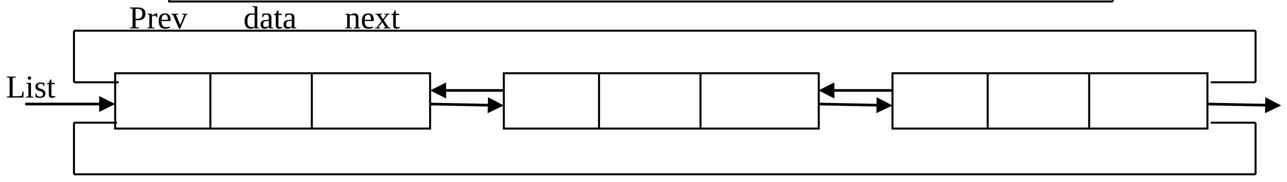


2) Circular list

In this list, the last node does not contain a NULL pointer but points back to the first node i.e. it contains the address of the first node. Each of these lists can be either a singly linked or a doubly list.

Circular singly linked list





Circular doubly linked list

- **OPERATIONS ON A LIST**

  The following are some of the basic list operations:-

1) Traversing a list:-
   Visiting each node of the list is called traversal
2) Insertion:-
   A node can be inserted at the beginning, end or in between two nodes of the list.
3) Deletion:-
   Deletion from a list may be done either position-wise or element-wise
4) Display:-
   Display each element of the list.
5) Searching:-
   This process searches for a specific element in the list.
6) Reversing or inversion:-
   This process reverses the order of nodes in the list
7) Concatenation:-
   This process appends the nodes of the second list at the end of the first list i.e. it joins two lists.
8) Computation of length:-
   Count the total no. of nodes in the list

9) Creating a linked list
10) Intersection, union, difference.

**SET A:**

1) Write a C program to implement a singly linked list with Create and Display operation.
2) Write a C program to implement a Circular Singly linked list with Create and Display operation.
3) Write a C program to implement a doubly linked list with Create and Display operation.

**SET B:**

1) Implement the following programs by adding the functions one by one in SET A (for all questions)

   i) Add the function for Append operation.
   ii) Add the function for Search operation.
   iii) Add the function to insert node at the beginning.
   iv) Add the function to insert node at the end.
   v) Add the function to insert node in the list by accepting the value in the specified linked list. (Note: accept a number; search the number in the list, then insert the new node after that searched node)
   vi) Add the function to insert node in the list by accepting the position. (Note: accept the Position, and then insert the new node in that particular position.)
   vii) Add the function to find length of linked list

**SET C:**

1) Implement the following programs by adding the functions one by one in SET A (for Q1 and Q3)

   i) Add the function to delete the first node.
   ii) Add the function to delete the last node.
   iii) Add the function to delete the node by accepting the value. (Note: accept the number, search the number in the list, then delete that node from the linked list)
   iv) Add the function to delete the node by accepting the position. (Note: accept the position, search the node at that position in the list and then delete the node from that position.)

2) Write a C program to add two polynomial.
3) Write a C program to find intersection of two linked list.
3) Write a C program to reverse a singly linked list.
4) Write a C program to display the data of list in reverse.
5) Write a C program to implement static linear linked list using array for following operation
   - Append() , Insert(), Display()

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No : 5:    Stack

A stack is an ordered collection of items into which items may be inserted and deleted from one end called the top of the stack.
The stack operates in a LIFO (last in first out) manner. i.e. the element which is put in last is the first to come out. That means it is possible to remove elements from stack in reverse order from the insertion of elements into the stack.
The real life e.g. of the stack are stack of coins, stack of dishes etc. only the topmost plate can be taken and any new plates has to be put at the top.

- **PRIMITIVE OPERATIONS ON A STACK.**

1) Create
2) Push
3) Pop
4) Isempty
5) Isfull
6) Peek

- **STACK IMPLEMENTATION:-**

The stack implementation can be done in two ways:-

**1) Static implementation:-**

- It can be achieved using arrays. Though it is very simple method it has few limitations.
- Once a size of an array is declared, its size cannot be modified during program execution.
- The vacant space of stack also occupies memory space.
- In both cases, if we store less argument than declared, memory is wasted and if we want to store more elements than declared array cannot be expanded. It is suitable only when we exactly know the number of elements to be stored.
- **Operations on static stack:-**

1) Declaring of a stack :-
A stack to be implemented using an array will require.
- An array of a fixed size.
- An integer called top which stored the index or position of the topmost element.

We can use a structure for the above purpose.
2) Creating a stack:-
This declaration only specifies the template. The actual stack can be declared as-
STACK s1;
3) initialize a stack:-
When a stack variable is declared the integer top has to be initialized to indicate an empty stack. Since we are using an array the first element will occupy position 0. Hence to indicate an empty stack top has to be initialized to -1
4) Checking whether stack is empty:-
An empty stack can be tested from the value contained in top. If top contains -1 it indicates an empty stack.
5) Checking whether stack is full:-
If the value of top reaches the maximum array index i.e. MAX-1 no more elements can be pushed into the stack.
6) The push operation:-

The element can be pushed into the stack only if it is not full. In such case the top has to be incremented first and the element has to be put in this position.

7) The pop operation:

An element can be removed from the stack if it is not empty. The topmost element can be removed after which top has to decremented

8) The peek operation:

It displays the topmost element of the stack without decrementing the top.

**2) Dynamic implementation:-**

- Pointers are used for implementation of stack. The linked list is an e.g. of this implementation.
- The limitations noticed in static implementation can be removed using dynamic implementation. The dynamic implementation is achieved using pointers.
- Using pointer implementation at runtime there is no restriction on the no. of elements. The stack may be expandable.
- The memory is efficiently utilized with pointers.
- Memory is allocated only after element is pushed to the stack
- In static representation there is a limitation on the size of the array if less elements are stored, memory will be wasted. To overcome the program the stack can be implemented using linked list.
- In the linked organization
  - The stack can grow to any size.
  - We need not have prior knowledge of the number of elements.
- When an element is popped the memory can be freed. Thus memory is not unnecessary occupied.
- Since random access to any element is not required in a stack, the linked representation is preferred over the sequential organization.

**SET A :**

1) Write a C program to implement Static implementation of stack of integers with following operation:
   -Initialize(), push(), pop(), isempty(), isfull(), display(), peek()

2) Write a C program to implement Dynamic implementation of stack of integers with following operation:
   -Initialize(), push(), pop(), isempty(), display(), peek()

**SET B :**

1) Write a C program to reverse the given string by using static and dynamic implementation of stack.

2) Write a C program to reverse the given number by using static and dynamic implementation of stack.

**SET C:**

1) Write a C program to reverse the stack by using recursive function.
2) Write a C program to convert infix expression to postfix expression
3) Write a C program to evaluate postfix expression.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No : 6
# Queue

A queue is an ordered collection of items from which items may be deleted from (or removed from ) one end called the front and into which items may be inserted at the other end called rear.

- **BASIC OPERATIONS ON QUEUE**
  1) Create:
     Creates a new queue. This operation creates an empty queue.
  2) Add or insert:
     Adds an element to the queue. A new element can be added to the queue at the rear.
  3) Delete:
     Remove an element from the queue. This operation removes the elements, which is at the front of the queue. This operation can only be performed if the queue is not empty.
     The result of an illegal attempt to remove an element from an empty queue is called underflow.
  4) isempty:
     Checks whether a queue is empty. The operation return true if the queue isempty and false otherwise
  5) isfull:
     Checks whether a queue is full. The operation return true if the queue isfull and false otherwise

- **REPRESENTATION OF LINEAR QUEUES.**
  There are two ways to represent a queue in memory.
  1) Static (using an array)
  2) Dynamic (using an linked list)

  1) Static implementation of queue
     Static implementation or array representation of queue requires three entities-
     - An array to hold queue elements.
     - A variable to hold the index of the front element.
     - A variable to hold the index of the rear element.

     The implementation of a queue using sequential representation is done by using some size MAX and two integer variable front and rear. Initially front and rear is set to -1. Whenever new element is added it is added from the rear and whenever an element is to be removed from the front. The queue full condition is when rear reaches to MAX - 1. Queue empty condition is when front is equal to rear.

  2) Dynamic implementation of linear queue (using an linked list)

A queue can be considered as a list in which all insertions are made at one end called the rear and all deletions from the other end from front.

A queue can be easily represented using a linked list. The front and rear will be two pointers pointing to the first and last node respectively.

**SET A :**

1) Write a C program to Implement Static implementation of Queue of integers with following operation:
   -Initialize(), insert(), delete(), isempty(), isfull(), display(), peek()

**SET B:**

1) Write a C program to Implement Dynamic implementation of Queue of integers with following operation:
   -Initialize(), insert(), delete(), isempty(), display(), peek()

**SET C:**

1) Write a C program to Implement Static implementation of circular queue of integers with following operation:
   -Initialize(), insert(), delete(), isempty(), isfull(), display(), peek()

2) Write a C program to Implement Dynamic implementation of circular queue of integers with following operation:
   -Initialize(), insert(), delete(), isempty(), display(), peek()

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: WellDone [ ] |

**Signature of Instructor**

# Assignment No : 7
# Trees

**Definition of tree:-**

A tree is a finite set of one or more nodes such that:- there is a specially designated node called the root. The remaining nodes are partitioned into n>=0 disjoint sets T1……Tn where each of these sets is a tree. T1…..Tn are called as sub-trees of the root.

The operations on binary search tree are
init (T) – creates an empty Binary search tree by initializing T to NULL
insert (T, x) – inserts the value x in the proper position in the Binary search tree
search (T, x) – searches if the value x is present in the search tree
inOrder (T) – displays the node using inorder traversal of binary search tree
postOrder (T) – displays the node using postorder traversal of binary search tree
preOrder (T) – displays the node using preorder traversal of binary search tree

**SET A:**

1) Write C programs to implement create and display operation for binary tree.
2) Write C programs to implement create and display operation for binary search tree.

**SET B:**

1) Implement the following programs by adding the functions one by one in Q2 of SET A
   i) Add a function to insert a new element in the tree.
   ii) Add a function to search an element in tree and give the proper message.
   iii) Add a function to create mirror image of the tree.
   iv) Add a function to count non-leaf nodes.
   v) Add a function to count leaf nodes.
   vi) Add a function to count total number of nodes.

**SET C:**

1) Write a C program to Implement Binary search tree with following operations
   - Create(), inorder(), preorder(), postorder()

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: WellDone [ ] |

**Signature of Instructor**

# Assignment No : 8
# Graph

A graph consists of a set of vertices and a set of edges. The two main ways of representing graphs are adjacency matrix representation and adjacency list representation. In adjacency matrix representation of a Graph with n vertices and e edges, a two dimensional nxn array , say a , is used , with the property that a[i,j] equals 1 if there is an edge from i to j and a[i,j] equals 0 if there is no edge from i to j.
In adjacency list representation of a graph with n vertices and e edges, there are n linked lists, one list for each vertex in the graph.

The usual operations on graph are:
Indegree(i) – returns the indegree (the number of edges ending on) of the ith vertex
Outdegree(i) – returns the outdegree(the number of edges moving out) of the ith vertex
displayAdjMatrix – displays the adjacency matrix for the graph

**SET A:**

1) Write a C program to read a graph as adjacency matrix and display the adjacency matrix.
2) Add a function in Q1 (above question) to count total degree , indegree and outdegree of the graph.

**SET B:**

1) Write a C program to convert adjacency matrix into adjacency list. Display the adjacency list.
2) Write a C program to traverse graph by using BFS.

**SET C:**

1) Write a C program to traverse graph by using DFS.
2) Implement a program for simple transpose of the matrix.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2: Late Complete [ ] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: WellDone [ ] |

**Signature of Instructor**

Savitribai Phule Pune University

# Section-II

S. Y. B. C. A. (Science)

SEMESTER III

BCA - 306

# Lab Course - II
# Advanced Relational Database Management System Assignments

**Table of Contents for Section-II**

**Assignment Completion Sheet**

| | | | |
|---|---|---|---|
| Lab Course II | | | |
| Advanced Relational Database Management System Assignments | | | |
| **Sr. No.** | **Assignment Name** | **Marks (out of 5)** | **Instructor Sign** |
| 1 | Queries | | |
| 2 | Views | | |
| 3 | Stored Functions | | |
| 4 | Error and Exception Handling | | |
| 5 | Cursors | | |
| 6 | Triggers | | |
| Total (Out of 30) | | | |
| Total (Out of 10) | | | |

AHMEDNAGAR JILHA MARATHA VIDYA PRASARAK SAMAJ'S

**NEW ARTS, COMMERCE AND SCIENCE COLLEGE**

Department of Computer Applications (BCA Science)

# Certificate

This is to certify that Mr. /Ms. ________________________________________

has successfully completed the course work for Lab Course II :BCA:306:ARDBMS and has scored ______ Marks out of 30.

Instructor H.O.D / Coordinator

**Internal Examiner** **External Examiner**

# Assignment No. 1 Queries

## PostgreSQL-Data Types

A datatype specifies, what kind of data you want to store in the table field. While creating table, for each column, you have to use a datatype. There are different categories of data types in PostgreSQL discussed below for your ready reference:

| Type | Data Type | Description |
|---|---|---|
| **Numeric Types** | smallint | 2-byte small-range integer |
| | integer, int | A signed, fixed precision 4-byte |
| | bigint | stores whole numbers, large range 8 byte |
| | real | 4-byte, single precision, floating-point number |
| | serial | 4-byte auto incrementing integer |
| | double precision | 8-byte, double precision, floating-point number |
| | numeric(m,d) | Where m is the total digits and d is the number of digits after the decimal. |
| **Character Types** | character(n), char(n) | Fixed n-length character strings. |
| | character varying(n), varchar(n) | A variable length character string with limit. |
| | text | A variable length character string of unlimited length. |
| **Monetary Types** | money | currency amount,8 bytes |
| **Boolean type** | boolean | It specifies the state of true or false,1 byte. |
| **Date/Time Type** | date | date (no time of day),4 byte. |
| | time | time of day (no date),8 byte |
| | time with time zone | times of day only, with time zone,12 bytes |

| | | |
|---|---|---|
| | bit(n) | Fixed-length bit string Where n is the length of the bit string. |
| | varbit(n)<br>bit varying(n) | Variable-length bit string, where n is the length of the bit string. |

## Instructions:-

1. Create a relational database in 3NF.
2. Insert sufficient number of records in the table.
3. Read the question carefully and insert data accordingly.
4. Count queries output should be more than 2 records.
5. Empty output should not be generated.

## SET A

### Student-Teacher Database

**Consider the following Entities and their Relationships for Student-Teacher database.**
**Student** (s_no int, s_name varchar (20), s_class varchar (10), s_addr varchar (30))
**Teacher** (t_no int, t_name varchar (20), qualification varchar (15), experience int)

Relationship between Student and Teacher is many to many with descriptive attribute subject.

**Constraints:** Primary Key,
s_class should not be null.

**Solve the following Queries in PostgreSQL:**

1. List the names of students of class 'FYBCA'.
2. List the names of the students to whom '_______' is teaching.
3. List the details of all teachers whose names start with the alphabet 'S'.
4. List the names of teachers teaching subject 'ARDBMS'.
5. Find the number of teachers having qualification as 'Ph. D.'.
6. Find the number of students living in 'Deccan'.
7. Find the details of maximum experienced teacher.

8. Find the names of students of class ‘SYBCA’ and living in ‘Sangvi’.
9. List the names of all teachers with their subjects along with the total number of students they are teaching.
10. List the names of students who are taught by most experienced teacher.

## SET B

## Book-Author Database

**Consider the following Entities and their Relationships for Book-Author database.**
**Book** (b_no int, b_name varchar (20), pub_name varchar (10), b_price float)
**Author** (a_no int, a_name varchar (20), qualification varchar (15), address varchar (15))

Relationship between Book and Author is many to many.

**Constraints:** Primary Key,
pub_name should not be null.

**Solve the following Queries in PostgreSQL:**

1. List details of all books written by ‘______’
2. Count the number of books published by ‘bpb’ publication.
3. List book details for which book price is between 300.00 and 500.00.
4. List all author details sorted by their name in descending order.
5. Change the publisher name from 'Niyati Publications' to 'Jagruti Publications'.
6. List the details of all books whose names start with the alphabet ‘R’.
7. List author wise details of books.
8. Display details of authors who have written more than 2 books.
9. List the details of all books written by author living in ‘Pune’.
10. Display details of authors who have written maximum number of books.

## SET C

## Student-Competition Database

**Consider the following Entities and their Relationships for Student-Competition database.**
**Student** (sreg_no int ,s_name varchar(20), s_class char(10))
**Competition** (c_no int ,c_name varchar(20), c_type char(10))

Relationship between Student and Competition is many to many with descriptive attributes rank and year.

**Constraints:** Primary Key,
c_type should not be null,
rank should be greater than 0,
c_type can be ‘sport’ or ‘academic’.

**Solve the following Queries in PostgreSQL:**

1. List out all the competitions held in the school for class 4th.
2. Count all the students who have secured 1st rank in running race from year 2015 to 2017.
3. Delete the record of student 'Amit Kale' which has taken part in drawing Competition.
4. List out all the competitions held in the school under 'academic' in year 2016.
5. List the names of all the students who have secured 3rd rank in dance competition in year 2017.
6. List out all the competitions held in the school for class 8th in year 2016.
7. List the names of all the students who have secured 1st rank in more than two competitions.
8. Change the rank to 1st of student 'Subodh Kadam' which has taken part in dance competition.
9. List the names of students of 'FYBCA' class participated under 'sport'.
10. List the competition wise participated students names.

## SET D

## Movie-Actor Database

**Consider the following Entities and their Relationships for Movie-Actor database.**
**Movie** (m_name varchar (25), release_year integer, budget money)
**Actor** (a_name varchar (20), role char (20), charges money, a_address varchar (20))
**Producer** (producer_id integer, p_name char (30), p_address varchar (20))

Each actor has acted in one or more movies. Each producer has produced many movies and each movie can be produced by more than one producers. Each movie has one or more actors acting in it, in different roles.

**Constraints:** Primary Key,
role and p_name should not be null.

**Solve the following Queries in PostgreSQL:**

1. List the names of the actors and their movie names.
2. List the names of movies whose producer is 'Mr. Karan Johar'
3. List the names of the movies with the minimum budget.
4. List the names of movies released after year 2011.
5. Display count and total budget of all movies released in year 2014.
6. List the names of actors who have acted in maximum number of movies.
7. List the names of movies produced by more than one producer.
8. List the names of movie and actor name, with maximum actor charges.
9. List the names of actors who have acted in at least one movie, in which Mr. Amir Khan has acted.
10. Display total number of actors acted in movie 'Dhoom'.

## SET E

## Person-Area Database

**Consider the following Entities and their Relationships for Person-Area database.**
**Person** (pnumber integer, pname varchar (20), birthdate date, income money)
**Area** (aname varchar (20), area_type varchar (5))

An area can have one or more persons living in it, but a person belongs to exactly one area.

**Constraints**: Primary Key,
area_type can be either 'urban' or 'rural'.

**Solve the following Queries in PostgreSQL:**

1. List the names of all persons living in 'Pune' area.
2. List the details of all people whose names start with the alphabet 'P'.
3. Count area wise persons whose income is above ______.
4. List the names of all people whose income is between ______and _______.
5. List the names of all people whose birthday falls in the month of September.
6. List names of persons whose income is same.
7. Display area wise maximum income of person.
8. Update the income of all person living in rural area by 5%.
9. Delete the record of person which has income below ________.
10. Count the total number of person according to area_type.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2:Late Complete[] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No. 2 Views

Views are imaginary tables, they are not real tables. A view can contain all rows of a table or selected rows from one or more tables. We can use views to restrict table access so that the users see only specific rows and columns of the tables.

- **Creating View:**

Views are created using the CREATE VIEW statement. The PostgreSQL views can be created from a single table or multiple tables or another view.

**Syntax for creating view:**

```
CREATE [or REPLACE] [TEMP|TEMPORARY] VIEW view_name AS
SELECT column1, column2…..
FROM table_name
WHERE [condition];
```

**Parameters Used:**

1. **view_name:** The name of a view to be created.
2. **TEMPORARY or TEMP:** If the optional TEMP or TEMPORARY keyword is present, the view will be created in the temporary space. Temporary views are automatically dropped at the end of the current session.
3. **column_name:** Name of the columns should be displayed in view.
4. **table_name:** Name of the table from which data should be fetched.
5. **condition:** According to the condition data fetched from the table.

**Example:**

- **List the details of all employees having qualification as 'MCS'.**

```
CREATE VIEW emp_view AS
SELECT *
FROM employee
WHERE qualification= 'MCS';
```

This will create a view containing the columns that are in the employee table at the time of view creation. Now we can query emp_view in similar way as we query an actual table to see the data.

- **To see the data from view:**
  **Syntax:**
  ```
  select * from view_name;
  E.g. select * from emp_view;
  ```
  The above query displays the record of only those employees whose qualification is 'MCS' from view.

- **Dropping View:**
  To delete a view, simply use the Drop view statement with the view name.
  **Syntax:**
  ```
  DROP VIEW view_name;
  E.g. DROP VIEW emp_view;
  ```

## <u>SET A</u>

### 1) Student-Marks Database

**Consider the following Entities and their Relationships for Student-Marks database.**
**Student** (rollno integer, s_name varchar (20), address varchar (25), class varchar (10))
**Subject** (scode varchar (10), subject_name varchar (20))

Relationship between Student and Marks is many to many with descriptive attribute marks_scored.

**Constraints:** Primary Key,
s_name should not be null.

**Create view for the following:**

1. To display student name, class and total marks scored by each student, sorted by student name.
2. To display student names along with subject and marks who has scored more than 80 marks.

### 2) Bus Transport Database

**Consider the following Entities and their Relationships for Bus Transport database.**
**Bus** (bus_no int , b_capacity int , depot_name varchar(20))
**Route** (route_no int, source char (20), destination char (20), no_of_stations int)
**Driver** (driver_no int ,driver_name char(20), license_no int, address char(20), d_age int , salary float)

Relationship between Bus and Route is many to one and relationship between Bus and Driver is many to many with descriptive attributes date_of_duty_allotted and shift.

**Constraints:** Primary Key,
license_no is unique,
b_capacity should not be null,
shift can be 1 (Morning) or 2(Evening).

**Create view for the following:**

1. To list details of bus no. 10 along with details of all drivers who have driven that bus in Evening shift.
2. To contain details of all the routes which are between the stations 'Nigdi' and 'Corporation'.

## **SET** B

### 1) Warehouse Database

**Consider the following Entities and their Relationships for Warehouse database.**
**Cities** (city char (20), state char (20))
**Warehouses** (wid integer, wname char (30), location char (20))
**Stores** (sid integer, store_name char (20), location_city char (20))
**Items** (itemno integer, description text, weight decimal (5, 2), cost decimal (5, 2))
**Customer** (cno integer, cname char (50), addr varchar (50), c_city char (20))
**Orders** (ono int, odate date)

Relationship between Cities-Warehouses is one to many, Warehouses-Stores is one to many, Customer-Orders is one to many, Items-Orders is many to many with descriptive attribute ordered_quantity, Stores-Items is many to many with descriptive attribute quantity.

**Constraints:** Primary Key,
wname should not be null.

**Create view for the following:**

1. To contain details of all the stores of a Warehouse named 'Spares' located at 'Pune'.
2. To list the details of all customers who have placed orders on the date '12-11-2015'.

### 2) Railway Reservation Database

**Consider the following Entities and their Relationships for Railway Reservation database.**
**Train** (tno int, tname varchar (20), depart_time time, arrival_time time, source_stn char (10), dest_stn char (10), no_of_res_bogies int ,bogie_capacity int)
**Passenger** (passenger_id int, passenger_name varchar (20), address varchar (30), age int, gender char)

Relationship between Train and Passenger is many to many with descriptive attribute ticket.
**Ticket** (train_no int, passenger_id int, ticket_no int,bogie_no int, no_of_berths int, tdate date, ticket_amt decimal (7,2),status char)

**Constraints:** Primary Key,
Status of a berth can be 'W' (waiting) or 'C' (confirmed).

**Create view for the following:**

1. To list the two bookings of 'Sahyadri Express' on date '11-10-2017'.
2. To list the passenger names who have reserved births more than 2 of 'Indrayani Express' on '24-06-2017'.

## <u>SET C</u>

### 1) Business-Trips Database

**Consider the following Entities and their Relationships for Business-Trips database.**
**Salesman** (sno integer, s_name varchar (30), start_year integer)
**Trip** (tno integer, from_citychar (20), to_citychar (20), departure_date date, return_date date)
**Dept** (deptno varchar (10), dept_name char (20))
**Expense** (eid integer, amount money)

Relationship between Dept and Salesman is one to many, Salesman and Trip is one to many, and Trip and Expense is one to one.

**Constraints:** Primary Key,
s_name should not be null.

**Create view for the following:**

1. To find the total expenses incurred by the salesman 'Mr. Pawar'.
2. To List the names of departments that have salesmen, who have done minimum number of trips.

### 2) Bank Database

**Consider the following Entities and their Relationships for Bank database.**
**Branch** (br_id integer, br_name char (30), br_city char (10))

**Customer** (cno integer, c_name char (20), caddr char (35), city char (20))
**Loan_application** (lno integer, l_amt_required money, l_amt_approved money, l_date date)

Relationship between Branch, Customer and Loan_application is Ternary.
**Ternary** (br_id integer, cno integer, lno integer)

**Constraints:** Primary Key,
l_amt_required should be greater than zero.

**Create view for the following:**

1. Display the details of all customers who have received loan amount less than their requirement.
2. Display details of those customers whose birthday falls in the month of 'June'.

## SET D

### 1) Project-Employee Database

**Consider the following Entities and their Relationships for Project-Employee database.**
**Project** (pno integer, pname char (30), ptype char (20), duration integer)
**Employee** (eno integer, ename char (20), qualification char (15), joining_date date)

Relationship between Project and Employee is many to many with descriptive attribute start_date date, no_of_hours_worked integer.

**Constraints**: Primary Key,
duration should be greater than zero,
pname should not be null.

**Create view for the following:**
1. Create a view containing the project details and start date of the project, sort it by start date of the project.
2. Create a view to display employee details and it should be sorted on employee name.

### 2) Person-Area Database

**Create view for the following:**

1. List the details of all persons living in ‘Pimpri’ area.
2. Display area wise person details sorted by area_type.
3.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2:Late Complete[] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No. 3 Stored Functions

PostgreSQL functions, also known as Stored Procedures, allow you to carry out operations that would normally take several queries. Stored Procedures are user-defined functions.

- **Syntax to create a function:**

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS'
DECLARE
        Variable_Declarations;
                [...]
BEGIN
        <function_body>
            [...]
            RETURN {variable_name | value}
END;'
LANGUAGE 'plpgsql';
```

Where,

1. **function_name:** Specifies the name of the function.
2. **Arguments:** In PL/pgSQL functions can accept arguments/parameters of different types. Function arguments allow a user to pass information to a function. Each function parameter has assigned identifier that begins with dollar ($) sign and labeled with the parameter number. Identifier $1 is used for first parameter, Identifier $2 is used for second parameter and so on.

PL/pgSQL allows us to create Aliases. With the help of aliases, it is possible to assign more than just one name to a variable. Aliases should be declared within the DECLARE section of a block.

**Syntax:**

```
variable_name ALIAS FOR $1;
E.g. ename ALIAS FOR $1;
```

3. **[OR REPLACE]:** This option allows modifying an existing function.
4. The AS keyword is used for creating a standalone function.
5. **function_body:** Contains the executable part.
6. **Return:** The function must contain a **return** statement. This clause specifies that data type you are going to return from the function.
7. **plpgsql** is the name of the language that the function is implemented in. For backward compatibility, the name can be enclosed by single quotes.
8. **Attributes:** PL/pgSQL provides two attributes to declare variables.

   a) **The %TYPE attribute:**
   It is used declare variables based on definitions of columns in a table. This attribute is used to declare the variable whose type will be same as that of a previously defined variable or a column in a table.

   **Syntax:**

   ```
   variable_name table_name.column_name%TYPE;
   E.g. ename employee.emp_name%type;
   ```

   b) **The %ROWTYPE attribute:**
   It is used to declare a record variable with same structure as the rows in a table that are specified. It will have structure exactly similar to the table's row.

   **Syntax:**

   ```
   variable_name table_name%ROWTYPE;
   E.g. e employee%rowtype;
   ```

- **Calling Function:** Function can be called by using below command on terminal.

  **Syntax:**

  ```
  select function_name(arguments);
  ```

- **Drop Function:** DROP FUNCTION removes the definition of an existing function.

  **Syntax:**

  ```
  DROP FUNCTION function_name (argtype[, ...]) [CASCADE | RESTRICT]
  ```

  Where,

  **1. function_name:** The name of an existing function.
  **2. argtype:** The data type(s) of the function's arguments if any.
  **3. CASCADE:** Automatically drop objects that depend on the function (such as triggers).
  **4. RESTRICT:** Refuse to drop the function if any objects depend on it. This is the default.

## Control Structures

Like the most programming languages, PL/pgSQL also provides ways for controlling flow of program execution by using conditional statements and loops.

1. **Conditional Statements:**

   A conditional statement specifies an action (or set of actions) that should be executed instead of continuing execution of the function, based on the result of logical condition specified within the statement.

   - **IF…. THEN Statement:**

   IF….THEN statement, a statement or block of statements is executed if given condition evaluates true.

   **Syntax:**

```
IF condition THEN
Statements;
END IF;
```

   **Example:**

```
create or replace function simple_if()
returns int as'
declare
n int;
begin
        select into n ano from a;
        if n IS null then
        return -1;
        end if;
        return 1;
end;'
language 'plpgsql';

Lab=# select simple_if();
```

   - **IF…. THEN…. ELSE Statement:**

     This statement allows to execute a block of statements if a condition evaluates to true, otherwise a block of statements in else part is executed.

     **Syntax:**

```
IF condition THEN
Statements
ELSE
Statements
END IF;
```

   **Example:**

```
create or replace function even_odd(int)
returns void as'
declare
n alias for $1;
begin
        if (n % 2 =0) then
        raise notice "Number is Even";
```

```
            else
            raise notice "Number is Odd";
            end if;
        end;'
        language 'plpgsql';

        Lab=# select even_odd(6);
```

- **IF…. THEN…. ELSEIF…. THEN…ELSEStatement:**
  IF-THEN-ELSEIF provides a convenient method of checking several alternatives in turn.
  **Syntax:**

```
IF condition THEN
Statements;
ELSEIF condition THEN
Statements;
ELSEIF condition THEN
Statements;
ELSE
Statements;
END IF;
```

**Example:**

```
            create or replace function Greater_Number(int, int, int)
            returns void as'
            declare
            n1 alias for $1;
            n2 alias for $2;
            n3 alias for $3;
            begin

            if(n1 > n2) and (n1 >n3) then
            raise notice "Largest Number Is: =%",n1;
            elseif(n2>n3) then
            raise notice "Largest Number Is: =%",n2;
            else
            raise notice "Largest Number Is: =%",n3;
            end if;
        end;
        'language 'plpgsql';

        Lab=# select Greater_Number(6,4,7);
```

- **CASE Statement:**
  The simple form of CASE provides conditional execution based on equality of operands. The search-expression is evaluated (once) and successively compared to each expression in the WHEN clauses. If a match is found, then the corresponding statements are executed,

and then control passes to the next statement after END CASE. If no match is found, the ELSE statements are executed; but if ELSE is not present, then a CASE_NOT_FOUND exception is raised.

- **Syntax:**

```
CASE search-expression
        WHEN expression [, expression [...]] THEN
        Statements
        [WHEN expression [, expression [...]] THEN
        Statements
        ... ]
        [ELSE
        Statements]
END;
```

**Example:**

```
create or replace function Case_Demo(int)
returns void as'
declare
msg text;
begin
select into msg
CASE $1
        when 1 then "One"
        when 2 then "Two"
        else "Other"
end;
raise notice "%",msg;
end;'
language 'plpgsql';


Lab=# select Case_Demo (1);

Lab=# select Case_Demo (3);
```

2. **Loop Statements:**

Loop condition is used to perform some task repeatedly for fixed number of times.

a) **Simple Loop:**

**Syntax:**

```
LOOP
                Statement;
                [...]
                EXIT [label] [WHEN condition];
 END LOOP;
```

**Example:**

```
create or replace function simple_loop()
```

```
returns void as'
declare
cnt int: =0;
begin
        loop
        cnt: =cnt+1;
        raise notice " Hello";
        exit when cnt=3;
        end loop;
end;'
language 'plpgsql';

Lab=# select simple_loop();
```

**b) While Loop:**

The While statement repeats a sequence of statements till the condition evaluates to true. The expression is checked just before each entry to the loop body.

**Syntax:**

```
[<<Label>>]
WHILE condition LOOP
        Statements;
        [...]
END LOOP;
```

**Example:**

```
create or replace function Disp_Sum(int)
returns int as'
declare
n1 alias for $1;
sum1 int:=0;
cnt int: =1;
begin
        while (cnt <=n1)
        loop

        sum1: =sum1+cnt;

        cnt: =cnt+1;
        end loop;
        raise notice "Sum of first % number is",sum1;
        return sum1;
end;'
language 'plpgsql';

Lab=# select Disp_Sum(4);
```

c) **For Loop:**

This form of FOR creates a loop that iterates over a range of integer values. The variable name is automatically defined as type integer and exists only inside the loop. The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop.

**Syntax:**

```
[<<Label>>]
FOR name IN [REVERSE] expression.. Expression [ BY expression ]
LOOP
        Statements
END LOOP [label];
```

**Example 1:**

```
create or replace function For_Demo()
returns void as'
declare
cnt integer;
begin
        for cnt in 1..4 loop
        raise notice '' Counter: =%'',cnt;
        exit when cnt=3;
end loop;
end;'
language 'plpgsql';

Lab=# select For_Demo();
```

**Example 2:**

```
create or replace function For_Demo()
returns void as'
declare
cnt integer;

begin
        for cnt in reverse 4..1 loop
        raise notice '' Counter: =%'', cnt;
        exit when cnt=3;

        end loop;

end;'
language 'plpgsql';

Lab=# select For_Demo();
```

**Looping Through Query Results by Using for Loop:**

Using a different type of FOR loop, you can iterate through the results of a query and manipulate that data accordingly.

**Syntax:**

```
[<<Label>>]
FOR record _variable IN query
 LOOP
        Statements;
END LOOP;
```

3. **Exit Statement:**
   These statements are used to exit from a loop.
   a) **exit:** This statement is used to exit from loop without condition. Condition needs to be specified separately.
      **Syntax:**
      ```
      exit;
      ```
      **Example:**
      ```
      if (condition) then
              EXIT;
      else
              statements;
      end if;
      ```

   b) **exit-when:** This statement is used to exit from loop by specifying condition within exit statement.
      **Syntax:**
      ```
      EXITWHEN condition;
      ```
      **Example:**
      ```
      .
      EXIT WHEN count > 2;
      ```

**Example 1: Simple Function.**

**Consider the following Relational Database:**
**Dept** (dno, dname)
**Emp** (eno, ename, dno)

- **Count total number of employees of computer department.**

```
create or replace function emp_cnt()
returns int as'


declare
cnt int;
begin
```

```
        select into cnt count(eno) from Emp, Dept where Emp.dno=Dept.dno and dname=
        "computer" ;
        return cnt;

 end;'
 language 'plpgsql';

 Lab=# select emp_cnt();
```

**Example 2: Looping Through Query Results by Using for Loop.**

**Consider the following Relational Database:**
**Project** (pno, pname, ptype, duration)
**Employee** (eno, ename, qualification, joining_date)
**PE** (pno, eno, start_date, no_of_hours_worked)

- **Accept project name as input and print the names of employees working on that project**.

```
create or replace function proj_chk(text)
returns void as'
declare
p_name alias for $1;
rec record;
begin
        raise notice "Employee Name || Project Name";
        for rec in select ename, pname from Employee, Project,PE where
        Employee.eno=PE.eno and Project.pno=PE.pno and pname="p_name"
        loop
        raise notice "% %",rec.ename, rec.pname;
        end loop;
end;'
language 'plpgsql';

Lab=# select proj_chk('System');
```

## <u>SET A</u>
## Person-Area Database

1. Write a stored function to print total number of persons of a particular area. Accept area name as input parameter.
2. Write a stored function to update the income of all persons living in urban area by 20%.

## <u>SET B</u>
## Student-Teacher Database

1. Write a stored function to count the number of the teachers who are teaching to a given student. Accept student name as input parameter.
2. Write a stored function to find the details of minimum experienced teacher.

## <u>SET C</u>

### Railway Reservation Database

1. Write a stored function to accept date and passenger name and display no. of berths reserved and ticket amount paid by him/her.

2. Write a stored function to update the status of the ticket from ‘waiting’ to ‘confirmed’ for
   passenger named ‘Mr. Sharma’.

## <u>SET D</u>

### Project-Employee Database

1. Write a stored function to find the number of employees whose joining date is before ‘01/01/2007’.
2. Write a stored function to accept eno as input parameter and count number of projects on which that employee is working.

## <u>SET E</u>

### Student-Competition Database

1. Write a stored function to accept a sreg_no and year as input parameter and returns the total number of prizes won by that student in that year.

2.Write a stored function to find the name of student who has participated in maximum number of competitions in the year 2017.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2:Late Complete[] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No. 4 Error and Exception Handling

- Any error occurring in a PL/pgSQL function aborts execution of the function. Errors can be trapped and recovered by using a Begin block with an Exception clause.
  **Syntax:**
  ```
  Declare
          Declarations
  Begin
          Statements
  Exception
          When condition then handler statements
  End;
  ```
- If no errors occur, this form of block simply executes all the statements, and then control passes to the next statement. But if error occurs within the statements, further processing of the statements is stopped and control passes to the exception list.
- **Raise Statement**: It is used to raise errors, report messages and exceptions during a PL/pgSQL function's execution.
  This statement can raise predefined exceptions, such as division_by_zero, not found.
  **Syntax**:
  ```
  RAISE level 'format' [, expression [, ...]];
  ```
- **level:** The level option that specifies the error severity.
  **There are following levels in PostgreSQL:**
  1. **DEBUG:** DEBUG level statements send the specified text as a message to the PostgreSQL log.
  2. **NOTICE:** This level statement sends the specified text as a message.
  3. **EXCEPTION:** This statement **s**ends the specified text as error message.
- **format:**
  The format is a string that specifies the message. The format uses percentage (%) placeholders that will be substituted by the next arguments. The number of placeholders

must match the number of arguments; otherwise PostgreSQL will report the following error message:

e.g. [Err] ERROR:  too many parameters specified for RAISE

**Example:**

- **In this example, the first raise statement gives a debug level message and sends specified text to PostgreSQL log. The second statements send a notice to the user. The third raise statement displays an error and throws an exception, causing the function to end.**

```
create or replace function raise_demo(int, int)
returns void as'
declare
result int;
n1 alias for $1;
n2 alias for $2;
begin
        RAISE DEBUG ''raise_demo() function began'';

        if (n2!=0) then
        result:=n1/n2;
        RAISE NOTICE ''Result is:= %'',result;
        else
        RAISE EXCEPTION ''Exception Raised.....Dividing by Zero...'';
        end if;
END;'
language 'plpgsql';

Lab=# select raise_demo (4, 2);

Lab=# select raise_demo (4, 0);
```

## <u>SET A</u>

### Movie-Actor Database

1. Write a stored function to accept producer name as input and display the names of movies produced by that producer. (Accept producer name as input parameter). Raise an exception for an invalid producer name.
2. Write a stored function to accept movie name as input and display the details of actors for that movie and sort it by their charges. (Accept movie name as input parameter). Raise an exception for an invalid movie name.

## <u>SET B</u>

### Bus Transport Database

1. Write a stored function to accept the bus_no, date and shift and display details of allotted driver. (Accept bus_no as input parameter) Raise an exception in case of invalid bus number.
2. Write a stored function to accept route_no as input and display the details of buses that run

on given route_no. (Accept route_no as input parameter). Raise an exception for an invalid route number.

**SET C**

**Bank Database**

1. Write a stored function to count the number of customers of particular branch. (Accept branch name as input parameter) Display appropriate error message if branch name is invalid.
2. Write a stored function to accept cno as input and display the loan details of that customer. (Accept customer number as input parameter). Raise an exception for an invalid customer number.

**SET D**

**Project-Employee Database**

1. Write a stored function to accept project name as input and display the names of employees working on that project. (Accept project name as input parameter). Raise an exception for an invalid project name.
2. Write a stored function to accept eno as input and count the number of projects of particular employee. (Accept employee number as input parameter) Display appropriate error message if employee number is invalid.

**SET E**

**Warehouse Database**

1. Write a stored function to accept store_name as input and count the number of items of that store. (Accept name of store as input parameter) Display appropriate error message if store name is invalid.
2. Write a stored function to accept a city as input and display all warehouses in that city. (Accept city name as input parameter) Raise exception if city name is not valid.

**Assignment Evaluation**

0: Not Done [ ] 1: Incomplete [ ] 2:LateComplete[]

3: Needs Improvement [ ]     4: Complete [ ]     5: Well Done [ ]

**Signature of Instructor**

# Assignment No. 5 Cursors

A PL/pgSQL uses a temporary work area in memory for its internal processing of SQL statements. This private work area is known as cursor. Cursor allows user to access or retrieve multiple rows of data from a table, so that user can perform different operations on that data one row at a time.

We use cursors when we want to divide a large result set into parts and process each part individually. If we process it at once, we may have a memory overflow error.

- **Steps for cursor:**
  1. First, declare a cursor.
  2. Next, open the cursor.
  3. Then, fetch rows from the result set into a target.
  4. After that, check if there are more rows left to fetch. If yes, go to step 3, otherwise go to step 5.
  5. Finally, close the cursor.

1. **Declaration:**
   We can declare a cursor variable by using two ways:

   a) **Bound Cursor Variable:** Query already bound to this cursor variable.
   **Syntax:** DECLARE
   cursor_name CURSOR [(arguments)] FOR Query
   E.g. cur1 CURSOR FOR SELECT * FROM EMPLOYEE;

   b) **Unbound Cursor Variable:** Declare the cursor variable of type REFCURSOR.
   **Syntax:** DECLARE
   cursor_name REFCURSOR;
   E.g. cur1 REFCURSOR;

2. **Opening Cursor:**
   Cursors must be opened before they can be used to fetch rows.

**Bound Cursor:** open cursor_name;
**Unbound Cursor:** This cursor variable not bounded to any query when we declared it, so we have to specify the query when we open it.
**Syntax:** open cursor_name FOR query;

3. **Fetching Rows:**
The FETCH statement retrieves the next row from the cursor and assigns it a target variable, which could be a record, row variable. If no more rows found, the target variable is set to NULL.
**Syntax:** FETCH [direction {FROM | IN} ] cursor INTO target;
FETCH cur1 INTO rec;
FETCH LAST FROM cur1 INTO eno, ename;

The direction clause can be any of the following variants:
NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, or BACKWARD. Default is NEXT.

**MOVE:** MOVE repositions a cursor without retrieving any data.

**Syntax:**
MOVE [direction {FROM | IN}] cursor;
The direction clause can be any of the variants NEXT, PRIOR, FIRST, LAST, ABSOLUTE
count, RELATIVE count, ALL, FORWARD [count | ALL], or BACKWARD [count | ALL].

4. **Closing Cursors:**
To close an opening cursor, we use CLOSE statement as follows:
**Syntax:** CLOSE cursor_variable;
**For Example:** CLOSE cur1;

The CLOSE statement releases resources or frees up cursor variable to allow it to be opened again using OPEN statement.

**Example 1: Multiple Cursors**

**Consider the following Relational Database:**
**Route** (rno, source, destination, no_of_stations )
**Bus** (bno, capacity, depot_name, rno)

- **Display the details of the buses that run on route_no=1 and route_no=2 using multiple cursors.**

```
create or replace function Disp_route()
            returns void as'
            declare
```

```
c1 cursor for select * from bus where rno=1;
c2 cursor for select * from bus where rno=2;
rec1 record;
rec2 record;
begin
        open c1;
        raise notice "Details of Buses run on route 1 are:";
        raise notice "Bus No Capacity Depot Name";
        loop
                fetch c1 into rec1;
                exit when not found;
                raise notice " % % % ",rec1.bno,rec1.capacity,rec1.depot_name;
        end loop;
        open c2;
        raise notice "Details of Buses run on route 2 are:";
        raise notice "Bus No Capacity Depot Name";
        loop
                fetch c2 into rec2;
                exit when not found;
                raise notice "% % % ",rec2.bno,rec2.capacity,rec2.depot_name;
        end loop;
        close c1;

        close c2;
end;'
language 'plpgsql';

Lab=# select Disp_route();
```

**Example 2: Parameterized Cursor**

**Consider the following Relational Database:**

**Doctor** (d_no, d_name, d_city)
**Hospital** (h_no, h_name, h_city)
**DH (**d_no, h_no**)**

- **Display Hospital wise doctor details.**

```
create or replace function param2_cursor() returns void as'
declare
c1 cursor for select * from Hospital;
c2 cursor(hno Hospital.h_no%type) for select h_name,DH.d_no,d_name from
Hospital, Doctor, DH where Hospital.h_no=DH.h_no
and Doctor.d_no=DH.d_no and DH.h_no=hno;
rec1 Hospital %rowtype;
rec2 record;
begin
open c1;
```

```
raise notice "Hospital Name  Doctor No Doctor Name ";
loop
        fetch c1 into rec1;
        exit when not found;
        open c2(rec1.h_no);
        loop
                fetch c2 into rec2;
                exit when not found;
                raise notice "%   %  %",rec2.h_name,rec2.d_no,rec2.d_name;
        end loop;
close c2;
end loop;
close c1;
end;'
language 'plpgsql';

Lab=# select param2_cursor ();
```

## SET A
**Movie-Actor Database**

1. Write a stored function using cursor to display the names of movies released in year 2017.
2. Write a stored function using cursor to display the names of actors who have acted in more than 2 movies.

## SET B
**Student-Marks Database**

1. Write a stored function using cursor, to accept an address from the user and display the name, subject and the marks of the students staying at that address.
2. Write a stored function using cursor, to calculate total marks of each student and display it.

## SET C
**Business-Trip Database**

1. Write a stored function using cursor to list all salesman’s name, his trip expenses details.
2. Write a stored function using cursor to display the count of salesmen for each department.

## SET D
**Student-Competition Database**

1. Write a function using cursor which will list all the competitions in which 6$^{th}$ class students have won 2$^{nd}$prize.
2. Write a function using cursor to accept year and display competitions which come

under the type 'sport' for each class in given year.

**<u>SET E</u>**

**Book-Author Database**

1. Write a stored function using cursor to accept publisher name as input and display details of author and their book details for that publisher.
2. Write a stored function using cursor to find number of books written by each author.

**Assignment Evaluation**

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2:LateComplete[] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**

# Assignment No. 6 Triggers

- PL/pgSQL can be used to define trigger procedures. PostgreSQL **Triggers** are database callback functions, which are automatically invoked when a specified database event (insert, delete, and update) occurs.
- A trigger procedure is created with the **CREATE FUNCTION** command, declaring it as a function with no arguments and a return type of trigger.
- A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies.

➢ **Creating Trigger:**
**Syntax:**

```
CREATETRIGGER trigger_name
{BEFORE |AFTER} {event_ name} ON table_name
FOR EACH {ROW | STATEMENT}
EXECUTE PROCEDURE function_name (arguments);
```

**1. trigger_name:** User defined name of the trigger.
**2. before/after:** Determines whether the function is called before or after an event.
**3. event_name:** Database events can be Insert, Update and Delete.
**4. table_name:** The name of the table the trigger is for.
**5. for each row /for each statement:** specifies whether the trigger procedure should be fired once
for every row affected by the trigger event or just once per SQL statement. If neither is specified,
for each statement is the default.
**6. function_name:** Name of the function to execute.

➢ **Drop Trigger:**
**Syntax:**
DROP TRIGGER trigger_name ON table_name;

1. **trigger_name:** The name of the trigger to remove.
2. **table_name:** The name of the table for which trigger is defined.

➢ **Variables used in Trigger:** When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

| Variable Name | Description |
|---|---|
| NEW | Data type `RECORD`; variable holding the new database row for `INSERT`/`UPDATE` operations in row-level triggers. This variable is `NULL` in statement-level triggers and for `DELETE` operations. |
| OLD | Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers and for INSERT operations |
| TG_NAME | Data type name; variable that contains the name of the trigger actually fired. |
| TG_WHEN | Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition. |
| TG_LEVEL | Data type text; a string of either ROW or STATEMENT depending on the trigger's definition. |
| TG_OP | Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired. |
| TG_TABLE_NAME | Data type name; the name of the table that caused the trigger invocation. |
| TG_TABLE_SCHEMA | Data type name; the name of the schema of the table that caused the trigger invocation. |
| TG_NARGS | Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement. |
| TG_ARGV [] | Data type array of text; the arguments from the CREATE TRIGGER statement. |

**Example1:**
**Consider the following Relational Database:**
**Customer** (cno, cname, city)
**Account** (ano, acc_type, balance, cno)

- **Delete Customer record from Customer table and display message to user 'Customer record being deleted'**
  **Function:**

```
create or replace function Cust_Del()
returns trigger as'
declare
begin
        raise notice " Customer record being deleted";
        return old;
end;'
language 'plpgsql';
```

**Trigger:**

```
create trigger trig_del
before delete on Customer
for each row
execute procedure Cust_Del();
```

After creating function and trigger type delete query on terminal.

```
Lab=# delete from Customer;
```

The trigger gets fired and executes procedure/function Cust_Del() and text message displayed to the user.

**Example 2:**
**Consider the following Relational Database:**

**Dept** (dno, dname)
**Emp** (eno, ename, sal, dno)

- **The below example ensures that if employee name entered as null or employee salary entered less than zero, trigger gets fired.**

**Function:**

```
create or replace function sal_chk()
returns trigger as'
begin
        if (NEW.sal <0) then
        raise exception " Employee Salary Should not be zero";
        end if;
        if (NEW.ename IS NULL) then
        raise exception "Employee Name should not be NULL";
        end if;
        return NEW;
end;'
language plpgsql;
```

**Trigger:**

```
create trigger sal_trig
```

```
before insert or update on Emp
for each row
execute procedure sal_chk();
```

After creating function and trigger type insert query on terminal.

```
Lab=# insert into emp values (4, -2000);
Lab=# insert into emp values (4, 2000, null);
```

The trigger gets fired and executes procedure/function sal_chk() and error message displayed to the user if employee name entered as null or employee salary entered less than zero.

## <u>SET A</u>

### Student-Marks Database

1. Write a trigger before deleting a student record from the student table. Raise a notice and
   display the message "student record is being deleted".
2. Write a trigger to ensure that the marks entered for a student, with respect to a subject is never less than 0 and greater than 70.

## <u>SET B</u>

### Business-Trips Database

1. Write a trigger before inserting into an Expense table to check amount. Amount should be always<50000. Display appropriate message.
2. Write a trigger before inserting into a trip table to check departure date. 'departure date' should be always>= current date. Display appropriate message.

## <u>SET C</u>

### Student-Teacher Database

1. Write a trigger before insert the record of Student. If the sno is less than or equal to zero give the message "Invalid Number".
2. Write a trigger before update a student's s_class from student table. Display appropriate message.

## <u>SET D</u>

### Railway Reservation Database

1. Write a trigger to restrict the bogie capacity of any train to 30.
2. Write a trigger after insert on passenger to display message "Age above 5 will be charged full fare" if age of passenger is more than 5.

## <u>SET E</u>

## Warehouse Database

1. Write a trigger to ensure that weight of item should not be less than 2kg and display the message “Weight is Less than 2 kg”.
2. Write a trigger before deleting a customer record from the customer table. Raise a notice and display the message “Customer record is being deleted”.

### Assignment Evaluation

| | | |
|---|---|---|
| 0: Not Done [ ] | 1: Incomplete [ ] | 2:LateComplete[] |
| 3: Needs Improvement [ ] | 4: Complete [ ] | 5: Well Done [ ] |

**Signature of Instructor**