|  |
| --- |
| **GUIDELINES** |
| 1. **Use only structures for all the Part – A programs** 2. **GOTO statements are not allowed.** |
| 1. **No global declarations allowed.** |
| 1. **Prototype for each user-defined-function must be provided before main.** |
| 1. **Students are encouraged to use user-defined-header files.** |
| 1. **Programs must be indented appropriately.** |
| 1. **Stick all datasheets together in the first sheet of each program.**   ***Note:***   1. ***Students have to come prepared for the lab programs before lab schedule and have to write the data sheet in the lab time itself. Only then students will be allowed to execute the program.*** 2. ***Students have to come prepared for the viva-voice every lab, which carries 4 marks for each program.*** 3. ***Stick data sheets in the space provided.***   ***Instructions for Part- B***   * ***Part- B should be documented in the space provided in the manual.*** * ***Complete Problem statement in the first week*** * ***Selection of data structure and definition – next 2 weeks*** * ***Design and partial implementation – next 2 weeks*** * ***Coding and output - next 2 weeks.*** |
| **Evaluation:**  **The Total CIE marks will be evaluated as follows: 30 Marks Part A (record) + 10 Marks Part B (record) + 10 marks Internals = 50 (Total)** |
|  |

***Stick Data sheets of Program-1 here:***

**PROGRAM – 1**

**Use stack operations to do the following:**

1. **Assign to a variable name Y the value of the third element from top of the stack and keep the stack undisturbed.**
2. **Given an arbitrary integer n pop out the top n elements. A message should be displayed if an unusual condition is encountered.**
3. **Assign to a variable name Y the value of the third element from the bottom of the stack and keep the stack undisturbed.**

**Algorithm pushstack(stack,data)**

***Insert an element into the stack.***

***Pre: data to be inserted into the stack.***

***Post: successful insertion of data into the stack.***

Check if the stack is full

Else increment the count and insert the element

**End pushstack**

**Algorithm popStack(stack)**

***Pre: Pops the element on the top of the stack and returns it to the user.***

***Post: data popped out of stack.***

If(stack empty)

Set success to false and return

Else

Set dataout as data in stack top node

Decrement the stack top

Decrement the stack count

End if

Return dataout

**End popStack**

**Algorithm Third\_element\_from\_top( )**

***Pre: Stack with minimum three elements***

***Post: displays third element from the top.***

Loop(for each i=0 to i<2)

Set x as popStack(stack)

pushStack(temp,x) //push top 2 elements into the temporary stack

End loop

Set Y as popStack(stack) // 3ed element from stack

Display the element Y

pushStack(stack,Y) //push the elements back to the stack

Loop(for each i=0 to i<2)

Set x as popStack(temp)

pushStack(stack,x) //push top 2 elements into the temporary stack

End loop

**End Third\_element\_from\_top**

**Algorithm Third\_element\_from\_the\_bottom( )**

***Pre: stack with minimum three elements***

***Post: third element from bottom***

Loop(until stack not empty)

Temdata=popStack(stack)

pushStack(temp,temdata)

end loop

loop(temp top not equal to 2)

Temdata=popStack(temp)

pushStack(stack, temdata)

end loop

Y=popstack(temp)

Y will be the third element from bottom

Push Y to stack

Loop(until temp not empty)

Temdata=popStack(temp)

pushStack(stack,temdata)

end loop

**End Third\_element\_from\_the\_bottom**

**Algorithm popNelements(int n)**

***Pre: Non empty stack with minimum n elements***

***Post: Pops out top N elements from stack***

i=0

Loop(until stack is not empty i<=n))

popStack(stack) and display the element

increment i

end loop

**End popNelements**

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-2 here:***

**PROGRAM -2**

**Write a C program that parses Infix arithmetic expressions to Postfix arithmetic expressions using a Stack.**

An algebraic expression is a legal combination of operators and operands. Operand is quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, \*, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

**Infix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) \* (C - D)

**Prefix:** It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation

Example: \* + A B – C D

**Postfix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B + C D - \*

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.

2. The parentheses are not needed to designate the expression un-ambiguously.

3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, \*, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR PRECEDENCE VALUE

Exponentiation ($ or ↑ or ^) Highest 3

\*, / Next highest 2

+, - Lowest 1

**Algorithm to Conversion from infix to postfix:**

1. Scan the infix expression from left to right.

2. a) If the scanned symbol is left parenthesis, push it onto the stack.

b) If the scanned symbol is an operand, then place it directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, - if the stack is empty or top of stack has left parenthesis push the operator or else if stack is not empty nor the top of stack is not left parenthesis, if the precendence of top of stack is greater than or equal to precedence of the scanned symbol , keep popping the stack content and place them on to postfix and finally push the operator on to the stack.

3. Pop the content of stack and place it onto the postfix expression until stack becomes empty.

**Example:**

Convert ((A – (B + C)) \* D) ↑ (E + F) infix expression to postfix form:

SYMBOL POSTFIX STRING STACK REMARKS

( (

( ( (

A A ( (

- A ( ( -

( A ( ( - (

B A B ( ( - (

+ A B ( ( - ( +

C A B C ( ( - ( +

) A B C + ( ( -

) A B C + - (

\* A B C + - ( \*

D A B C + - D ( \*

) A B C + - D \*

↑ A B C + - D \* ↑

( A B C + - D \* ↑ (

E A B C + - D \* E ↑ (

+ A B C + - D \* E ↑ ( +

F A B C + - D \* E F ↑ ( +

) A B C + - D \* E F + ↑

End of The input is now empty. Pop the output symbols

string A B C + - D \* E F + ↑ from the stack until it is empty.

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-3 here:***

**PROGRAM 3**

**Write a C program to simulate the working of Messaging System in which a message is placed in a circular Queue by a Message Sender, a message is removed from the circular queue by a Message Receiver, which can also display the contents of the Queue.**

**To implement a messaging system, a circular queue data structure can be used, we first perform the following steps before we implement actual operations.**

**Initialization**

**Step 1: Define a constant ‘SIZE’ which specifies the size of the queue and ‘MSIZE’ which specifies the size of a message with specific values.**

**Step 2: Create a two dimensional array of the char data type to hold the messages (char CQueue[ SIZE][MSIZE] )**

**Step 3: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)**

**enQueue(message) - Inserting message into the Circular Queue**

**In a circular queue, enQueue() is a function which is used to insert an message into the circular queue. In a circular queue, the new messaege is always inserted at rear position. The enQueue() function takes string as parameter and inserts that message into the circular queue. We can use the following steps to insert an message into the circular queue...**

**Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))**

**Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.**

**Step 3: If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.**

**Step 4: Increment rear value by one (rear++), set CQueue[rear] = message and check 'front == -1' if it is TRUE, then set front = 0.**

**deQueue() - Deleting a message from the Circular Queue**

**In a circular queue, deQueue() is a function used to delete a message from the circular queue. In a circular queue, the message is always deleted from front position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...**

**Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)**

**Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.**

**Step 3: If it is NOT EMPTY, then display CQueue[front] as deleted message and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).**

**display() - Displays the messages in a Circular Queue**

**We can use the following steps to display the elements of a circular queue...**

**Step 1: Check whether queue is EMPTY. (front == -1)**

**Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.**

**Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.**

**Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.**

**Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i <= SIZE - 1' becomes FALSE.**

**Step 6: Set i to 0.**

**Step 7: Again display 'CQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.**

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program 4 here:***

**PROGRAM – 4**

|  |
| --- |
| **Implement a program to multiply two polynomials using single linked list.** |

**Algorithm poly\_mult(struct node \*p1, struct node \*p2)  
*Multiplies two polynomials***

***Pre: polynomials to be multiplied***

***Post: resultant polynomial***

Set resultant polynomial result as null initially

Check if(p1 is NULL or p2 is NULL)

Multiplied polynomial is zero polynomial  
      End

End if

Loop(until p1 is not null)  
       Set p2 as p2\_beg  
        Loop(until p2 is not null)

result=insert(result,p1’s coef \* p2’s coef,p1’s expo + p2’s expo)  
            Set p2 to point to next node      
        End loop  
        Set p1 to point to next node

End loop  
result gives the multiplication of two polynomails p1 and p2

**End poly\_mult**

**Algorithm insert\_s(result,floatco,int ex)**  
    allocate memory dynamically for tmp  
    set tmp of coef as co  
    set tmp of expo as ex  
     /\*list empty or exp greater than first one \*/  
    check if(start is NULL or ex is greater than result of expo)

Set temp of next node as resultant  
        Set result as temp

else      
        Set ptr as result  
        Loop(until ptr is not null and ptr’s next nodes expo is >= ex)

until condition satisfies traverse ptr

Set tmp of link as ptr of link   
        ptr of link as tmp  
End if     
return the result

**End insert**

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-5 here:***

**PROGRAM – 5**

**Write a C program to implement addition of long positive integers using circular single linked list with header node.**

**Algorithm add\_longint(struct node \*h1, struct node \*h2)  
*addition of two long integers***

***Pre: two long integers pointed by h1 and h2.***

***Post: resultant addition pointed by h3***

struct node \*c, \*c1, \*c2;  
carry = 0;  
  
set c2 to point to h2  
  
loop until (c1 != h1 && c2 != h2)  
{  
 sum = add the value pointed by c1 and c2 and previous carry   
 digit = sum % 10;  
 carry = sum / 10;  
 h3 = insert\_begin(digit, h3);  
 set c1 to point to next node

set c2 to point to next node  
}  
  
check if(c1 != h1) //un equal lists  
{  
 c = c1;  
 h = h1;  
}  
else  
{  
 c = c2;   
 h = h2;  
}  
loop until (c != h)  
{  
 sum = add the value and carry  
 digit = sum % 10;  
 carry = sum / 10;  
 h3 = insert\_begin(digit, h3);  
 set c2 to point to next node  
}  
if(carry==1)  
{  
 h3 = insert\_begin(carry, h3);  
}  
return(h3);  
}

**Algorithm store\_into\_list(longinteger i)  
*stores the long integer in linked list***

***Pre: long integers i***

***Post: resultant linked list h***

*Loop until( i! = 0 )*

*{*

*digit=i%10*

*i=i/10*

*h=insert\_end(h,digit)*

*}*

*Return(h)*

*}*

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-6 here:***

**PROGRAM – 6**

**Design a doubly linked list to represent sparse matrix. Each node in the list can have the row and column index of the matrix element and the value of the element. Print the complete matrix as the output.**

**Step 1:** START

**Step 2:** Create a doubly linked list to store a matrix with index.

**Step 3:** Scan the values for number of rows and columns.

**Step 4:** Scan the values for matrix with respect to rows and columns. If the value is not zero then allocate memory dynamically for the element and insert the element as the last node of the doubly linked list.

**Step 6:** Create a function to display the contents of the doubly linked list. If the list is empty display message else traverse through the list and display the list contents.

**Step 7:** Create a function to display matrix. For each row and column check the index value in the list, if the index is their display the value else display zero for that row and column.

**Step 8:** END.

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-7 here:***

**PROGRAM – 7**

|  |
| --- |
| **Write a C program to create Binary Tree and provide insertion and deletion operations and to traverse the tree using In-order, Preorder and Post order (recursively)** |

**Algorithm insertBT()**

***Inserts node containing new data into BT using recursion***

***Pre: root is address of current node in a BT***

***Newnode is address of node containing data***

***Post: newnode inserted into the tree***

If(empty tree)

Set root to newnode

Return newnode

End if

//Locate null subtree for insertion

If(newnode<root)

Return insertBT(left subtree, newnode)

Else

Return insertBT(right subtree,newnode)

End if

**End insertBT**

**Algorithm deleteBT(root,deletekey)**

***This algorithm deletes a node from a Binary tree.***

***Pre: root is reference to node to be deleted***

***deletekey is key of node to be deleted***

***Post: node deleted***

***If deletekey not found, root unchanged***

***Return true if node deleted, false if not found***

If(empty tree)

Return false

End if

If(deletekey< root)

Return deleteBT(left subtree,deletekey)

Else if(deletekey> root)

Return deleteBT(right subtree,deletekey)

Else

// Delete node found –test for leaf node.

If(no left subtree)

Make right subtree the root

Return true

Else if(no right subtree)

Make left subtree the root

Return true

Else

Node to be deleted not a leaf. Find largest node on the left subtree.

Save root in deleteNode

Set largest to largestBT(left subtree)

Move data in largest to deletenode

Return deleteBT(left subtree of deletenode,key of largest)

End if

End if

**End deleteBT**

**Algorithm Preorder(root)**

***Traverse a binary tree in node-left-right sequence.***

***Pre: root is the entry node of a tree or subtree***

***Post: each node has been processed in order***

If(root is not null)

Process(root)

Preorder(leftSubtree)

Preorder(rightSubtree)

End if

**End Preorder**

**Algorithm Inorder(root)**

***Traverse a binary tree in left-node-right sequence.***

***Pre: root is the entry node of a tree or subtree***

***Post: each node has been processed in order***

If(root is not null)

Inorder(leftSubtree)

Process(root)

Inorder(rightSubtree)

End if

**End Inorder**

**Algorithm Postorder(root)**

***Traverse a binary tree in left-right-node sequence.***

***Pre: root is the entry node of a tree or subtree***

***Post: each node has been processed in order***

If(root is not null)

Postorder(leftSubtree)

Postorder(rightSubtree)

Process(root)

End if

**End Postorder**

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-8 here:***

**PROGRAM – 8**

**Given a String representing a parentheses-free infix arithmetic expression, implement a program to place it in a tree in the infix form. Assume that a variable name is a single letter. Traverse the tree to produce an equivalent postfix and prefix expression string.**

**Algorithm priority(char operator)**

**Returns the priority of operator**

***Pre: Operator to be checked for priority***

***Post: Operator priority will sent back***

Check for the priority of operator

If(operator is equal to ^ )

Priority is high

Else if(operator is equal to / or \*)

Priority is medium

Else if(operator is equal to + or /)

Priority is low

End if

**End priority**

**Algorithm insert()**

***Inserts the node in the tree.***

***Pre: Element to be inserted.***

***Post: Element will be inserted at proper position in the tree.***

Constructing an expression tree involves two parts:

Lexical Analysis: Dividing the input into tokens, each of which represents either integer constant, an operator or a variable name.

Parsing: Determining if the individual tokens represent a legal expression and finding out the structure of that expression.

**End insert**

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-9 here:***

**PROGRAM – 9**

**Write a C program to implement Hashing using Linear probing. Implement insertion, deletion, search and display.**

**Algorithm open\_addressing(num)**

***Stores the number using hashing technique***

***Pre: Number to be hashed***

***Post: Number successfully stored using hashing or not.***

Use array to store hashing or use any data structure

Assign initial value of -1 for all elements in the array

Generate hash key for the number (ex: num%10)

Check if(array[key] is equal to -1) //free space

Set array[key] as num

Else Check if array is full

If full exit from algorithm

End if

Loop(for each key+1 to max) //linear probing technique

If(array[key] is -1)

Set array[key] as num

exit from algorithm

End if

End loop

Loop(for each i=0 to i<key) //search all the positions less than the key

Check if(array[i] is -1)

Set array[i] as num

exit from algorithm

End if

End loop

**End open\_addressing**

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***Stick Data sheets of Program-10 here:***

**PROGRAM - 10**

**Write a C program to implement priority queue to insert, delete and display the elements.**

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the highest value of key is at front and item with the lowest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority.

Priority queue can be implemented using a heap. A heap is a specific tree based data structure in which all the nodes of tree are in a specific order. Let’s say if X is a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree. There can be two types of heap:

**Max Heap:** In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

**Min Heap:** In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.

Implementation of priority queue using heap has 2 steps:

1. Create a max heap for a set of N elements.
2. Extract max from heap.
3. **Create a max heap for a set of N elements.**

Now let’s say we have N elements stored in the array Arr indexed from 1 to N. They are currently not following the property of max heap. So we can use max-heapify function to make a max heap out of the array.

void max\_heapify (int Arr[ ], int i)

{

int left = 2\*i //left child

int right = 2\*i +1 //right child

if(left<= N and Arr[left] > Arr[i] )

largest = left;

else

largest = i;

if(right <= N and Arr[right] > Arr[largest] )

largest = right;

if(largest != i )

{

swap (Ar[i] , Arr[largest]);

max\_heapify (Arr, largest,N);

}

}

From the above property we observed that elements from Arr[ N/2+1 ] to Arr[ N ] are leaf nodes, and each node is a 1 element heap. We can use max\_heapify function in a bottom up manner on remaining nodes, so that we can cover each node of tree.

void build\_maxheap (int Arr[ ])

{

for(int i = N/2 ; i >= 1 ; i-- )

{

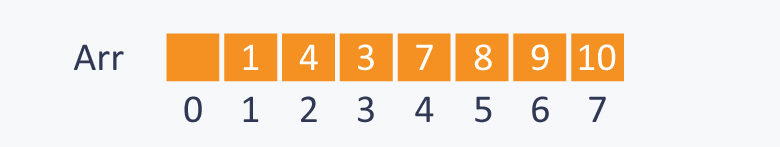
max\_heapify (Arr, i) ;

}

}

**Example:**

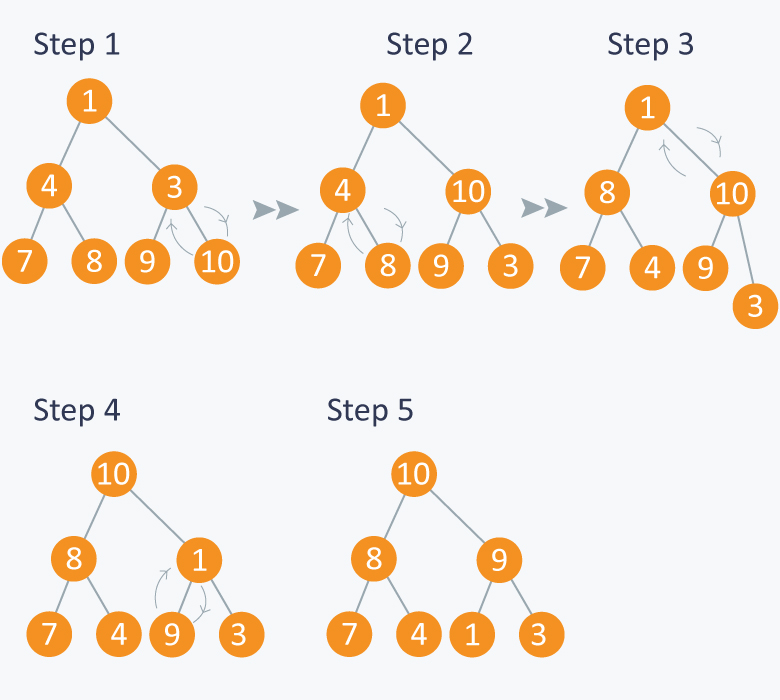
Suppose you have 7 elements stored in array Arr.



Here N = 7, so starting from node having index N/2 = 3, (also having value 3 in the above diagram), we will call max\_heapify from index N/2 to 1.

In the diagram below:

In step 1, in max\_heapify(Arr, 3), as 10 is greater than 3, 3 and 10 are swapped and further call to max\_heap(Arr, 7) will have no effect as 3 is a leaf node now.

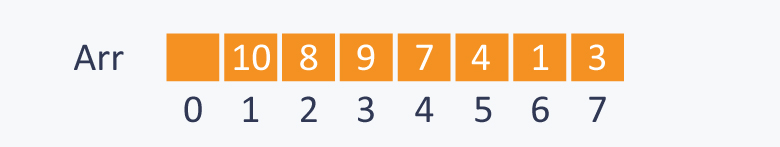


**In step 2, calling max\_heapify(Arr, 2) , (node indexed with 2 has value 4) , 4 is swapped with 8 and further call to max\_heap(Arr, 5) will have no effect, as 4 is a leaf node now.**

**In step 3, calling max\_heapify(Arr, 1) , (node indexed with 1 has value 1 ), 1 is swapped with 10**

**Step 4 is a subpart of step 3, as after swapping 1 with 10, again a recursive call of max\_heapify(Arr, 3) will be performed , and 1 will be swapped with 9. Now further call to max\_heapify(Arr, 7) will have no effect, as 1 is a leaf node now.**

**In step 5, we finally get a max- heap and the elements in the array Arr will be :**



1. Extract max from heap

**extract\_maximum (Arr) - It removes and return the maximum element from the Arr, which is a heap.**

**In this operation, the maximum element will be returned and the last element of heap will be placed at index 1 and max\_heapify will be performed on node 1 as placing last element on index 1 will violate the property of max-heap.**

int extract\_maximum (int Arr[ ])

{

if(length == 0)

{

Printf(“\n “Can’t remove element as queue is empty”);

return -1;

}

int max = Arr[1];

Arr[1] = Arr[length];

length = length -1;

max\_heapify(Arr, 1);

return max;

}

**OUTPUT:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sl. No** | **Criteria** | **Measuring methods** | **Excellent** | **Good** | **Poor** |  |
| 1 | **Understanding of problem statement and design.**  **(2 Marks)**  **CO1,CO2** | Observations | Student exhibits thorough understanding of requirements and applies suitable logic and programming concepts for the problem  **(2 M)** | Student has sufficient understanding of requirements and applies suitable logic and programming concepts for the problem.  **(<2 M and >=1 M)** | Student does not have a clear understanding of requirements and is unable to apply suitable logic and programming concept for the problem.    **(0 M)** |  |
| 2 | **Execution and debugging**  **(2 Marks)**    **CO4** | Observations | Student demonstrates the execution of the program with efficient code. Appropriate validations with all test cases are handled.  **(2 M)** | Student demonstrates the execution of the program without efficiency of the code and validates only few cases.      **(1 M)** | Student has not executed the program.    **(0 M)** |  |
| 3 | **Results and Documentation**  **(2 Marks)**  **CO2** | Observations | Documentation with appropriate comments and output is covered in data sheets and manual.  **(2 M)** | Documentation with only few comments and only few output cases is covered in data sheets and manual.  **(1 M)** | Documentation with no comments and no output cases is covered in data sheets and manual.  **(0 M)** |  |
| **Viva Voce rubrics (Max: 4 marks)** | | | | | |  |
| 1 | **Conceptual Understanding**  **(2 Marks)**  **CO1** | Viva Voce | Explains thoroughly the programming concepts and the data structure.  **(2 M)** | Adequately explains the programming concepts and data structures.    **(1 M)** | Unable to explain the programming concepts and data structure.      **(0 M)** |  |
| 2 | **Use of data structure**  **(1 Marks)**  **CO4** | Viva Voce | Insightful explanation of data structure design technique for the given problem to derive solution.  **(1 M)** | Sufficiently explains the use of appropriate data structure design technique for the given problem to derive solution.  **(0.5 M)** | Unable to explain the data structure design technique for the given problem.  **(0 M)** |  |
| 3 | **Communication of Concepts**  **(1 Marks)**  **CO4** | Viva Voce | Communicates the concept used in problem solving well.  **(1 M)** | Sufficiently communicates the concepts used in problem solving.  **(0.5 M)** | Unable to communicate the concepts used in problem.  **(0 M)** |  |
| **Record Marks (6)** | | **Viva Marks (4)** | **Total (10)** | **Signature of Staff 1** | **Signature of Staff 2** | |
|  | |  |  |  |  | |

***PART – B***

**Problem Statement:**

**Signature of the Staff**

**Introduction to the problem**

**Relevant Data Structure used with justification**

**Design of the problem (Draw system diagram or system architecture diagram and explain)**

**Source code (if space is not sufficient, stick data sheets)**

**Output (all possible output)**

**Part B Evaluation Rubrics**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Components** |  | | | **Marks Allocated** |
|  | **Excellent** | **Good** | **Not Satisfactory** |  |
| **Problem definition(1)** | Exhibit Clear understanding of the problem and requires various data structures to bring out the solution  1 marks | Exhibit understanding of the program but requires simple data structures to bring out the solution.  < 1 marks | Not Clear about the program- 0 Marks |  |
| **Application of relevant data structure with justification**  **(1)** | Relevant Data Structure used with correct justification  1 marks | Not proper data structure used and improper justification  <1 marks | No justification 0 marks |  |
| **Incorporation of suggestions (1)** | Changes are made as per modification suggested during evaluation and new innovations added.  (1) | Changes are made as per modifications suggested during evaluation and good justification.  <1 marks | Suggested changes not incorporated  0 marks |  |
| **Design (1)** | Analyze and draw a system diagram for the problem  1 marks | Diagram not proper  <1 marks | No design  0 |  |
| **Source Code (2)** | Coded with proper coding guide lines and all the cases handled (3) | Coded with coding guidelines but does not cover all the cases  <3 and >=1 | Coded without any coding guide lines and does not handle all the cases  <1 marks |  |
| **Demonstration of Output (2)** | Demonstrated correct output satisfying all the cases. –2marks | Demonstrated partial output. - <=1 | Not clear output. -0 marks |  |
| **Documentation (1)** | Correctly documented with all the result – 2 marks | Partial documentation - <=1 marks | No documentation – 0 marks |  |
| **Total** |  | | |  |
| **Signature of the Staff** |  | |  | |