Data Structure Assignment

Algorithms

Q1. Binary Search (Insertion, deletion).

Binary Search Algorithm:

```
1.  [Initialize segment variables.]
    Set BEG:= LB, END := UB and MID = INT(BEG + END)/2).
2.  Repeat Steps 3 and 4 while BEG <= END and DATA[MID] != ITEM.
3.  If ITEM < DATA[MID], then:
    Set END:= MID - 1.
    Else:
    Set BEG := MID + 1.
    [End of If structure.]
4.  Set MID:= INT(BEG + END)/2).
    [End of Step 2 loop.]
5.  If DATA[MID = ITEM, then:
    Set LOC= MID.
    Else:
    Set LOC:= NULL.
    [End of If structure.]
6.  Exit.
```

Insertion Algorithm:

```
1. [Initialize variables.]
     • Set ITEM to the element you want to insert.
     • Set BEG to the lower bound (LB).
     • Set END to the upper bound (UB).
2. Repeat Steps 3 and 4 while BEG is less than or equal to END:
3. Set MID to the integer value of (BEG + END) / 2.
4. If ITEM is equal to DATA[MID], insert ITEM at the position MID and shift the elements to
   the right to make space for the new item. Then, set LOC to MID.
5. If ITEM is less than DATA[MID], update END to be MID - 1 to search in the lower half.
6. If ITEM is greater than DATA[MID], update BEG to be MID + 1 to search in the upper half.
7. Exit.
```

Deletion Algorithm:

1. [Initialize variables.]
   - Set ITEM to the element you want to delete.
   - Set BEG to the lower bound (LB).
   - Set END to the upper bound (UB).
2. Repeat Steps 3 and 4 while BEG is less than or equal to END:
3. Set MID to the integer value of (BEG + END) / 2.
4. If ITEM is equal to DATA[MID]:
   - Delete DATA[MID].
   - Shift the elements to the left to fill the gap created by the deleted item.
   - Set LOC to MID.
5. If ITEM is less than DATA[MID], update END to be MID - 1 to search in the lower half.
6. If ITEM is greater than DATA[MID], update BEG to be MID + 1 to search in the upper half.
7. Exit.

Q2. Singly Linked List (Insert in first and Insert in last).

Insert in last

```
1. Allocate memory to a node FRESH.
2. Check if FRESH == NULL, then:
    Print "NO MEMORY RECEIVED" and return.

3. Set NAME[FRESH] := DATA_NAME.
    Set ROLL[FRESH] := DATA_ROLL.
    Set N[FRESH] := NULL.

4. Check if START = NULL, then: [If linked list is absent]
    Set START := FRESH.

5. Else [If linked list is present]
        (a)   Set LOC := START.
        (b)   Repeat while N[LOC] != NULL.
              Set LOC := N[LOC].
        [End of while loop of step (a)]
        (c)   Set N[LOC] := FRESH.
    [End of step 4 if structure]
6. Exit.
```

Q3. Deletion in Linked list (deletion first and deletion last).

<u>**Algorithm On Deletion Of First Node From Linked List.**</u>

DELETE_BEGIN()

```
1. Check if START = NULL, then: [If linked list is absent]
   Print "THE LIST IS EMPTY." and return.
```

```
2. Else [If linked list is present]
       (a)  Set LOC := START.
       (b)  Set START := N[LOC].
       (c)  Free the memory occupied by LOC.
   [End of step 1 if structure]
3. Exit
```

<u>Algorithm On Deletion Of Last Node From Linked List.</u>

```
DELETE_LAST()
1. Check if START = NULL, then: [If linked list is absent]
   Print "THE LIST IS EMPTY." and return.

2. Else [If linked list is present]
       (a)  Set LOC := START.
       (b)  Repeat while LOC != NULL.
            Set PLOC := LOC and LOC := N[LOC]._
       [End of while loop of step (a)]
       (c)  Set N[PLOC] := NULL.
       (d)  Free memory occupied by LOC.
   [End of step 1 if structure]
3. Exit.
```

Q4. Doubly Linked list (Insertion and Display)

```
1. Allocate memory to a node FRESH.
2. Check if FRESH == NULL, then:
   Print "NO MEMORY RECEIVED" and return.

3. Set NAME[FRESH] := DATA_NAME.
   Set ROLL[FRESH] := DATA_ROLL.
   Set NXT[FRESH] := NULL.
   Set PRV[FRESH] := NULL.

4. Check if HEAD = NULL and TAIL = NULL , then:
       Set HEAD := FRESH and TAIL := FRESH.

5. Else [If linked list is present]
       (a)  Set PRV[FRESH] := TAIL.
       (b)  Set NXT[TAIL] := FRESH.
       (c)  Set TAIL := FRESH.
   [End of step 4 if structure]
6. Exit.
```

<u>Algorithm To Display Doubly Linked List (Forward)</u>

```
1. Repeat steps 2 and 3 while HEAD != NULL.
2. Apply process to ROLL[HEAD],NAME[HEAD].
3. Set HEAD := NXT[HEAD].
[End of step 1 while loop]
4. Exit.
```

Q5. Circular Linked list (Insertion and Display)

```
1. Allocate memory to a node FRESH.
2. Check if FRESH == NULL, then:
   Print "NO MEMORY RECEIVED" and return.

3. Set NAME[FRESH] := DATA_NAME.
   Set ROLL[FRESH] := DATA_ROLL.
   Set N[FRESH] := NULL.

4. Check if START = NULL, then: [If linked list is absent]
   Set N[FRESH] : FRESH and START := FRESH.

5. Else [If linked list is present]
        (a)  Set LOC := START.
        (b)  Repeat while N[LOC] != START.
             Set LOC := N[LOC].
        [End of while loop of step (a)]
        (c)  Set N[FRESH] := START and N[LOC] := FRESH.
   [End of step 4 if structure]
6. Exit.
```

Q6. Stack using array (push, pop and display)

```
PUSH(int ele)_
The new data to be added in the stack is represented by 'ele'.
   1. Check if top = full, then:
      Print "OVERFLOW ! NO DATA CAN BE INSERTED" and return.
   2. Else set top := top + 1 and set top[stack] := ele.
      [End of if structure in step 1]
   3. Exit.
POP()
        (d)  Check if top = -1, then:
      Print "UNDERFLOW ! NO DATA TO DELETE" and return.
        (e)  Else set top := top - 1
      [End of if structure in step 1]
        (f)  Exit.

Function DisplayStackLinkedList(STACK):
1. Initialize a pointer (PTR) to the top of the stack and set it
to STACK's top node.
2. If PTR is NULL, exit (the stack is empty).
3. While PTR is not NULL:
   a. Print the data in the node pointed to by PTR.
   b. Move PTR to the next node in the stack.
4. Exit.
```

Q7. Stack using Linked list (push, pop and display)

```
PUSH(int ele)
```

Here pointer 'TOP' points to the node at top position, i.e. where the insertion is to take place.

```
  1. Allocate memory to a node FRESH.
  2. Check if FRESH = NULL, then:
     Print "NO MEMORY RECEIVED" and return.

  3. Set INFO[FRESH] := ITEM.
     Set N[FRESH] := NULL.

  4. Check if TOP = NULL, then: [If stack linked list is absent]
      Set TOP := FRESH.

  5. Else [If stack linked list is present]
   7. Set LOC := TOP
   8. Set TOP := FRESH.
   9. Set N[FRESH] := LOC.
      [End of step 4 if structure]
  6. Exit.
  7.
POP()
   1. Check if TOP = NULL, then: [If stack linked list is absent]
      Print "UNDERFLOW, THE STACK IS EMPTY." and return.

   2. Else [If stack linked list is present]
          (d)   Set PTR := TOP.
          (e)   Set TOP := N[TOP].
          (f)   Free the memory pointed by PTR.
      [End of step 1 if structure]
   3. Exit.

Function DisplayStackArray(STACK, TOP, MAX_SIZE):
1. Initialize an index variable (IDX) to 0.
2. If TOP is -1 (indicating an empty stack), exit.
3. While IDX is less than or equal to TOP:
   a. Print the element in STACK at index IDX.
   b. Increment IDX by 1.
4. Exit.
```

Q8. Queue using array (Insert, Delete and Display)

```
INSERT_IN_QUEUE(int ele)
Here the name of the array is "queue", with 10 elements. Value of
full is set to 10 and the starting position of the "queue" is 0.
Initially "front" and "rear" is set to -1.
1. Check if rear = full, then:
     Print "OVERFLOW ! NO DATA CAN BE INSERTED " and return.
2. Else Check if rear = -1, then:
     Set front := 0 and rear := 0 and goto step 4.
3. Else
     Set rear := rear + 1 goto step 4.
     [End of if structure of step 1]
4. Set rear[queue] := ele.
5. Exit.
```

```
DELETE_FROM_QUEUE()
  1. Check if front = -1,then:
     Printf "UNDERFLOW ! EMPTY QUEUE" and return.
  2. Else
        a) Check if front = rear,then:
           Set front = -1 and rear = -1.
        b) Else
            Set front++.
        [End of if structure of step a]
     [End of if structure of step 1]
  3. Exit.
```

```
     Function DisplayQueueArray(QUEUE, FRONT, REAR, MAX_SIZE):
     1. Initialize an index variable (IDX) to FRONT.
     2. If FRONT is -1 (indicating an empty queue), exit.
     3. While IDX is less than or equal to REAR:
        a. Print the element in QUEUE at index IDX.
        b. Increment IDX by 1, wrapping around if necessary (e.g.,
     to 0 if it reaches the end of the array).
     4. Exit.
```

Q9. Queue using Linked list (Insert, Delete and Display)

```
INSERTION(int ele)
Here pointer ‘FRONT’ points to the node at first position and
“REAR” points to last node of the queue.
  1. Allocate memory to a node FRESH.
  2. Check if FRESH = NULL, then:
     Print "NO MEMORY RECEIVED" and return.
  3. Set INFO[FRESH] := ITEM.
     Set N[FRESH] := NULL.
  4. Check if FRONT = NULL, then: [If queue linked list is absent]
     Set FRONT := FRESH and REAR := FRESH.
  5. Else [If queue linked list is present]
     Set N[REAR] := FRESH and REAR := FRESH.
  6. Exit.
```

```
DELETE_FROM_QUEUE()
     1. Check if FRONT = NULL, then:
     Print "UNDERFLOW ! NO DATA PRESENT IN QUEUE" and return.
     2. Else
        (g)  Set PTR := FRONT.
        (h)  Set FRONT := N[FRONT];
        (i)  free the memory pointed by PTR.
     3. Exit.
```

```
Function DisplayQueueLinkedList(QUEUE):
1. Initialize a pointer (FRONT_PTR) to the front of the queue and
set it to QUEUE's front node.
2. If FRONT_PTR is NULL, exit (the queue is empty).
3. While FRONT_PTR is not NULL:
   a. Print the data in the node pointed to by FRONT_PTR.
   b. Move FRONT_PTR to the next node in the queue.
4. Exit.
```

Q10. Circular Queue (Insert, Delete and Display)

```
INITIALIZE_QUEUE()
   1. Repeat for i=0 to MAX:
      Set arr[i] := 0.
    [End of for loop of step 1]
   2. Exit.

ADD_ELEMENT(front, rear, ele)
   5. Check if((rear=MAX-1 and front=0) or (rear+1=front)), then:
      Print "OVERFLOW ! QUEUE IS FULL" and return.
      [End of if structure of step 1]
   6. Check if rear=MAX-1, then:
           a) Set rear := 0.
   7. Else if rear != MAX-1, then:
           b) Check if rear=-1, then: set rear := 0 and front := 0.
           c) Else set rear := rear + 1.
          [End of if structure of step b]
     [End of if structure of step 2]
   8. Set rear[arr] := ele.
   9. Exit.

DELETE_ELEMENT(front, rear)
   1. Check if front = -1, then:
      Print "UNDERFLOW ! QUEUE IS EMPTY" and return.
   [End of if structure of step 1]
   2. Check if front = rear, then: set front := -1 and rear := -1.
   3. Else if front != rear, then:
   7. Check if front = MAX-1, then: set front := 0.
   8. Else set front = front + 1.
      [End of if structure of step a]
  [End of if structure of step 2]

Function DisplayCircularQueueArray(CQUEUE, FRONT, REAR, MAX_SIZE):
1. If FRONT is -1 (indicating an empty circular queue), exit.
2. Initialize an index variable (IDX) to FRONT.
3. While True:
   a. Print the element in CQUEUE at index IDX.
   b. Increment IDX by 1, wrapping around to 0 if it reaches the
end of the array.
   c. If IDX is equal to (REAR + 1) modulo MAX_SIZE (indicating
we've come back to the starting point), break the loop. 4. Exit.
```

Q11. Infix, prefix, postfix

```
Function EvaluateInfixExpression(INFIX):
1. Initialize an empty stack for operators (OPERATOR_STACK).
2. Initialize an empty stack for operands (OPERAND_STACK).
3. Iterate through each symbol in the INFIX expression from left to right:
   a. If the symbol is an operand, push it onto OPERAND_STACK.
   b. If the symbol is an open parenthesis '(', push it onto OPERATOR_STACK.
   c. If the symbol is a closing parenthesis ')':
      i. Pop operators from OPERATOR_STACK and apply them to operands from
OPERAND_STACK until an open parenthesis is encountered.
      ii. Discard the open parenthesis.
   d. If the symbol is an operator:
      i. While the top of OPERATOR_STACK has higher or equal precedence and is left-associative,
pop the operator and apply it to the operands from OPERAND_STACK.
      ii. Push the current operator onto OPERATOR_STACK.
4. After processing all symbols, pop and apply any remaining operators on the stacks until both
stacks are empty.
5. The result is the value remaining on OPERAND_STACK, which is the final result of the
expression.
6. Exit.
```

```
Function EvaluatePrefixExpression(PREFIX):
1. Initialize an empty stack (OPERAND_STACK).
2. Reverse the PREFIX expression and iterate through it from left to right:
   a. If the symbol is an operand, push it onto OPERAND_STACK.
   b. If the symbol is an operator:
      i. Pop the top two operands from OPERAND_STACK.
      ii. Apply the operator to these operands.
      iii. Push the result back onto OPERAND_STACK.
3. The result is the value remaining on OPERAND_STACK, which is the final result of the
expression.
4. Exit.
```

```
Function EvaluatePostfixExpression(POSTFIX):
1. Initialize an empty stack (OPERAND_STACK).
2. Iterate through each symbol in the POSTFIX expression from left to right:
   a. If the symbol is an operand, push it onto OPERAND_STACK.
   b. If the symbol is an operator:
      i. Pop the top two operands from OPERAND_STACK.
      ii. Apply the operator to these operands.
      iii. Push the result back onto OPERAND_STACK.
3. The result is the value remaining on OPERAND_STACK, which is the final result of the
expression.
4. Exit.
```

Q12. Sparse matrix (using array and linked list)

A sparse matrix is a mathematical or computational matrix in which most of the elements are zero. In contrast to a dense matrix, which has a significant number of non-zero elements, a sparse matrix is characterized by its large number of zero elements relative to its total size.

Algorithm for Handling Sparse Matrix

1. Define a structure `sparse` with fields for row, column, and element values.
2. Create an array `s` of the `sparse` structure to store the sparse matrix.
3. Initialize variables `tot_ele`, `tot_row`, `tot_col`, `ptr`, `i`, `j`, `k`, `flag`, and `data`.
4. Clear the screen using `clrscr()`.
5. Prompt the user for the total number of rows and columns in the matrix and store them in `tot_row` and `tot_col`.
6. Initialize `s[0].row` and `s[0].col` with the total rows and columns.
7. Iterate through each row and column of the matrix: a. Prompt the user to input an element (`data`) at the current row and column. b. If `data` is not equal to zero, update the `s` array with the `row`, `col`, and `ele` values. c. Increment `ptr` and `tot_ele`.
8. Update `s[0].ele` with the total number of non-zero elements.
9. Display the non-zero elements with their row, column, and data.
10. Reset `ptr` to 1.
11. Display the original matrix: a. Iterate through each row and column. b. If the current `s[ptr]` row and column match the current indices, print the element (`ele`). c. If there's no match, print "0".
12. End the program.

Q13. Polynomial

Algorithm for Adding Two Polynomials

1. Define a structure `TERM` with fields for coefficient (`COE`) and exponent (`EXP`).
2. Declare three arrays of type `TERM`: `EXP1` for the first polynomial, `EXP2` for the second polynomial, and `ADD_EXP` for the addition result.
3. Declare variables `fc` (for the first polynomial count), `sc` (for the second polynomial count), and `ac` (for the addition count).
4. Create functions for inputting and displaying the first polynomial, inputting and displaying the second polynomial, adding the two polynomials, and displaying the addition result.
5. Initialize the `fc`, `sc`, and `ac` variables.
6. Implement the `INPUT_FIRST_EXPRESSION` function to input the first polynomial terms, storing them in the `EXP1` array.
7. Implement the `DISPLAY_FIRST_EXPRESSION` function to display the first polynomial.
8. Implement the `INPUT_SECOND_EXPRESSION` function to input the second polynomial terms, storing them in the `EXP2` array.

9. Implement the `DISPLAY_SECOND_EXPRESSION` function to display the second polynomial.
10. Implement the `ADD_POLYNOMIALS` function to add the two polynomials and store the result in the `ADD_EXP` array.
11. Implement the `DISPLAY_ADDED_EXPRESSION` function to display the addition result.
12. In the `main` function, clear the screen using `clrscr()`.
13. Call the functions to input, display, and add the polynomials.
14. Display the addition result.
15. Use `getch()` to wait for a key press before exiting the program.

Q14. Tree (Insertion)

loc′ is a pointer which points to node to be processed. ‘par′ is pointer which points to parent of ‘loc′. ‘item′ contains the information value of the leaf node to be deleted.

```
CREATE_TREE(BINARY_S_TREE *loc, int data)
  7.  Check if data == info[loc], then:
             Print "DUPLICATE VALUE" and return.

  8.  Else check if data < info[loc], then:
   3. If left[loc] != NULL, then:
                Call CREATE_TREE (left[loc], data).

   4. Else
                Print "INSERTING TO LEFT".
                Allocate memory to a node named FRESH.
                Set info[FRESH] := data.
                Set left[FRESH] := NULL.
                Set right[FRESH] := NULL.
                set left[loc] := FRESH.

  3. Else check if data > info[loc], then:
   5. If right[loc] != NULL, then:
                Call CREATE_TREE (right[loc], data).

   6. Else
                Print "INSERTING TO RIGHT".
                Allocate memory to a node named FRESH.
                Set info[FRESH] := data.
                Set left[FRESH] := NULL.
                Set right[FRESH] := NULL.
                Set right[loc] := FRESH.
     [End of step 1 if else if structure]
  4. Exit
```

Q15. Tree (Deletion)

```
DELETE_LEAF()
   4. Set loc := root
   5. Repeat steps (a) and (b) while info[loc] != item.
   4. Check if item < info[loc], then:
          Set par := loc and loc := left[loc].

  5. Else check if item > info[loc], then:
          Set par := loc and loc := right[loc].
     [End of while loop in step 2]

  6. Check if info[par] > info[loc], then:
     Set left[par] := NULL.

  7. Else Check if info[par] < info[loc], then:
     Set right[par] := NULL.

  8. Free the memory occupied by loc.
  9. Exit.
```

'loc' is a pointer which points to node to be processed. 'par' is pointer which points to parent of 'loc'. 'item' contains the information value of the leaf node to be deleted. 'child' is a pointer which points towards the left or right child of 'loc'.

```
DELETE_NODE_WITH_SINGLE_CHILD()
  1. Set loc := root
  2. Repeat while info[loc] != item.
          (j)   Check if item < info[loc], then:
          Set par := loc and loc := left[loc].

          (k)   Else check if item > info[loc], then:
          Set par := loc and loc := right[loc].
     [End of while loop in step 2]

  3. Check if left[loc] == NULL then:
          Set child := right[loc].

  4. Check if right[loc] == NULL then:
          Set child := left[loc].

  5. Check if info[par] > info[loc] then:
          Set left[par] := child.

  6. Else Check if info[par] < info[loc] then:
          Set right[par] := child.

  7. Set right[loc] := NULL and left[loc] := NULL.
  8. Free the memory occupied by loc.
  9. Exit.

DELETE_NODE_WITH_TWO_CHILDS()
  9. Set loc := root.
  10. Repeat step (a) while info[loc] != item
          (g)   Check if item < info[loc], then
                Set par := loc and loc := left[loc].
```

```
              Else
              Set par := loc and loc := right[loc].
          [End of step (a) if structure]

     [End of step (2) while loop]
11.  Set ptr=loc;    Node Position to Be Deleted (25)
12.  Set ptrr := right[loc].

13.  Repeat step (b) while left[ptrr] != NULL
          (h)     Set ptr := ptrr and ptrr := left[ptrr].
     As The INORDER Successor Of Any Node Is The Left Most Child
     In The Right Sub Tree Of That Node.
     [End of step (5) while loop]

14.  Check if left[ptrr] = NULL and right[ptrr] = NULL, then:
                                              Checking For Leaf Node
          (i)    Check if info[ptr] > info[ptrr], then:
                 Set left[ptr] := NULL.
                 Else
                 Set right[ptr] := NULL.
          [End of step (c) if structure]

15.  Check if left[ptrr] != NULL or right[ptrr] != NULL, then:
                                        Checking For Single Child Node
          (j)    Check if left[ptrr]= NULL, then:
                 Set child := right[ptrr].
                 Else
                 Set child := left[ptrr].
          [End of step (d) if structure]

          (k)    Check if info[ptr] > info[ptrr], then
                 Set left[ptr] := child.
                 Else
                 Set right[ptr] := child.
          [End of step (e) if structure]

    [End of step (6) if structure]
16. [Changing position of node to be deleted (25) with node with
    value 33]
          (l)     Check else if loc = left[par], then:
                  Set left[par] := ptrr.
                  Else
                  Set right[par] := ptrr.
          [End of step (f) if structure]

17.       Set left[ptrr] := left[loc].
18. Set right[ptrr] := right[loc].
19. Free the memory occupied by loc.
20.       Exit.
```