## Note : Head is used to Point to the First Node of the LinkedList here

1. **Traversing a Singular Linked List**

➔

```
Procedure TraverseSLL(Head , data){
    Set ptr = Head
    If Head != NULL{
            While ptr != NULL{
                    Display ptr->Data
                    Set ptr = ptr->Next
            }
    }
    Else{
            Return -1 //Linked list does not exist
    }
}
End Procedure
```

2. **Inserting a Node at The start of SLL(Singular LinkedList)**

➔

```
Procedure InsertAtBeginning(Head, Data) {
   // Create a new node with the given data
   Set node = CreateNewNode(Data)

   // Point the new node's Next to the current head
   Set node->Next = Head

   // Update the head to point to the new node
   Set Head = node
```

```
        Return 1  // Successfully inserted at the beginning
    }
    End Procedure
```

3. **Insert Node At the End of SLL**

➔

```
Procedure InsertAtEnd(Head, Data) {
  // Create a new node with the given data
  Set node = CreateNewNode(Data)

  // If the list is empty (Head is NULL), insert the new node at the beginning
  If Head == NULL {
     Set Head = node  // Make the new node the head
     Return 1  // Successfully inserted at the end (which is the beginning if list was empty)
  }

  // Otherwise, traverse to the last node
  Set ptr = Head
  While ptr->Next != NULL {
     ptr = ptr->Next
  }

  // Now, ptr is the last node, so set its Next pointer to the new node
  Set ptr->Next = node

  Return 1  // Successfully inserted at the end
}
```

**4. Insert Node in Between a SLL / At a Given Position **(imp)**

➔

```
Procedure InsertInBetween(Head, Data, Position) {
   // Position refers to the Position in SLL where you want to add a new node
   Set counter = 0
   Set ptr = Head
   Set node = CreateNewNode(Data)

   // Check if position is valid or Head is NULL
   If Head == NULL or Position < 0 {
      Return -1  // Linked list does not exist or invalid position
   }

   // Handle insertion at the beginning (Position 0)
   If Position == 0 {
      Set node->Next = Head
      Set Head = node
      Return 1  // Successfully inserted at the beginning
   }
   // Traverse to the node just before the specified position
   While ptr != NULL and counter < Position - 1 {
      ptr = ptr->Next
      counter++
   }
   // Check if position is out of bounds
   If ptr == NULL {
      Return -1  // Position does not exist in the Linked List
   }
```

```
    // Perform the insertion at the desired position
    Set node->Next = ptr->Next
    Set ptr->Next = node

    Return 1  // Successfully inserted at the desired position
}
End Procedure
```

**5. Delete At the Start :**

➔

```
Procedure DeleteAtStart(Head) {
    // If the list is empty, return an error
    If Head == NULL {
        Return -1  // List is empty, cannot delete
    }

    // Store the node to be deleted
    Set temp = Head

    // Move the head to the next node
    Set Head = Head->Next

    // Delete the old head node
    Free(temp)

    Return 1  // Successfully deleted the node at the start
}
```

**6. Delete At the End of SLL :**

➔

```
Procedure DeleteAtEnd(Head) {
   // If the list is empty, return an error
   If Head == NULL {
      Return -1  // List is empty, cannot delete
   }

   // Special case: If there's only one node in the list
   If Head->Next == NULL {
      Free(Head)
      Set Head = NULL  // Now the list is empty
      Return 1  // Successfully deleted the last node
   }

   // Traverse to the second-to-last node
   Set ptr = Head
   While ptr->Next != NULL and ptr->Next->Next != NULL {
      ptr = ptr->Next
   }
   // ptr is now the second-to-last node
   Set temp = ptr->Next  // The last node to be deleted
   Set ptr->Next = NULL  // Remove the last node from the list

   // Delete the last node
   Free(temp)

   Return 1  // Successfully deleted the node at the end
}
```

**7. Deletion In between Nodes in SLL / At a given Position **(Imp)**

➔

```
Procedure DeleteInBetween(Head, Position) {
   // Position refers to the position of the node to be deleted in the list
   // If the position is invalid or the list is empty, return an error
   If Head == NULL or Position < 1 {
      Return -1  // Invalid position or empty list
   }

   // Special case: If we want to delete the head node (Position 0)
   If Position == 0 {
      Set temp = Head
      Set Head = Head->Next
      Free(temp)
      Return 1  // Successfully deleted the node at the start
   }

   // Traverse to the node just before the node to be deleted
   Set counter = 0
   Set ptr = Head

   While ptr != NULL and counter < Position - 1 {
      ptr = ptr->Next
      counter++
   }
```

```
    // If ptr is NULL or the next node is NULL, the position is out of bounds
    If ptr == NULL or ptr->Next == NULL {
        Return -1  // Position does not exist in the Linked List
    }

    // Node to be deleted is the node after ptr
    Set temp = ptr->Next
    Set ptr->Next = ptr->Next->Next  // Bypass the node to be deleted

    // Delete the node
    Free(temp)

    Return 1  // Successfully deleted the node at the specified position
}

End Procedures
```

# Circular Linked List

**1. Insert in Beginning of circular linked list**

➔

```
Procedure InsertAtBeginning(Head, Data) {
   // Create a new node with the given data
   Set node = CreateNewNode(Data)

   // If the list is empty (Head is NULL), make the new node point to itself
   If Head == NULL {
      Set node->Next = node  // Circular reference: node points to itself
      Set Head = node
      Return 1  // Successfully inserted at the beginning (only node in the list)
   }

   // Otherwise, find the last node (node whose Next is Head)
   Set ptr = Head
   While ptr->Next != Head {
      ptr = ptr->Next
   }

   // Now, ptr is the last node. Insert the new node.
   Set node->Next = Head  // New node points to the current head
   Set ptr->Next = node   // Last node points to the new node
   Set Head = node        // Update the head to point to the new node

   Return 1  // Successfully inserted at the beginning
}
```

**2. Inserting at the end of Circular Linked List**

➔

```
Procedure InsertAtEnd(Head, Data) {
   // Create a new node with the given data
   Set node = CreateNewNode(Data)

   // If the list is empty (Head is NULL), make the new node point to itself
   If Head == NULL {
      Set node->Next = node  // Circular reference: node points to itself
      Set Head = node
      Return 1  // Successfully inserted at the end (only node in the list)
   }

   // Otherwise, find the last node (node whose Next is Head)
   Set ptr = Head
   While ptr->Next != Head {
      ptr = ptr->Next
   }

   // Now, ptr is the last node. Insert the new node at the end.
   Set ptr->Next = node   // Last node's Next points to the new node
   Set node->Next = Head  // New node points to the head (circular linkage)

   Return 1  // Successfully inserted at the end
}
```

**3. Insertion In between of Circular Linked List / At a Given Position **(Imp)**

➔

```
Procedure InsertInBetween(Head, Data, Position) {
  // If position is invalid (less than 0), return an error
  If Position < 0 {
    Return -1  // Invalid position
  }

  // Create a new node with the given data
  Set node = CreateNewNode(Data)

  // If the list is empty (Head is NULL), treat the position as 0 and insert at the beginning
  If Head == NULL {
    Set node->Next = node  // Circular reference: node points to itself
    Set Head = node
    Return 1  // Successfully inserted at position 0
  }

  // Special case: Insertion at the beginning (Position 0)
  If Position == 0 {
    // Find the last node (whose Next is Head)
    Set ptr = Head
    While ptr->Next != Head {
      ptr = ptr->Next
    }

    // Insert the new node at the beginning
    Set node->Next = Head  // New node points to the current head
    Set ptr->Next = node   // Last node points to the new node
```

```
        Set Head = node       // Update the head to point to the new node

        Return 1  // Successfully inserted at the beginning
    }

    // Traverse to the node just before the insertion point (Position - 1)
    Set ptr = Head
    Set counter = 0
    While counter < Position - 1 and ptr->Next != Head {
        ptr = ptr->Next
        counter++
    }

    // If we've reached the end of the list (i.e., Position is out of bounds), return error
    If ptr->Next == Head and counter != Position - 1 {
        Return -1  // Position does not exist in the list
    }

    // Insert the new node after ptr (at the desired position)
    Set node->Next = ptr->Next  // New node points to the next node
    Set ptr->Next = node       // Previous node points to the new node

    Return 1  // Successfully inserted the node at the specified position
}
```

**4. Deletion at the Start of Circular Linked List**

➔

```
Procedure DeleteAtBeginning(Head) {
  // If the list is empty (Head is NULL), return error
  If Head == NULL {
    Return -1  // List is empty, cannot delete
  }

  // If the list has only one node
  If Head->Next == Head {
    Delete Head
    Set Head = NULL  // List is now empty
    Return 1  // Successfully deleted the only node
  }

  // Otherwise, traverse to the last node (whose Next is Head)
  Set ptr = Head
  While ptr->Next != Head {
    ptr = ptr->Next
  }

  // Now, ptr is the last node. Update its Next pointer to the second node
  Set temp = Head
  Set Head = Head->Next  // Move head to the next node
  Set ptr->Next = Head   // Last node now points to the new head

  // Delete the old head node
  Delete temp
```

```
    Return 1  // Successfully deleted the node at the beginning
}
```

**5. Deletion At the End of Circular Linked List**

➔

```
Procedure DeleteAtEnd(Head) {
    // If the list is empty (Head is NULL), return error
    If Head == NULL {
        Return -1  // List is empty, cannot delete
    }

    // If the list has only one node
    If Head->Next == Head {
        Delete Head
        Set Head = NULL  // List is now empty
        Return 1  // Successfully deleted the last node
    }

    // Otherwise, traverse to the second-to-last node (whose Next is Head)
    Set ptr = Head
    While ptr->Next->Next != Head {
        ptr = ptr->Next
    }

    // Now, ptr is the second-to-last node. Delete the last node.
    Set temp = ptr->Next
    Set ptr->Next = Head  // Second-to-last node's Next points to Head

    // Delete the last node
```

```
    Delete temp

    Return 1  // Successfully deleted the node at the end
  }
```

**6. Deletion in between a Circular Linked list / At a given Position **(Imp)**

➔

```
Procedure DeleteAtPosition(Head, Position) {
  // If the list is empty (Head is NULL), return error
  If Head == NULL {
    Return -1  // List is empty, cannot delete
  }

  // Special case: Deletion at the beginning (Position 0)
  If Position == 0 {
    // If the list has only one node (Head points to itself)
    If Head->Next == Head {
      Delete Head  // Delete the only node
      Set Head = NULL  // List becomes empty
      Return 1  // Successfully deleted the node
    }

    // Otherwise, find the last node (whose Next is Head)
    Set ptr = Head
    While ptr->Next != Head {
      ptr = ptr->Next
    }

    // Now, ptr is the last node. Update its Next pointer to the second node
```

```
        Set temp = Head
        Set Head = Head->Next  // Move head to the next node
        Set ptr->Next = Head   // Last node now points to the new head

        // Delete the old head node
        Delete temp
        Return 1  // Successfully deleted the node at the beginning
    }

    // Traverse to the node just before the position (Position - 1)
    Set ptr = Head
    Set counter = 0
    While counter < Position - 1 and ptr->Next != Head {
        ptr = ptr->Next
        counter++
    }

    // If we've reached the end of the list (ptr->Next == Head), position is out of bounds
    If ptr->Next == Head or counter != Position - 1 {
        Return -1  // Position does not exist in the list
    }
    // Now ptr points to the node just before the node to be deleted
    Set temp = ptr->Next  // The node to be deleted
    Set ptr->Next = ptr->Next->Next  // Bypass the node to be deleted

    // Delete the node
    Delete temp
    Return 1  // Successfully deleted the node at the given position
}
```

**7. Traversing a Circular Linked List**

➔

```
Procedure Traverse(Head) {
   // If the list is empty (Head is NULL), return
   If Head == NULL {
      Return  // List is empty, no nodes to traverse
   }

   // Start from the head and print each node's data
   Set ptr = Head
   Repeat
      Print(ptr->Data)  // Print current node's data
      Set ptr = ptr->Next  // Move to the next node
   Until ptr == Head  // Stop when we reach the head again

   Return  // Traversal complete
}
```

# Doubly Linked List

**1. Transversing a Doubly Circular Linked List**

➔

```
Procedure Traverse(Head) {
   // If the list is empty (Head is NULL), return
   If Head == NULL {
      Return  // No nodes to traverse
   }

   // Start from the head and print each node's data
   Set ptr = Head
   While ptr != NULL {
      Print(ptr->Data)  // Print current node's data
      Set ptr = ptr->Next  // Move to the next node
   }

   Return  // Traversal complete
}
```

**2. Insertion At a Given Position in a Doubly Linked List **(imp)**

➔

```
Procedure InsertAtPosition(Head, Data, Position) {
  // If position is invalid (less than 0), return an error
  If Position < 0 {
    Return -1  // Invalid position
  }
  // Create a new node with the given data
  Set newNode = CreateNewNode(Data)

  // Special case: Insertion at the beginning (Position 0)
  If Position == 0 {
    // Insert at the beginning (before Head)
    If Head == NULL {
      Set Head = newNode  // If the list is empty, set the new node as the head
    } Else {
      Set newNode->Next = Head  // New node points to the current head
      Set Head->Prev = newNode  // Current head's previous points to the new node
      Set Head = newNode        // Update the head to the new node
    }
    Return 1  // Successfully inserted at position 0
  }

  // Traverse to the node just before the given position (Position - 1)
  Set ptr = Head
  Set counter = 0
  While ptr != NULL and counter < Position - 1 {
    ptr = ptr->Next
    counter++
```

```
    }

    // If the position is out of bounds (ptr is NULL), return error
    If ptr == NULL {
        Return -1  // Position is out of bounds
    }

    // Insert the new node after ptr
    Set newNode->Next = ptr->Next  // New node points to the next node
    Set newNode->Prev = ptr       // New node's previous points to ptr

    // If the node after ptr exists, update its Prev pointer to newNode
    If ptr->Next != NULL {
        Set ptr->Next->Prev = newNode
    }

    // Update ptr's Next pointer to point to the new node
    Set ptr->Next = newNode

    Return 1  // Successfully inserted the node at the given position
}
```

**3. Deletion At a Given Position in a Doubly Linked List **(imp)**

➔

```
Procedure DeleteAtPosition(Head, Position) {
  // If the list is empty (Head is NULL), return error
  If Head == NULL {
    Return -1  // List is empty, cannot delete
  }

  // Special case: Deletion at the beginning (Position 0)
  If Position == 0 {
    Set temp = Head  // The node to be deleted
    Set Head = Head->Next  // Move the head pointer to the next node

    // If the list has more than one node, update the new head's previous pointer
    If Head != NULL {
      Set Head->Prev = NULL
    }

    Delete temp  // Delete the old head node
    Return 1  // Successfully deleted the node at the beginning
  }

  // Traverse to the node just before the given position (Position - 1)
  Set ptr = Head
  Set counter = 0
  While ptr != NULL and counter < Position - 1 {
    ptr = ptr->Next
    counter++
  }
```

```
    // If ptr is NULL or the next node is NULL, the position is out of bounds
    If ptr == NULL or ptr->Next == NULL {
        Return -1  // Position does not exist in the list
    }

    // The node to be deleted is ptr->Next
    Set temp = ptr->Next  // The node to be deleted
    Set ptr->Next = ptr->Next->Next  // Bypass the node to be deleted

    // If the node to be deleted is not the last node, update the next node's Prev pointer
    If ptr->Next != NULL {
        Set ptr->Next->Prev = ptr
    }

    // Delete the node
    Delete temp
    Return 1  // Successfully deleted the node at the given position
}
```

# Doubly Circular Linked List

**1. Transversing a Doubly Circular LinkedList**

➔

```
Procedure Traverse(Head) {
   // If the list is empty (Head is NULL), return
   If Head == NULL {
      Return  // No nodes to traverse
   }

   // Start from the head and print each node's data
   Set ptr = Head
   Repeat
      Print(ptr->Data)  // Print current node's data
      Set ptr = ptr->Next  // Move to the next node
   Until ptr == Head  // Stop when we return to the head

   Return  // Traversal complete
}
```

**2. Insertion at a Given Position in a Doubly Circular Linked List **(imp)**

➔

```
Procedure InsertAtPosition(Head, Data, Position) {
   // If position is invalid (less than 0), return an error
   If Position < 0 {
      Return -1  // Invalid position
   }

   // Create a new node with the given data
   Set newNode = CreateNewNode(Data)
```

```
// Special case: Insertion at the beginning (Position 0)
If Position == 0 {
   // If the list is empty, new node points to itself
   If Head == NULL {
      Set newNode->Next = newNode
      Set newNode->Prev = newNode
      Set Head = newNode
   } Else {
      // Insert new node before Head (circular adjustment)
      Set newNode->Next = Head
      Set newNode->Prev = Head->Prev
      Set Head->Prev->Next = newNode
      Set Head->Prev = newNode
      Set Head = newNode
   }
   Return 1  // Successfully inserted at position 0
}

// Traverse to the node just before the given position (Position - 1)
Set ptr = Head
Set counter = 0
While ptr->Next != Head and counter < Position - 1 {
   ptr = ptr->Next
   counter++
}

// If the position is out of bounds (ptr->Next == Head), return error
If ptr->Next == Head and counter != Position - 1 {
```

```
    Return -1  // Position does not exist in the list
  }

  // Insert the new node after ptr
  Set newNode->Next = ptr->Next  // New node points to the next node
  Set newNode->Prev = ptr       // New node's previous points to ptr

  // Update the next node's Prev pointer (if it's not the head)
  If ptr->Next != Head {
    Set ptr->Next->Prev = newNode
  }

  // Update ptr's Next pointer to point to the new node
  Set ptr->Next = newNode

  Return 1  // Successfully inserted the node at the given position
}
```

**3. Deletion at a Given Position in a Doubly Circular Linked List **(imp)**

➔

```
Procedure DeleteAtPosition(Head, Position) {
  // If the list is empty (Head is NULL), return error
  If Head == NULL {
    Return -1  // List is empty, cannot delete
  }

  // Special case: Deletion at the beginning (Position 0)
  If Position == 0 {
    Set temp = Head  // The node to be deleted
```

```
    // If there's only one node
    If Head->Next == Head {
        Set Head = NULL  // The list will become empty
    } Else {
        Set Head = Head->Next  // Move head to the next node
        Set Head->Prev = temp->Prev  // Update the new head's prev pointer
        Set temp->Prev->Next = Head  // Last node's next points to new head
    }

    Delete temp  // Delete the old head node
    Return 1  // Successfully deleted the node at the beginning
}

// Traverse to the node just before the given position (Position - 1)
Set ptr = Head
Set counter = 0
While ptr->Next != Head and counter < Position - 1 {
    ptr = ptr->Next
    counter++
}

// If ptr->Next == Head or position is out of bounds, return error
If ptr->Next == Head or counter != Position - 1 {
    Return -1  // Position does not exist in the list
}

// The node to be deleted is ptr->Next
Set temp = ptr->Next  // The node to be deleted
Set ptr->Next = ptr->Next->Next  // Bypass the node to be deleted
```

```
    // If the node to be deleted is not the last node, update the next node's Prev pointer
    If ptr->Next != Head {
        Set ptr->Next->Prev = ptr
    }

    // Delete the node
    Delete temp
    Return 1  // Successfully deleted the node at the given position
}
```

# Stack

1. **Push (Insertion)**

➔

```
Procedure Push(Stack, Top, Data) {
   // Check if stack is full (if there's a size limit)
   If Stack is Full {
      Return -1  // Stack overflow error
   }

   // Increment the Top pointer
   Set Top = Top + 1

   // Insert the data at the top of the stack
   Set Stack[Top] = Data

   Return 1  // Successfully pushed the element
}
```

2. **Pop (Deletion)**

➔

```
Procedure Pop(Stack, Top) {
   // Check if stack is empty
   If IsEmpty(Stack, Top) {
      Return -1  // Stack underflow error
   }

   // Get the top element
   Set Data = Stack[Top]
```

```
    // Decrement the Top pointer
    Set Top = Top - 1
    Return Data  // Return the popped element
}
```

3. **Peek/Top (View the top element)**

➔

```
Procedure Peek(Stack, Top) {
    // Check if the stack is empty
    If IsEmpty(Stack, Top) {
        Return -1  // Error: Stack is empty
    }

    // Return the element at the top
    Return Stack[Top]
}
```

4. **IsEmpty (Check if the stack is empty)**

➔

```
Procedure IsEmpty(Stack, Top) {
    // If Top is -1, the stack is empty
    If Top == -1 {
        Return True  // Stack is empty
    }

    Return False  // Stack is not empty
}
```

5. **IsFull (Check if the stack is full, if there's a capacity limit)**

➔

```
Procedure IsFull(Stack, Top, MaxSize) {
    // If Top is equal to MaxSize - 1, the stack is full
```

```
    If Top == MaxSize - 1 {
        Return True  // Stack is full
    }

    Return False  // Stack is not full
}
```

6. **Traversal (Traverse all elements in the stack)**

➔

```
Procedure Traverse(Stack, Top) {
    // If the stack is empty, return
    If IsEmpty(Stack, Top) {
        Return  // No elements to traverse
    }

    // Traverse from the top to the bottom
    Set i = Top
    While i >= 0 {
        Print(Stack[i])  // Print the element at index i
        Set i = i - 1  // Move to the next element
    }

    Return  // Traversal complete
}
```

**7. Pseudocode for Infix to Postfix **(imp)**

➔

```
Procedure InfixToPostfix(InfixExpression) {
  // Initialize an empty stack and an empty postfix expression list
  Set Stack = Empty Stack
  Set PostfixExpression = Empty List

  // Define precedence of operators
  Set Precedence = { '+': 1, '-': 1, '*': 2, '/': 2, '^': 3 }

  // Traverse the infix expression character by character
  For each character `ch` in InfixExpression {
    If ch is an operand (e.g., variable or number) {
      Append `ch` to PostfixExpression  // Add operand to result
    } Else If ch is a left parenthesis `(` {
      Push `ch` onto Stack  // Push '(' onto the stack
    } Else If ch is a right parenthesis `)` {
      // Pop from stack until we encounter '('
      While Stack is not empty and Top of Stack is not '(' {
        Pop from Stack and Append to PostfixExpression
      }
      Pop '(' from Stack  // Pop the left parenthesis from the stack
    } Else If ch is an operator {
      While Stack is not empty and Precedence(Top of Stack) >= Precedence(ch) {
        Pop from Stack and Append to PostfixExpression
      }
      Push `ch` onto Stack  // Push the current operator onto the stack
    }
  }
```

```
    // Pop all remaining operators from the stack
    While Stack is not empty {
        Pop from Stack and Append to PostfixExpression
    }

    Return PostfixExpression
}
```

**8. Pseudocode for Infix to Prefix **(imp)**

➔

```
Procedure InfixToPrefix(InfixExpression) {
    // Reverse the infix expression
    Set ReversedInfix = Reverse(InfixExpression)

    // Swap parentheses: Replace '(' with ')' and vice versa
    For each character `ch` in ReversedInfix {
        If `ch` is '(' {
            Set ch = ')'
        } Else If `ch` is ')' {
            Set ch = '('
        }
    }

    // Convert the reversed infix expression to postfix
    Set PostfixReversed = InfixToPostfix(ReversedInfix)

    // Reverse the postfix expression to get the prefix expression
    Set PrefixExpression = Reverse(PostfixReversed)
```

```
    Return PrefixExpression
}
```

# Queue

1. **Enqueue (Insert an Element into the Queue)**

➔

```
Procedure Enqueue(Queue, Rear, Data, MaxSize) {
   // Check if the queue is full
     If Queue is Full{
      Return -1  // Error: Queue overflow
   }

   // Increment the rear pointer (queue's back)
   Set Rear = Rear + 1  //For Linear Queue
   // Set Rear = (Rear + 1) % MaxSize   // Use this for Circular Queue instead of that

   // Add the data to the rear position
   Set Queue[Rear] = Data

   Return 1  // Successfully enqueued the data
}
```

2. **Dequeue (Remove an Element from the Queue)**

➔

```
Procedure Dequeue(Queue, Front, Rear) {
   // Check if the queue is empty
   If Queue is Full {
      Return -1  // Error: Queue underflow
   }

   // Get the data from the front of the queue
   Set Data = Queue[Front]
```

```
    // Increment the front pointer (queue's front)
    Set Front =Front + 1  //For Linear Queue
    // Set Rear = (Front + 1) % MaxSize   // Use this for Circular Queue instead of that

    Return Data  // Return the dequeued element
}
```

**3. Peek/Front (View the Front Element)**

➔

```
Procedure Peek(Queue, Front, Rear) {
    // Check if the queue is empty
    If Queue is Full {
        Return -1  // Error: Queue is empty
    }

    // Return the element at the front
    Return Queue[Front]
}
```

**4. IsEmpty (Check if the Queue is Empty) for Circular Queue**

➔

```
Procedure IsEmpty(Queue, Front, Rear) {
    // If front equals rear, the queue is empty
    If Front == Rear {
        Return True  // Queue is empty
    }
    Return False  // Queue is not empty
}
```

**5. IsEmpty for Linear Queue :**

➔

```
Procedure IsEmptyLinear(Queue, Front, Rear) {
   // Queue is empty when front > rear
   If Front > Rear {
      Return True  // The queue is empty
   }

   Return False  // The queue is not empty
}
```

**6. IsFull (Check if the Queue is Full) for Circular Queue**

➔

```
Procedure IsFull(Queue, Rear, MaxSize) {
   // If the next rear position is the front position, the queue is full
   If (Rear + 1) % MaxSize == Front {
      Return True  // Queue is full
   }
   Return False  // Queue is not full
}
```

**7. IsFull for Linear Queue :**

➔

```
Procedure IsFull(Queue, Rear, MaxSize) {
   // Queue is full when rear == MaxSize - 1
   If Rear == MaxSize - 1 {
      Return True  // Queue is full
   }
```

```
    Return False  // Queue is not full
}
```

**8. Traversing/Viewing All Elements in the Circular Queue**

➔

```
Procedure Traverse(Queue, Front, Rear, MaxSize) {
    // If the queue is empty, there is nothing to traverse
    If Queue is Full {
        Return  // No elements to traverse
    }

    // Start from the front and traverse through the queue
    Set i = Front
    While i != Rear {
        Print(Queue[i])  // Print the element at position i
        i = (i + 1) % MaxSize  // Move to the next element in a circular manner
    }

    // Print the rear element
    Print(Queue[Rear])

    Return  // Traversal complete
}
```

**9. Transversal In Linear Queue**

➔

```
Procedure Traverse(Queue, Front, Rear) {
   // Check if the queue is empty
   If Queue is Full {
      Return  // No elements to traverse
   }

   // Traverse the queue and print elements from front to rear
   For i = Front to Rear {
      Print(Queue[i])
   }

   Return  // Traversal complete
}
```