1. Singly Linked List – Insertion

Aim: To perform insertion operations.

Description:

A singly linked list is a dynamic linear data structure where each node contains data and a reference (pointer) to the next node. It does not store elements in contiguous memory like arrays. This allows efficient insertion and deletion operations without shifting elements. However, traversal is only possible in one direction.

pesudocode :

```
START

Create Node with data and next pointer

Create LinkedList with head = NULL

INSERT_BEGINNING(data)
   Create new node
   new node next = head
   head = new node
END

INSERT_END(data)
   Create new node

   IF head is NULL
      head = new node
   ELSE
      temp = head
      WHILE temp next is not NULL
         move temp to next node
      END WHILE
      temp next = new node
   END IF
END

INSERT_POSITION(data, position)
   Create new node

   IF position = 1
      new node next = head
      head = new node
   ELSE
      temp = head
      Move temp to (position - 1) node

      IF position invalid
         Print "Position out of range"
      ELSE
         new node next = temp next
```

```
        temp next = new node
      END IF
   END IF
END

DISPLAY
   temp = head
   WHILE temp is not NULL
      Print temp data
      temp = temp next
   END WHILE
END

STOP
```

2. Doubly Linked List – Deletion

Aim: To perform deletion operations.

Description:

A doubly linked list is an extension of singly linked list where each node contains two pointers: one pointing to the previous node and one to the next node. This allows traversal in both forward and backward directions, making deletion operations easier compared to singly linked lists.

pesudocode :

```
START

Create Node with:
   data
   prev pointer
   next pointer

Create DoublyLinkedList with head = NULL

DELETE_BEGINNING
   IF head is NULL
      Print "List is empty"
   ELSE
      head = head next
      IF head is not NULL
         head prev = NULL
      END IF
   END IF
END

DELETE_END
   IF head is NULL
      Print "List is empty"
   ELSE IF only one node exists
      head = NULL
   ELSE
      temp = head
      Move temp to last node
      temp prev next = NULL
   END IF
END

DELETE_POSITION(position)
   IF list is empty
      Print "List is empty"
   ELSE IF position = 1
      DELETE_BEGINNING
   ELSE
```

```
      Move temp to given position

      IF position invalid
         Print "Position out of range"
      ELSE
         temp prev next = temp next

         IF temp next exists
            temp next prev = temp prev
         END IF
      END IF
   END IF
END

DISPLAY list

STOP
```

3. Circular Linked List – Searching

Aim: To search element.

Description:

In a circular linked list, the last node connects back to the first node instead of pointing to NULL. This structure is useful in applications requiring a continuous loop, such as round-robin scheduling.

pesudocode :

```
START

Create Node with data and next pointer

Create CircularLinkedList with head = NULL

INSERT_END(data)
  Create new node

  IF head is NULL
    head = new node
    new node next = head
  ELSE
    temp = head

    WHILE temp next != head
      temp = temp next
    END WHILE

    temp next = new node
    new node next = head
  END IF
END

TRAVERSE
  IF head is NULL
    Print "List is empty"
  ELSE
    temp = head

    REPEAT
      Print temp data
      temp = temp next
    UNTIL temp = head
  END IF
END

SEARCH(key)
  IF head is NULL
    Print "List is empty"
```

```
  ELSE
    temp = head
    position = 1

    REPEAT
      IF temp data = key
        Print element found
        STOP
      END IF

      temp = temp next
      position = position + 1
    UNTIL temp = head

    Print "Element not found"
  END IF
END

STOP
```

4. Stack using Linked List

Aim: To implement stack.

Description:

A stack is a linear data structure that follows the LIFO (Last In First Out) principle. Using linked list avoids fixed size limitations and allows dynamic memory usage.

pesudocode :

```
START

Create Node with data and next

Create Stack with top = NULL

PUSH(data)
   Create new node
   new node next = top
   top = new node
END

POP
   IF top is NULL
      Print "Stack Underflow"
   ELSE
      temp = top
      top = top next
      Delete temp
   END IF
END

PEEK
   IF top is NULL
      Print "Stack is empty"
   ELSE
      Print top data
   END IF
END

DISPLAY
   temp = top

   WHILE temp is not NULL
      Print temp data
      temp = temp next
   END WHILE
END

STOP
```

5. Queue using Linked List

Aim: To implement queue.

Description:

A queue is a FIFO (First In First Out) data structure where insertion happens at the rear and deletion at the front. Linked list implementation avoids overflow limitations of arrays.

pesudocode :

```
START

Create Node with data and next

Create Queue with:
   front = NULL
   rear = NULL

ENQUEUE(data)
   Create new node

   IF rear is NULL
      front = rear = new node
   ELSE
      rear next = new node
      rear = new node
   END IF
END

DEQUEUE
   IF front is NULL
      Print "Queue Underflow"
   ELSE
      temp = front
      front = front next

      IF front is NULL
         rear = NULL
      END IF

      Delete temp
   END IF
END

PEEK
   IF front is NULL
      Print "Queue is empty"
   ELSE
      Print front data
```

```
    END IF
END

DISPLAY queue

STOP
```

6. Expression Conversion

Aim: Infix to Postfix/Prefix.

Description:

Infix expressions are human-readable but require precedence rules. Postfix and prefix notations eliminate ambiguity and are used in compilers and expression evaluation.

pesudocode :

```
START

PRECEDENCE(operator)
   Return priority of operator
END

INFIX_TO_POSTFIX(expression)
   Create empty stack
   postfix = ""

   FOR each character in expression

      IF operand
         Add to postfix

      ELSE IF '('
         Push into stack

      ELSE IF ')'
         Pop until '(' found

      ELSE
         WHILE stack not empty AND
              precedence(top) >= precedence(current)
            Pop and add to postfix
         END WHILE

         Push operator
      END IF
   END FOR

   Pop remaining operators

   RETURN postfix
END

INFIX_TO_PREFIX(expression)
   Reverse expression
   Swap brackets
```

```
    Convert to postfix
    Reverse postfix result
END

STOP
```

7. Binary Tree Traversals

Aim: To perform traversal.

Description:

A binary tree is a hierarchical structure where each node has at most two children. Traversal methods help visit all nodes systematically for processing data.

pesudocode :

```
START

Create Node with:
   data
   left child
   right child

INORDER(root)
   IF root exists
      INORDER(left)
      Print root data
      INORDER(right)
   END IF
END

PREORDER(root)
   IF root exists
      Print root data
      PREORDER(left)
      PREORDER(right)
   END IF
END

POSTORDER(root)
   IF root exists
      POSTORDER(left)
      POSTORDER(right)
      Print root data
   END IF
END

STOP
```

8. Graph Traversals (BFS & DFS)

Aim: To traverse graph.

Description:

A graph consists of vertices and edges. BFS explores level by level using a queue, while DFS explores depth-wise using recursion or stack.

pesudocode :

```
START

Create Graph using adjacency list

ADD_EDGE(u, v)
  Add v to u list
  Add u to v list
END

BFS(start)
  Create visited set
  Create queue

  Insert start into queue

  WHILE queue not empty
    Remove node from queue

    IF node not visited
      Print node
      Mark visited

      Add all unvisited neighbors into queue
    END IF
  END WHILE
END

DFS(node)
  IF node not visited
    Print node
    Mark visited

    FOR each neighbor
      DFS(neighbor)
    END FOR
  END IF
END

STOP
```

9. Binary Search Tree (BST)

Aim: To perform operations.

Description:

A Binary Search Tree is a tree where left subtree contains smaller values and right subtree contains larger values. It provides efficient searching, insertion, and deletion.

pesudocode :

```
START

Create Node with:
   key
   left child
   right child

INSERT(root, key)
   IF root is NULL
      Create node

   ELSE IF key < root key
      Insert into left subtree

   ELSE
      Insert into right subtree
   END IF

   Return root
END

SEARCH(root, key)
   IF root is NULL OR key found
      Return root

   ELSE IF key < root key
      Search left subtree

   ELSE
      Search right subtree
END

DELETE(root, key)
   Find node

   IF node has no child
      Delete node

   ELSE IF one child
```

```
      Replace with child

   ELSE
      Find inorder successor
      Replace node value
      Delete successor
   END IF
END

INORDER / PREORDER / POSTORDER traversal

STOP
```

10. AVL Tree

Aim: To maintain balance.

Description:

AVL tree maintains balance by ensuring height difference between left and right subtrees is at most 1. Rotations are used to maintain balance after insertion

pesudocode :

```
START

Create Node with:
   key
   left
   right
   height

GET_HEIGHT(node)
   Return node height
END

GET_BALANCE(node)
   Return left height - right height
END

RIGHT_ROTATE(node)
   Perform right rotation
END

LEFT_ROTATE(node)
   Perform left rotation
END

INSERT(root, key)

   Perform normal BST insertion

   Update height

   Calculate balance factor

   IF LL imbalance
      Right rotate

   ELSE IF RR imbalance
      Left rotate

   ELSE IF LR imbalance
```

```
        Left rotate left child
        Right rotate root

    ELSE IF RL imbalance
        Right rotate right child
        Left rotate root
    END IF

    Return root
END

INORDER traversal

STOP
```

11. Quick Sort & Insertion Sort

Aim: Sorting.

Description:

Sorting is essential for efficient searching and data organization. Quick sort uses divide-and-conquer strategy, while insertion sort is simple and efficient for small datasets.

pesudocode :

```
START

QUICK_SORT(array)

   IF array size <= 1
      Return array

   Choose pivot

   Create left array with smaller elements
   Create right array with greater elements

   Return QUICK_SORT(left) + pivot + QUICK_SORT(right)
END

INSERTION_SORT(array)

   FOR i = 1 to n-1
      key = array[i]
      j = i - 1

      WHILE j >= 0 AND array[j] > key
         Shift array[j]
         j = j - 1
      END WHILE

      Insert key at correct position
   END FOR

   Return array
END

STOP
```

12. Heap Sort & Merge Sort

Aim: Efficient sorting.

Description:

Heap sort uses a binary heap structure, while merge sort divides the array into halves and merges them after sorting. Both are efficient for large datasets.

pesudocode :

```
START

HEAPIFY(array, n, i)
  largest = i

  Compare left child
  Compare right child

  IF largest changes
    Swap elements
    HEAPIFY again
  END IF
END

HEAP_SORT(array)
  Build max heap

  FOR each element from end
    Swap first and last
    Heapify reduced heap
  END FOR
END

MERGE_SORT(array)

  IF array size > 1

    Divide array into two halves

    MERGE_SORT(left)
    MERGE_SORT(right)

    Merge sorted halves
  END IF
END

STOP
```

13. Linear & Binary Search

Aim: Searching techniques.

Description:

Searching helps locate elements in a dataset. Linear search scans sequentially, while binary search works on sorted arrays and reduces search space efficiently.

pesudocode :

```
START

LINEAR_SEARCH(array, key)

   FOR each element
      IF element = key
         Return index
      END IF
   END FOR

   Return -1
END

BINARY_SEARCH(array, key)

   low = 0
   high = n - 1

   WHILE low <= high

      mid = (low + high) / 2

      IF array[mid] = key
         Return mid

      ELSE IF array[mid] < key
         low = mid + 1

      ELSE
         high = mid - 1
      END IF
   END WHILE

   Return -1
END

STOP
```