STACKS

```
DECLARE Stack : ARRAY[1:10] OF INTEGER
DECLARE TopPointer : INTEGER
DECLARE BasePointer : INTEGER
DECLARE StackFull : INTEGER

BasePointer ← 1
TopPointer ← 0
StackFull ← 10

PROCEDURE PUSH(Item : INTEGER)
    IF TopPointer < StackFull THEN
        TopPointer ← TopPointer + 1
        Stack[TopPointer] ← Item
    ELSE
        OUTPUT "Stack is full, cannot push"
    ENDIF
ENDPROCEDURE

PROCEDURE POP()
    DECLARE Item : INTEGER
    IF TopPointer = BasePointer - 1 THEN
        OUTPUT "Stack is empty, cannot pop"
    ELSE
        Item ← Stack[TopPointer]
        TopPointer ← TopPointer - 1
        OUTPUT "Popped item: ", Item
    ENDIF
ENDPROCEDURE
```

QUEUES

```
DECLARE Queue : ARRAY[1:10] OF INTEGER
DECLARE FrontPointer : INTEGER
DECLARE RearPointer : INTEGER
DECLARE QueueFull : INTEGER
DECLARE QueueLength : INTEGER
DECLARE UpperBound : INTEGER

FrontPointer ← 1
RearPointer ← 0
QueueFull ← 10
QueueLength ← 0
UpperBound ← 10

PROCEDURE ENQUEUE(Item : INTEGER)
    IF QueueLength < QueueFull THEN
        IF RearPointer < UpperBound THEN
            RearPointer ← RearPointer + 1
        ELSE
            RearPointer ← 1
        ENDIF
        QueueLength ← QueueLength + 1
        Queue[RearPointer] ← Item
    ELSE
        OUTPUT "Queue is full, cannot enqueue"
    ENDIF
ENDPROCEDURE

PROCEDURE DEQUEUE()
    DECLARE Item : INTEGER
    IF QueueLength = 0 THEN
        OUTPUT "Queue is empty, cannot dequeue"
    ELSE
        Item ← Queue[FrontPointer]
        IF FrontPointer = UpperBound THEN
            FrontPointer ← 1
        ELSE
            FrontPointer ← FrontPointer + 1
        ENDIF
        QueueLength ← QueueLength − 1
        OUTPUT "Dequeued item: ", Item
    ENDIF
ENDPROCEDURE
```

```
FUNCTION IS_QUEUE_EMPTY() RETURNS BOOLEAN
    IF QueueLength = 0 THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF
ENDFUNCTION

FUNCTION IS_STACK_EMPTY() RETURNS BOOLEAN
    IF TopPointer = BasePointer − 1 THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF
ENDFUNCTION

FUNCTION IS_QUEUE_FULL() RETURNS BOOLEAN
    IF QueueLength = QueueFull THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF
ENDFUNCTION

FUNCTION IS_STACK_FULL() RETURNS BOOLEAN
    IF TopPointer = StackFull THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF
ENDFUNCTION
```

```
PROCEDURE REVERSE_QUEUE()
    DECLARE TempStack : ARRAY[1:10] OF INTEGER
    DECLARE TempTopPointer : INTEGER
    DECLARE TempItem : INTEGER

    TempTopPointer ← 0

    // Dequeue all elements from queue and push onto stack
    WHILE NOT IS_QUEUE_EMPTY() DO
        TempItem ← Queue[FrontPointer]
        CALL DEQUEUE()
        TempTopPointer ← TempTopPointer + 1
        TempStack[TempTopPointer] ← TempItem
    ENDWHILE

    // Pop all elements from stack and enqueue back to queue
    WHILE TempTopPointer > 0 DO
        TempItem ← TempStack[TempTopPointer]
        TempTopPointer ← TempTopPointer - 1
        CALL ENQUEUE(TempItem)
    ENDWHILE
ENDPROCEDURE
```

LINKED LIST

```
DECLARE LinkedList : ARRAY[1:10] OF INTEGER
DECLARE Pointers : ARRAY[1:10] OF INTEGER
DECLARE StartPointer : INTEGER
DECLARE HeapPointer : INTEGER
DECLARE ListFull : INTEGER
DECLARE TempPointer : INTEGER
DECLARE PrevPointer : INTEGER
StartPointer ← −1  // No elements in the list
HeapPointer ← 1    // Points to first free space
ListFull ← 10

// Initialize heap (free space management)
FOR i ← 1 TO ListFull − 1
    Pointers[i] ← i + 1
NEXT i
Pointers[ListFull] ← −1
```

```
PROCEDURE INSERT(Item : INTEGER)
    IF HeapPointer = −1 THEN
        OUTPUT "List is full, cannot insert"
    ELSE
        TempPointer ← HeapPointer
        HeapPointer ← Pointers[HeapPointer]  // Move heap pointer to
next free space
        LinkedList[TempPointer] ← Item

        // If list is empty, new item becomes first node
        IF StartPointer = −1 THEN
            StartPointer ← TempPointer
            Pointers[TempPointer] ← −1
        ELSE
            PrevPointer ← −1
            CurrPointer ← StartPointer

            // Find correct position (sorted order)
            WHILE CurrPointer <> −1 AND LinkedList[CurrPointer] <
Item DO
                PrevPointer ← CurrPointer
                CurrPointer ← Pointers[CurrPointer]
            ENDWHILE

            // Insert at start
            IF PrevPointer = −1 THEN
                Pointers[TempPointer] ← StartPointer
                StartPointer ← TempPointer
            ELSE
                Pointers[TempPointer] ← Pointers[PrevPointer]
                Pointers[PrevPointer] ← TempPointer
            ENDIF
        ENDIF
    ENDIF
ENDPROCEDURE
```

```
PROCEDURE DELETE(Item : INTEGER)
    IF StartPointer = −1 THEN
        OUTPUT "List is empty, cannot delete"
    ELSE
        PrevPointer ← −1
        CurrPointer ← StartPointer

        // Find the item to delete
        WHILE CurrPointer <> −1 AND LinkedList[CurrPointer] <> Item
DO
            PrevPointer ← CurrPointer
            CurrPointer ← Pointers[CurrPointer]
        ENDWHILE

        IF CurrPointer = −1 THEN
            OUTPUT "Item not found"
        ELSE
            // Delete first item
            IF PrevPointer = −1 THEN
                StartPointer ← Pointers[CurrPointer]
            ELSE
                Pointers[PrevPointer] ← Pointers[CurrPointer]
            ENDIF

            // Return space to heap
            Pointers[CurrPointer] ← HeapPointer
            HeapPointer ← CurrPointer
        ENDIF
    ENDIF
ENDPROCEDURE
```

```
FUNCTION SEARCH(Item : INTEGER) RETURNS BOOLEAN
    CurrPointer ← StartPointer
    WHILE CurrPointer <> -1 DO
        IF LinkedList[CurrPointer] = Item THEN
            RETURN TRUE
        ENDIF
        CurrPointer ← Pointers[CurrPointer]
    ENDWHILE
    RETURN FALSE
ENDFUNCTION


PROCEDURE REVERSE_LL()
    DECLARE PrevPointer : INTEGER
    DECLARE CurrPointer : INTEGER
    DECLARE NextPointer : INTEGER

    PrevPointer ← -1
    CurrPointer ← StartPointer

    WHILE CurrPointer <> -1 DO
        NextPointer ← Pointers[CurrPointer]  // Store next node
        Pointers[CurrPointer] ← PrevPointer  // Reverse pointer
        PrevPointer ← CurrPointer
        CurrPointer ← NextPointer
    ENDWHILE

    StartPointer ← PrevPointer  // Update head
ENDPROCEDURE


PROCEDURE DISPLAY()
    CurrPointer ← StartPointer
    IF CurrPointer = -1 THEN
        OUTPUT "List is empty"
    ELSE
        WHILE CurrPointer <> -1 DO
            OUTPUT LinkedList[CurrPointer]
            CurrPointer ← Pointers[CurrPointer]
        ENDWHILE
    ENDIF
ENDPROCEDURE
```

BINARY TREES

```
DECLARE Tree : ARRAY[1:10] OF INTEGER
DECLARE LeftPointer : ARRAY[1:10] OF INTEGER
DECLARE RightPointer : ARRAY[1:10] OF INTEGER
DECLARE RootPointer : INTEGER
DECLARE FreePointer : INTEGER
DECLARE TreeSize : INTEGER

TreeSize ← 10
RootPointer ← −1  // Empty tree
FreePointer ← 1   // Points to first free node

// Initialize free space
FOR i ← 1 TO TreeSize − 1
    LeftPointer[i] ← −1
    RightPointer[i] ← −1
NEXT i
```

```
PROCEDURE INSERT(Item : INTEGER)
    IF FreePointer = −1 THEN
        OUTPUT "Tree is full, cannot insert"
    ELSE
        NewNode ← FreePointer
        FreePointer ← FreePointer + 1
        Tree[NewNode] ← Item
        LeftPointer[NewNode] ← −1
        RightPointer[NewNode] ← −1

        IF RootPointer = −1 THEN
            RootPointer ← NewNode
        ELSE
            CurrPointer ← RootPointer
            WHILE TRUE DO
                IF Item < Tree[CurrPointer] THEN
                    IF LeftPointer[CurrPointer] = −1 THEN
                        LeftPointer[CurrPointer] ← NewNode
                        RETURN
                    ELSE
                        CurrPointer ← LeftPointer[CurrPointer]
                    ENDIF
                ELSE
                    IF RightPointer[CurrPointer] = −1 THEN
                        RightPointer[CurrPointer] ← NewNode
                        RETURN
                    ELSE
                        CurrPointer ← RightPointer[CurrPointer]
                    ENDIF
                ENDIF
            ENDWHILE
        ENDIF
    ENDIF
ENDPROCEDURE
```

```
FUNCTION SEARCH(Item : INTEGER) RETURNS BOOLEAN
    CurrPointer ← RootPointer
    WHILE CurrPointer <> -1 DO
        IF Tree[CurrPointer] = Item THEN
            RETURN TRUE
        ELSEIF Item < Tree[CurrPointer] THEN
            CurrPointer ← LeftPointer[CurrPointer]
        ELSE
            CurrPointer ← RightPointer[CurrPointer]
        ENDIF
    ENDWHILE
    RETURN FALSE
ENDFUNCTION


PROCEDURE INORDER_TRAVERSAL(CurrPointer : INTEGER)
    IF CurrPointer <> -1 THEN
        CALL INORDER_TRAVERSAL(LeftPointer[CurrPointer])
        OUTPUT Tree[CurrPointer]
        CALL INORDER_TRAVERSAL(RightPointer[CurrPointer])
    ENDIF
ENDPROCEDURE


PROCEDURE PREORDER_TRAVERSAL(CurrPointer : INTEGER)
    IF CurrPointer <> -1 THEN
        OUTPUT Tree[CurrPointer]
        CALL PREORDER_TRAVERSAL(LeftPointer[CurrPointer])
        CALL PREORDER_TRAVERSAL(RightPointer[CurrPointer])
    ENDIF
ENDPROCEDURE


PROCEDURE POSTORDER_TRAVERSAL(CurrPointer : INTEGER)
    IF CurrPointer <> -1 THEN
        CALL POSTORDER_TRAVERSAL(LeftPointer[CurrPointer])
        CALL POSTORDER_TRAVERSAL(RightPointer[CurrPointer])
        OUTPUT Tree[CurrPointer]
    ENDIF
ENDPROCEDURE


FUNCTION TREE_HEIGHT(CurrPointer : INTEGER) RETURNS INTEGER
    IF CurrPointer = -1 THEN
        RETURN 0
    ELSE
        LeftHeight ← TREE_HEIGHT(LeftPointer[CurrPointer])
        RightHeight ← TREE_HEIGHT(RightPointer[CurrPointer])
        RETURN MAX(LeftHeight, RightHeight) + 1
    ENDIF
ENDFUNCTION
```

```
FUNCTION FIND_MINIMUM() RETURNS INTEGER
    CurrPointer ← RootPointer
    WHILE LeftPointer[CurrPointer] <> -1 DO
        CurrPointer ← LeftPointer[CurrPointer]
    ENDWHILE
    RETURN Tree[CurrPointer]
ENDFUNCTION


FUNCTION FIND_MAXIMUM() RETURNS INTEGER
    CurrPointer ← RootPointer
    WHILE RightPointer[CurrPointer] <> -1 DO
        CurrPointer ← RightPointer[CurrPointer]
    ENDWHILE
    RETURN Tree[CurrPointer]
ENDFUNCTION
```

```
PROCEDURE DELETE(Item : INTEGER)
    CurrPointer ← RootPointer
    PrevPointer ← -1

    // Find the node to delete
    WHILE CurrPointer <> -1 AND Tree[CurrPointer] <> Item DO
        PrevPointer ← CurrPointer
        IF Item < Tree[CurrPointer] THEN
            CurrPointer ← LeftPointer[CurrPointer]
        ELSE
            CurrPointer ← RightPointer[CurrPointer]
        ENDIF
    ENDWHILE

    IF CurrPointer = -1 THEN
        OUTPUT "Item not found"
        RETURN
    ENDIF

    // Case 1: No children
    IF LeftPointer[CurrPointer] = -1 AND RightPointer[CurrPointer] = -1 THEN
        IF PrevPointer = -1 THEN
            RootPointer ← -1
        ELSEIF LeftPointer[PrevPointer] = CurrPointer THEN
            LeftPointer[PrevPointer] ← -1
        ELSE
            RightPointer[PrevPointer] ← -1
        ENDIF
    ENDIF

    // Case 2: One child
    ELSEIF LeftPointer[CurrPointer] = -1 OR RightPointer[CurrPointer] = -1
THEN
        IF LeftPointer[CurrPointer] <> -1 THEN
            ChildPointer ← LeftPointer[CurrPointer]
        ELSE
            ChildPointer ← RightPointer[CurrPointer]
        ENDIF

        IF PrevPointer = -1 THEN
            RootPointer ← ChildPointer
        ELSEIF LeftPointer[PrevPointer] = CurrPointer THEN
            LeftPointer[PrevPointer] ← ChildPointer
        ELSE
            RightPointer[PrevPointer] ← ChildPointer
        ENDIF
    ENDIF

    // Case 3: Two children (replace with inorder successor)
    ELSE
        SuccessorPointer ← RightPointer[CurrPointer]
        While LeftPointer[SuccessorPointer] <> -1 DO
            SuccessorPointer ← LeftPointer[SuccessorPointer]
        ENDWHILE
        Tree[CurrPointer] ← Tree[SuccessorPointer]
        CALL DELETE(Tree[SuccessorPointer])
    ENDIF
ENDPROCEDURE
```

LINEAR SEARCH

```
PROCEDURE LINEAR_SEARCH(Array : ARRAY OF INTEGER, Size : INTEGER,
Target : INTEGER) RETURNS INTEGER
    DECLARE Index : INTEGER
    FOR Index ← 1 TO Size
        IF Array[Index] = Target THEN
            RETURN Index  // Item found, return position
        ENDIF
    NEXT Index
    RETURN -1  // Item not found
ENDPROCEDURE
```

BINARY SEARCH

```
PROCEDURE BINARY_SEARCH(Array : ARRAY OF INTEGER, Low : INTEGER,
High : INTEGER, Target : INTEGER) RETURNS INTEGER
    DECLARE Mid : INTEGER
    WHILE Low <= High DO
        Mid ← (Low + High) DIV 2
        IF Array[Mid] = Target THEN
            RETURN Mid  // Item found
        ELSEIF Array[Mid] < Target THEN
            Low ← Mid + 1
        ELSE
            High ← Mid - 1
        ENDIF
    ENDWHILE
    RETURN -1  // Item not found
ENDPROCEDURE
```

BUBBLE SORT

```
PROCEDURE BUBBLE_SORT(Array : ARRAY OF INTEGER, Size : INTEGER)
    DECLARE Swapped : BOOLEAN
    DECLARE i, Temp : INTEGER
    REPEAT
        Swapped ← FALSE
        FOR i ← 1 TO Size - 1
            IF Array[i] > Array[i + 1] THEN
                // Swap elements
                Temp ← Array[i]
                Array[i] ← Array[i + 1]
                Array[i + 1] ← Temp
                Swapped ← TRUE
            ENDIF
        NEXT i
    UNTIL Swapped = FALSE
ENDPROCEDURE
```

INSERTION SORT

```
PROCEDURE INSERTION_SORT(Array : ARRAY OF INTEGER, Size : INTEGER)
    DECLARE i, j, Key : INTEGER
    FOR i ← 2 TO Size
        Key ← Array[i]
        j ← i - 1
        WHILE j > 0 AND Array[j] > Key DO
            Array[j + 1] ← Array[j]
            j ← j - 1
        ENDWHILE
        Array[j + 1] ← Key
    NEXT i
ENDPROCEDURE
```

SELECTION SORT

```
PROCEDURE SELECTION_SORT(Array : ARRAY OF INTEGER, Size : INTEGER)
    DECLARE i, j, MinIndex, Temp : INTEGER
    FOR i ← 1 TO Size − 1
        MinIndex ← i
        FOR j ← i + 1 TO Size
            IF Array[j] < Array[MinIndex] THEN
                MinIndex ← j
            ENDIF
        NEXT j
        // Swap min element with the first unsorted element
        Temp ← Array[i]
        Array[i] ← Array[MinIndex]
        Array[MinIndex] ← Temp
    NEXT i
ENDPROCEDURE
```

QUICK SORT

```
PROCEDURE QUICK_SORT(Array : ARRAY OF INTEGER, Low : INTEGER, High :
INTEGER)
    DECLARE PivotIndex : INTEGER
    IF Low < High THEN
        PivotIndex ← PARTITION(Array, Low, High)
        CALL QUICK_SORT(Array, Low, PivotIndex − 1)
        CALL QUICK_SORT(Array, PivotIndex + 1, High)
    ENDIF
ENDPROCEDURE

FUNCTION PARTITION(Array : ARRAY OF INTEGER, Low : INTEGER, High :
INTEGER) RETURNS INTEGER
    DECLARE Pivot, i, j, Temp : INTEGER
    Pivot ← Array[High]
    i ← Low − 1
    FOR j ← Low TO High − 1
        IF Array[j] < Pivot THEN
            i ← i + 1
            Temp ← Array[i]
            Array[i] ← Array[j]
            Array[j] ← Temp
        ENDIF
    NEXT j
    Temp ← Array[i + 1]
    Array[i + 1] ← Array[High]
    Array[High] ← Temp
    RETURN i + 1
ENDFUNCTION
```

MERGE SORT

```
PROCEDURE MERGE_SORT(Array : ARRAY OF INTEGER, Left : INTEGER, Right : INTEGER)
    DECLARE Mid : INTEGER
    IF Left < Right THEN
        Mid ← (Left + Right) DIV 2
        CALL MERGE_SORT(Array, Left, Mid)
        CALL MERGE_SORT(Array, Mid + 1, Right)
        CALL MERGE(Array, Left, Mid, Right)
    ENDIF
ENDPROCEDURE

PROCEDURE MERGE(Array : ARRAY OF INTEGER, Left : INTEGER, Mid : INTEGER, Right :
INTEGER)
    DECLARE i, j, k : INTEGER
    DECLARE LeftSize, RightSize : INTEGER
    DECLARE LeftArray, RightArray : ARRAY OF INTEGER

    LeftSize ← Mid − Left + 1
    RightSize ← Right − Mid

    // Copy data into temp arrays
    FOR i ← 1 TO LeftSize
        LeftArray[i] ← Array[Left + i − 1]
    NEXT i

    FOR j ← 1 TO RightSize
        RightArray[j] ← Array[Mid + j]
    NEXT j

    // Merge temp arrays back into main array
    i ← 1
    j ← 1
    k ← Left
    WHILE i <= LeftSize AND j <= RightSize DO
        IF LeftArray[i] <= RightArray[j] THEN
            Array[k] ← LeftArray[i]
            i ← i + 1
        ELSE
            Array[k] ← RightArray[j]
            j ← j + 1
        ENDIF
        k ← k + 1
    ENDWHILE

    // Copy remaining elements
    WHILE i <= LeftSize DO
        Array[k] ← LeftArray[i]
        i ← i + 1
        k ← k + 1
    ENDWHILE

    WHILE j <= RightSize DO
        Array[k] ← RightArray[j]
        j ← j + 1
        k ← k + 1
    ENDWHILE
ENDPROCEDURE
```