Section 19.1 - Algorithms

Layer 6: High-Order Language

Syllabus Content Section 19: Computational Thinking and Problem-Solving

S19.1.1 Show understanding of linear and binary searching methods

- Write an algorithm to implement a linear search
- Write an algorithm to implement a binary search
- The conditions necessary for the use of a binary search
- How the performance of a binary search varies according to the number of data items

---

> Linear Search

```
FUNCTION linearSearch(arr:ARRAY,s:INTEGER,n:INTEGER) RETURN INTEGER
    FOR i <- 1 to n
        IF arr[i]=n
            RETURN i
        ENDIF
    NEXT i
    RETURN -1
ENDFUNCTION

// Time complexity O(n)
// Space complexity O(1)
```

> Binary Search(sorted array)

```
FUNCTION BinarySearch(arr:ARRAY,s:INTEGER,n:INTEGER) RETURN INTEGER
    DECLARE left:INTEGER
    DECLARE right:INTEGER
    left <- 1
    right <- s
    WHILE left <= right
        DECLARE mid:INTEGER
        mid <- (left + right) DIV 2
        IF arr[mid] = n
            RETURN mid
        IF arr[mid] < n
            left <- mid + 1
        ELSE
            right <- mid - 1
    ENDWHILE
    RETURN -1
ENDFUNCTION
```

```
// Time complexity O(log n)
// Space complexity O(1)
```

S19.1.2 Show understanding of insertion sort and bubble sort methods

- Write an algorithm to implement an insertion sort
- Write an algorithm to implement a bubble sort
- Performance of a sorting routine may depend on the initial order of the data and the number of data items

---

> insertion sort

```
PROCEDURE InsertSort(arr:ARRAY,n:INTEGER) RETURN INTEGER
    FOR i <- 1 TO n
        DECLARE j
        j <- i
        WHILE j>0 AND arr[j]<arr[j-1]
            DECLARE temp
            temp <- arr[j]
            arr[j]<-arr[j-1]
            arr[j-1]<-temp
            j-=1
        ENDWHILE
    ENDFOR
ENDPROCEDURE

// Time complexity O(n^2)
// Space complexity O(1)
```

> Bubble sort

```
PROCEDURE BubblesSort(arr:ARRAY,n:INTEGER) RETURN INTEGER
    FOR i <- 1 TO n
        FOR j <- 1 TO n-i
            IF arr[j]>arr[j+1]
                DECLARE temp:INTEGER
                temp <- arr[j]
                arr[j] <- arr[j+1]
                arr[j+1] <- temp
            ENDIF
        ENDFOR
    ENDFOR
ENDPROCEDURE

// Time complexity O(n^2)
// Space complexity O(1)
```

S19.1.3 Show understanding of and use Abstract Data Types (ADT)

- Write algorithms to find an item in each of the following: linked list, binary tree
- Write algorithms to insert an item into each of the following: stack, queue, linked list, binary tree
- Write algorithms to delete an item from each of the following: stack, queue, linked list
- Show understanding that a graph is an example of an ADT. Describe the key features of a graph and justify its use for a given situation
- Candidates will not be required to write code for this structure.

> Linked lists

- Create a new linked list

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = -1
// Declare record type to store data and pointer
TYPE ListNode
        DECLARE Data : STRING
        DECLARE Pointer : INTEGER
ENDTYPE

DECLARE StartPointer : INTEGER
DECLARE FreeListPtr : INTEGER
DECLARE List : ARRAY[0 : 6] OF ListNode
PROCEDURE InitialiseList
        StartPointer ← NullPointer // set start pointer
        FreeListPtr ← 0 // set starting position of free list
        FOR Index ← 0 TO 5 // link all nodes to make free list
        List[Index].Pointer ← Index + 1
        NEXT Index
        List[6].Pointer ← NullPointer // last node of free list
ENDPROCEDURE
```

- Insert a new node into an ordered linked list

```
PROCEDURE InsertNode(NewItem)
        IF FreeListPtr <> NullPointer THEN // there is space in the array
                // take node from free list and store data item
                NewNodePtr ← FreeListPtr
                List[NewNodePtr].Data ← NewItem
                FreeListPtr ← List[FreeListPtr].Pointer
                // find insertion point
                ThisNodePtr ← StartPointer // start at beginning of list
                PreviousNodePtr ← NullPointer
                WHILE ThisNodePtr <> NullPointer AND List[ThisNodePtr].Data < NewItem
DO // while not end of list
                        PreviousNodePtr ← ThisNodePtr // remember this node
                        // follow the pointer to the next node
                        ThisNodePtr ← List[ThisNodePtr].Pointer
                ENDWHILE
                IF PreviousNodePtr = StartPointer THEN // insert new node at start of
```

```
list
                        List[NewNodePtr].Pointer ← StartPointer
                        StartPointer ← NewNodePtr
                ELSE // insert new node between previous node and this node
                        List[NewNodePtr].Pointer ← List[PreviousNodePtr].Pointer
                        List[PreviousNodePtr].Pointer ← NewNodePtr
                ENDIF
        ENDIF
ENDPROCEDURE
```

- Find an element in an ordered linked list

```
FUNCTION FindNode(DataItem) RETURNS INTEGER // returns pointer to node
        CurrentNodePtr ← StartPointer // start at beginning of list
        WHILE CurrentNodePtr <> NullPointer // not end of list
                AND List[CurrentNodePtr].Data <> DataItem DO // item not found
                // follow the pointer to the next node
                CurrentNodePtr ← List[CurrentNodePtr].Pointer
        ENDWHILE
        RETURN CurrentNodePtr // returns NullPointer if item not found
ENDFUNCTION
```

- Delete a node from an ordered linked list

```
PROCEDURE DeleteNode(DataItem)
        ThisNodePtr ← StartPointer // start at beginning of list
        WHILE ThisNodePtr <> NullPointer AND List[ThisNodePtr].Data <> DataItem DO //
while not end of list and item not found
                PreviousNodePtr ← ThisNodePtr // remember this node
                // follow the pointer to the next node
                ThisNodePtr ← List[ThisNodePtr].Pointer
        ENDWHILE
        IF ThisNodePtr <> NullPointer THEN // node exists in list
                IF ThisNodePtr = StartPointer THEN // first node to be deleted
                        // move start pointer to the next node in list
                        StartPointer ← List[StartPointer].Pointer
                ELSE
                        // it is not the start node;
                        // so make the previous node’s pointer point to
                        // the current node’s 'next' pointer; thereby removing all
                        // references to the current pointer from the list
                        List[PreviousNodePtr].Pointer ← List[ThisNodePtr].Pointer
                ENDIF
                List[ThisNodePtr].Pointer ← FreeListPtr
                FreeListPtr ← ThisNodePtr
        ENDIF
ENDPROCEDURE
```

- Access all nodes stored in the linked list

```
PROCEDURE OutputAllNodes
        CurrentNodePtr ← StartPointer // start at beginning of list
        WHILE CurrentNodePtr <> NullPointer DO // while not end of list
                OUTPUT List[CurrentNodePtr].Data
                // follow the pointer to the next node
```

```
                CurrentNodePtr ← List[CurrentNodePtr].Pointer
        ENDWHILE
ENDPROCEDURE
```

> binary tree

- Create a new binary tree

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = -1
// Declare record type to store data and pointers
TYPE TreeNode
        DECLARE Data : STRING
        DECLARE LeftPointer : INTEGER
        DECLARE RightPointer : INTEGER
ENDTYPE

DECLARE RootPointer : INTEGER
DECLARE FreePtr : INTEGER
DECLARE Tree : ARRAY[0 : 6] OF TreeNode

PROCEDURE InitialiseTree
        RootPointer ← NullPointer // set start pointer
        FreePtr ← 0 // set starting position of free list
        FOR Index ← 0 TO 5 // link all nodes to make free list
                Tree[Index].LeftPointer ← Index + 1
        NEXT Index
        Tree[6].LeftPointer ← NullPointer // last node of free list
ENDPROCEDURE
```

- Insert a new node into a binary tree

```
PROCEDURE InsertNode(NewItem)
        IF FreePtr <> NullPointer THEN // there is space in the array
                // take node from free list, store data item, set null pointers
                NewNodePtr ← FreePtr
                FreePtr ← Tree[FreePtr].LeftPointer
                Tree[NewNodePtr].Data ← NewItem
                Tree[NewNodePtr].LeftPointer ← NullPointer
                Tree[NewNodePtr].RightPointer ← NullPointer
                // check if empty tree
                IF RootPointer = NullPointer THEN // insert new node at root
                        RootPointer ← NewNodePtr
                ELSE // find insertion point
                        ThisNodePtr ← RootPointer // start at the root of the tree
                        WHILE ThisNodePtr <> NullPointer DO // while not a leaf node
                                PreviousNodePtr ← ThisNodePtr // remember this node
                                IF Tree[ThisNodePtr].Data > NewItem THEN // follow left
pointer
                                        TurnedLeft ← TRUE
                                        ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
                                ELSE // follow right pointer
                                        TurnedLeft ← FALSE
                                        ThisNodePtr ← Tree[ThisNodePtr].RightPointer
                                ENDIF
```

```
                        ENDWHILE
                        IF TurnedLeft = TRUE THEN
                                Tree[PreviousNodePtr].LeftPointer ← NewNodePtr
                        ELSE
                                Tree[PreviousNodePtr].RightPointer ← NewNodePtr
                        ENDIF
                ENDIF
        ENDIF
ENDPROCEDURE
```

- Find a node in a binary tree

```
FUNCTION FindNode(SearchItem) RETURNS INTEGER // returns pointer to node
        ThisNodePtr ← RootPointer // start at the root of the tree
        WHILE ThisNodePtr <> NullPointer AND Tree[ThisNodePtr].Data <> SearchItem DO //
while a pointer to follow and search item not found
                IF Tree[ThisNodePtr].Data > SearchItem THEN // follow left pointer
                        ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
                ELSE // follow right pointer
                        ThisNodePtr ← Tree[ThisNodePtr].RightPointer
                ENDIF
        ENDWHILE
        RETURN ThisNodePtr // will return null pointer if search item not found
ENDFUNCTION
```

S19.1.4 Show how it is possible for ADTs to be implemented from another ADT

- Describe the following ADTs and demonstrate how they can be implemented from appropriate builtin types or other ADTs: stack, queue, linked list, dictionary, binary tree

---

> Stacks

- Create a new stack

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = -1
CONSTANT MaxStackSize = 8
DECLARE BaseOfStackPointer : INTEGER
DECLARE TopOfStackPointer : INTEGER
DECLARE Stack : ARRAY[1 : MaxStackSize - 1] OF STRING
PROCEDURE InitialiseStack
        BaseOfStackPointer ← 0 // set base of stack pointer
        TopOfStackPointer ← NullPointer // set top of stack pointer
ENDPROCEDURE
```

- Push an item onto the stack

```
PROCEDURE Push(NewItem)
        IF TopOfStackPointer < MaxStackSize - 1 THEN // there is space on the stack
                // increment top of stack pointer
                TopOfStackPointer ← TopOfStackPointer + 1
                // add item to top of stack
                Stack[TopOfStackPointer] ← NewItem
        ENDIF
ENDPROCEDURE
```

- Pop an item off the stack

```
FUNCTION Pop()
        DECLARE Item : STRING
        Item ← EMPTYSTRING
        IF TopOfStackPointer > NullPointer THEN // there is at least one item on the
stack
                // pop item off the top of the stack
                Item ← Stack[TopOfStackPointer]
                // decrement top of stack pointer
                TopOfStackPointer ← TopOfStackPointer - 1
        ENDIF
        RETURN Item
ENDFUNCTION
```

> Queues

- Create a new queue

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = -1
CONSTANT MaxQueueSize = 8
DECLARE FrontOfQueuePointer : INTEGER
DECLARE EndOfQueuePointer : INTEGER
DECLARE NumberInQueue : INTEGER
DECLARE Queue : ARRAY[0 : MaxQueueSize - 1] OF STRING
PROCEDURE InitialiseQueue
        FrontOfQueuePointer ← NullPointer // set front of queue pointer
        EndOfQueuePointer ← NullPointer // set end of queue pointer
        NumberInQueue ← 0 // no elements in queue
ENDPROCEDURE
```

- Add an item to the queue

```
PROCEDURE AddToQueue(NewItem)
        IF NumberInQueue < MaxQueueSize THEN // there is space in the queue
                // increment end of queue pointer
                EndOfQueuePointer ← EndOfQueuePointer + 1
                // check for wrap-round
                IF EndOfQueuePointer > MaxQueueSize - 1 THEN // wrap to beginning of
array
                        EndOfQueuePointer ← 0
                        // add item to end of queue
                ENDIF
```

```
                Queue[EndOfQueuePointer] ← NewItem
                // increment counter
                NumberInQueue ← NumberInQueue + 1
        ENDIF
ENDPROCEDURE
```

- Remove an item from the queue

```
FUNCTION RemoveFromQueue()
        DECLARE Item : STRING
        Item ← EMPTYSTRING
        IF NumberInQueue > 0 THEN // there is at least one item in the queue
                // remove item from the front of the queue
                Item ← Queue[FrontOfQueuePointer]
                NumberInQueue ← NumberInQueue - 1
                IF NumberInQueue = 0 THEN // if queue empty, reset pointers
                        CALL InitialiseQueue
                ELSE
                        // increment front of queue pointer
                        FrontOfQueuePointer ← FrontOfQueuePointer + 1
                        // check for wrap-round
                        IF FrontOfQueuePointer > MaxQueueSize - 1 THEN // wrap to
beginning of array
                                FrontOfQueuePointer ← 0
                        ENDIF
                ENDIF
        ENDIF
        RETURN Item
ENDFUNCTION
```

S19.1.5 Show understanding that different algorithms which perform the same task can be compared by using criteria (e.g. time taken to complete the task and memory used)

- Including use of Big O notation to specify time and space complexity

---

Order of growth	Example	Explanation
$O(1)$	FUNCTION GetFirstItem(List : ARRAY) RETURN List[1]	The complexity of the algorithm does not change regardless of data set size
$O(n)$	Linear search Bubble sort performed on an already sorted list	Linear growth

Order of growth	Example	Explanation
$O(log_2 n)$	Binary search	The total time taken increases as the data set size increases, but each comparison halves the data set. So the time taken increases by smaller amounts and approaches constant time.
$O(n^2)$	Bubble sort Insertion sort	Polynomial growth Common with algorithms that involve nested iterations over the data set
$O(n^3)$		Polynomial growth Deeper nested iterations will result in O(n3), O(n4), ...
$O(2^n)$	Recursive calculation of Fibonacci numbers	Exponential growth