# Abstract Data Type

- An Abstract Data Type is a collection of data and set of operations on that data

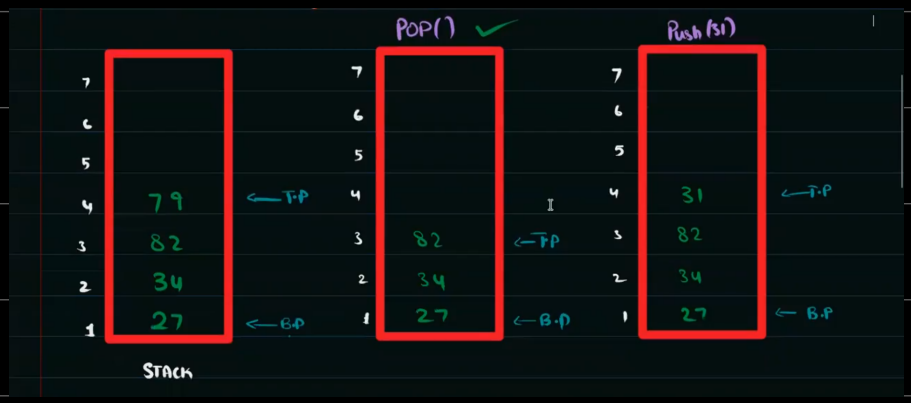
① Stack ② Queue ③ Linked List ④ Binary Tree

## Stack

- A list containing several items operating on the Last in fist out principle (LIFO) principle. Items can be added to the stack (Push) and removed from the stack (Pop). The first item added to the stack is the last item removed from the stack.

B.P= Base Pointer

T.P= Tail Pointer

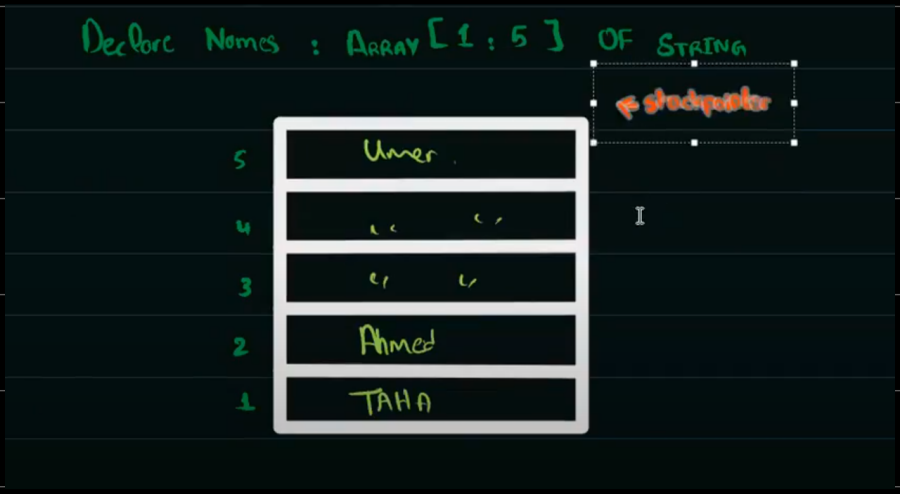


A stack can be implemented using an array and set of pointers.

- The procedure POP() removes an item from the stack.
- The procedure PUSH (<identifier>) adds an item to the stack.

Q- Suppose there is an array "Names" and is used to implement the stack

DECLARE Names: ARRAY [1:5] OF STRING




Q- Write the pseudocode for the procedure Push.

```
PROCEDURE Push ( Value: STRING) ↷ depends on data type of array

    IF stackpointer >5 ↷ Upper bound of Array
        THEN
            OUTPUT " Stack Full"
        ELSE
            Names [stackpointer] ← Value
            Stackpointer ← Stackpointer + 1
    END IF
END PROCEDURE
```

Q- Write the pseudocode for the procedure Pop.

```
PROCEDURE Pop ()
    IF stackpointer = 0
        THEN
            OUTPUT "The stack is empty"
```

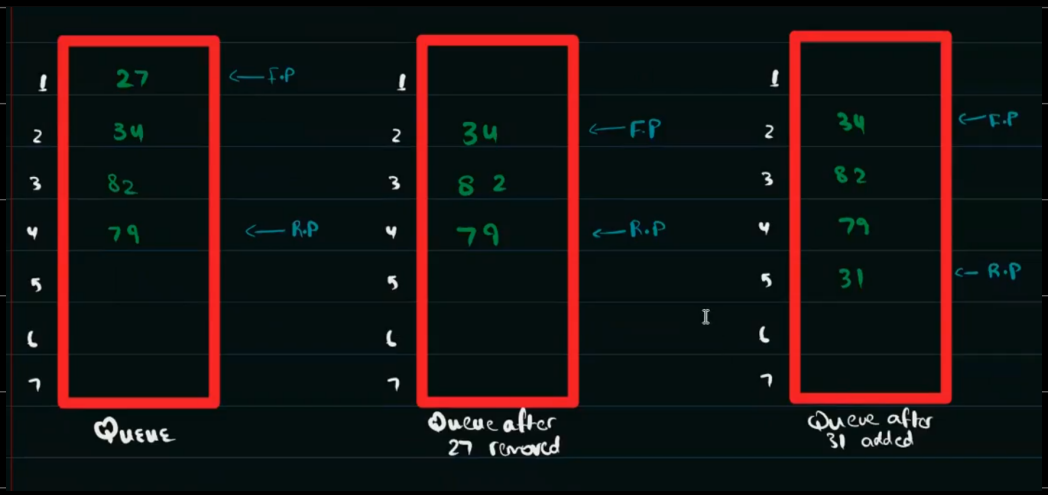
```
        ELSE
            stackpointer ← Stackpointer -1
            OUTPUT Name [Stackpointer]
        END IF
END PROCEDURE
```

# Queue

- A list containing several items operating on the first in first out (FIFO) principle.
- The first item added is the first item removed from the queue

F.P = Front Pointer

R.P = Rear Pointer



Q- Describe a Queue

- Linear Data Structure
- First in First Out // An item is added at the end of the queue and an item is removed from the front
- All items are kept in the order they are entered. It has a head pointer and a tail pointer

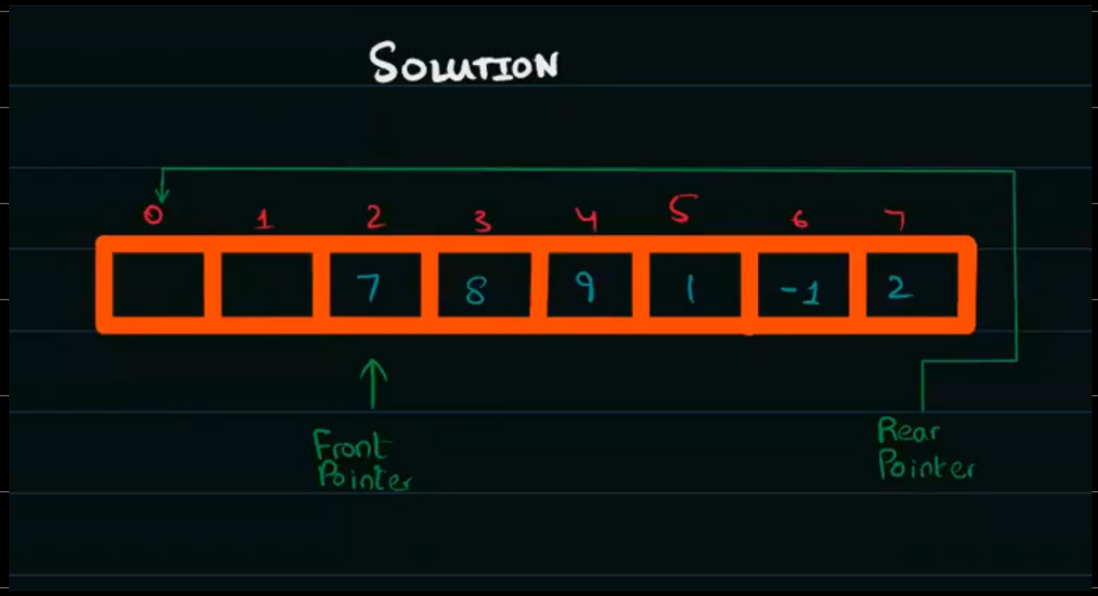
A queue can be implemented using an array and set of pointers.

## Circular Queue


- The condition for a queue being full is that rear pointer should point towards upper bound / max index / last index / 7


- According to the condition of Queue being full is still true.

## Solution



# Questions + Queue Code

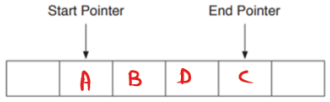
**(b)** When a student prints a document, a print job is created. The print job is sent to a print server.

The print server uses a queue to hold each print job waiting to be printed.

**(i)** The queue is circular and has six spaces to hold jobs.

The queue currently holds four jobs waiting to be printed. The jobs have arrived in the order A, B, D, C.

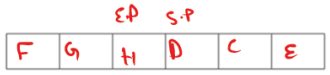
Complete the diagram to show the current contents of the queue.




[1]

**(ii)** Print jobs A and B are now complete. Four more print jobs have arrived in the order E, F, G, H.

Complete the diagram to show the current contents and pointers for the queue.




[3]

**(iii)** State what would happen if another print job is added to the queue in the status in **part (b)(ii)**.

An error message.

.....

.....

[1]

**(iv)** The queue is stored as an array, `Queue`, with six elements. The following algorithm removes a print job from the queue and returns it.

Complete the following **pseudocode** for the function `Remove`.

```
FUNCTION Remove RETURNS STRING
    DECLARE PrintJob : STRING
    IF ..... = EndPointer
        THEN
            RETURN "Empty"
        ELSE
            PrintJob ← Queue[.....]
            IF StartPointer = .....
                THEN
                    StartPointer ← .....
                ELSE
                    StartPointer ← StartPointer + 1
            ENDIF
            RETURN PrintJob
    ENDIF
ENDFUNCTION
```

[4]

**(v)** Explain why the circular queue could not be implemented as a stack.

.....

.....

.....

.....

[2]

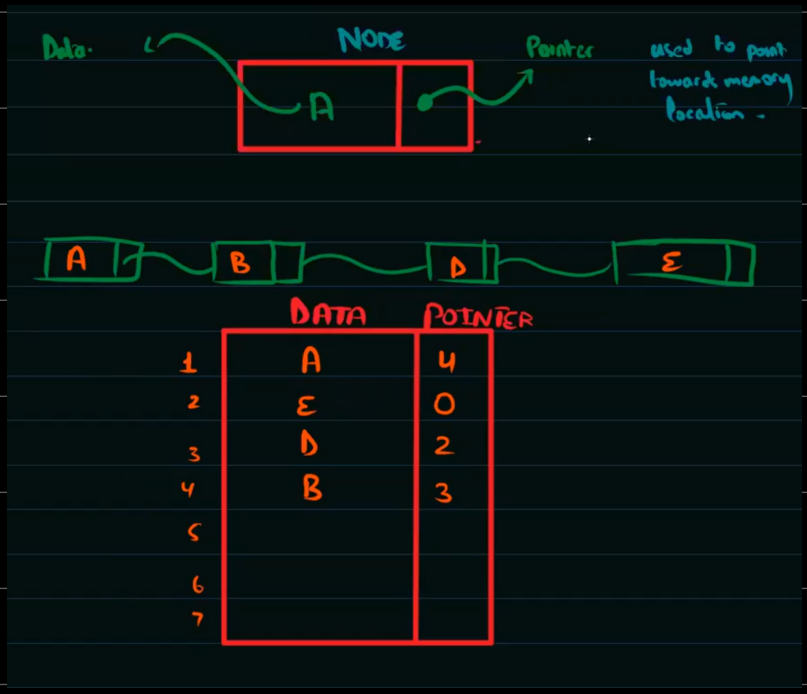
**2** The number of cars that cross a bridge is recorded each hour. This number is placed in a circu queue before being processed.

**(a)** The queue is stored as an array, `NumberQueue`, with eight elements. The funct `AddToQueue` adds a number to the queue. `EndPointer` and `StartPointer` are glo variables.

Complete the following **pseudocode** algorithm for the function `AddToQueue`.

```
FUNCTION AddToQueue(Number : INTEGER) RETURNS BOOLEAN
    DECLARE TempPointer : INTEGER
    CONSTANT FirstIndex = 0
    CONSTANT LastIndex = .....
    TempPointer ← EndPointer + 1
    IF ..... > LastIndex
        THEN
            TempPointer ← .....
    ENDIF
    IF TempPointer = StartPointer
        THEN
            RETURN .....
        ELSE
            EndPointer ← TempPointer
            NumberQueue[EndPointer] ← .....
            RETURN TRUE
    ENDIF
ENDFUNCTION
```

# Linked List



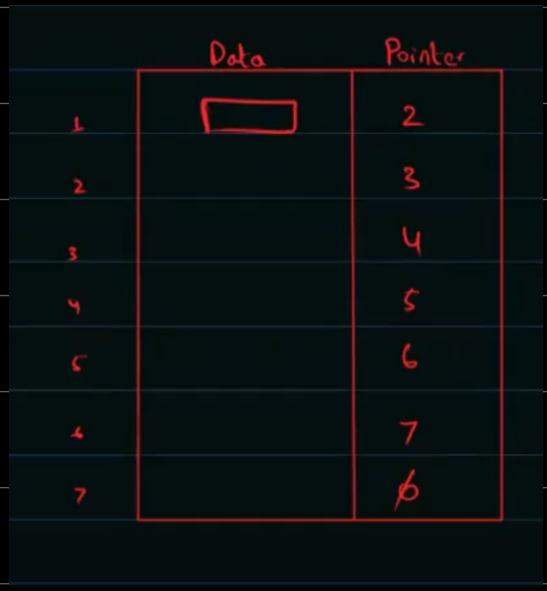

# Creating Linked List

CONSTANT nullpointer = 0 (If index starts with 0, then -1)

TYPE ListNode

DECLARE Data: STRING ↱ data type of data to be stored

```
    DECLARE Pointer: INTEGER
END TYPE

DECLARE List : ARRAY[1:7] OF ListNode

PROCEDURE Initialize()
    Startpointer ← nullpointer      (As there is no item in the list)
    Freelistpointer ← 1

    FOR Index ← 1 TO 6
        List[Index].Pointer ← Index +1
    END FOR
    List[7].Pointer ← nullpointer
END PROCEDURE
```



# Accessing Pointers

- Point ← List[Point].Pointer

# Searching Linked List

**(b)** The club also considers storing the data in the order in which it receives the scores as a linked list in a 1D array of records.

The following pseudocode algorithm searches for an element in the linked list.

Complete the **six** missing sections in the algorithm.

```
FUNCTION FindElement(Item : INTEGER) RETURNS Integer
    currentpointer ← RootPointer
    WHILE CurrentPointer <> NullPointer                    until reached end of list
        IF List[CurrentPointer].Data <> Item
            THEN
                CurrentPointer ← List[currentpointer].Pointer
            ELSE
                RETURN CurrentPointer
        ENDIF
    ENDWHILE
    CurrentPointer ← NullPointer
    Return CurrentPointer
ENDFUNCTION
```

[6]

# Inserting a Node (Ascending)

```
PROCEDURE InsertNode(Newitem: STRING)
    IF Freelistptr <> nullpointer        (If equal to null pointer, then linked list is full)
        THEN
Storing     Newnodeptr ← Freelistpointer
            List[NewNodepointer].Data ← Newitem
            Freelistpointer ← list[Freelistptr].Pointer

            PreviousNodePointer ← startPointer
            ThisNodepointer ← Startpointer
            WHILE ThisNodePointer <> NullPointer AND List[ThisNodepointer].Data < Newitem DO
Searching       Previousnodepointer ← ThisNodepointer
correct         Thisnodepointer ← List[Thisnodepointer].Pointer
position    END WHILE
```

End of list

```
Changing    IF  Previousnodepointer = Startpointer
pointers        THEN
for correct         List[Newnodepointer].pointer ← Startpointer
placement           Startpointer ← Newnodepointer
of nodes        ELSE
                    List[Newnodepointer].Pointer ← List[previouspointer].pointer
                    List[previousnodepointer].pointer ← Newnodepointer
            END IF
    END IF
END PROCEDURE
```

# Deleting Node

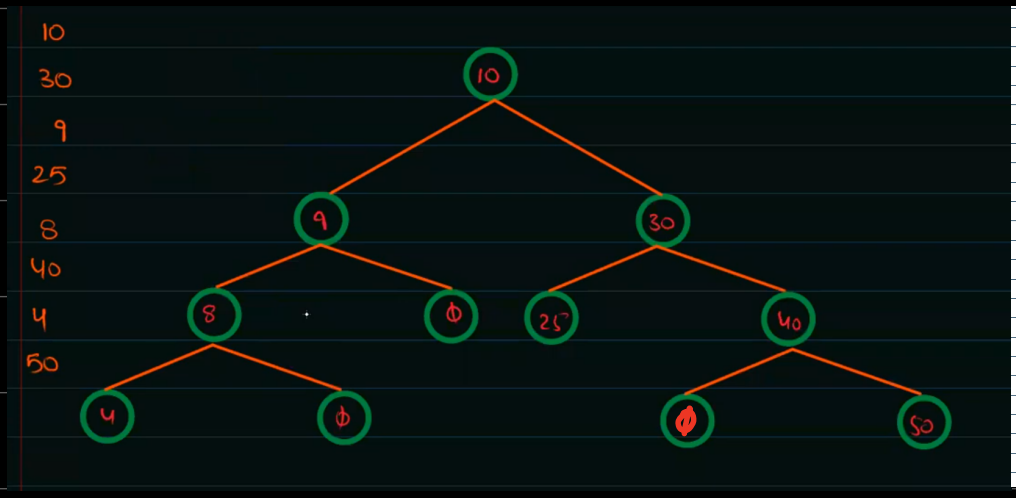
```
PROCEDURE  DeleteNode(Dataitem : STRING)
        ThisNodepointer ← startpointer

        WHILE (ThisNodepointer <> nullpointer) AND (List[ThisNodepointer].Data <> Dataitem) DO
            previousnodepointer ← Thisnodepointer
            ThisNodepointer ← List[ThisNodePointer].pointer
        END WHILE

        IF ThisNodePointer <> nullpointer
           THEN
              IF ThisNodePointer = StartPointer
                 THEN
                    Startpointer ← List[StartPointer].Pointer
                 ELSE
```

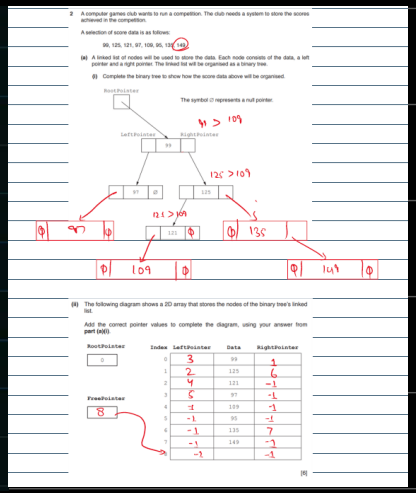
```
        List[previousNodePointer].Pointer ← List[ThisNodePointer].Pointer
    END IF
END IF
List[ThisNodePointer].Pointer ← FreeListPointer
FreeListPointer ← ThisNodePointer
END PROCEDURE
```

# Binary Tree

**Key**

- Always start comparing from the root
- If bigger value than root => Move right
- If smaller value than root => Move Left



Q 23 , 17

# Binary Tree Construction

```
TYPE Node
    DECLARE Data : STRING
    DECLARE RightPointer : INTEGER
    DECLARE LeftPointer : INTEGER
END TYPE

DECLARE Tree : ARRAY[1:8] OF Node
FreePointer ← 1
RootPointer ← 0

FOR Index ← 1 TO 7
    Tree[Index].LeftPointer ← Index + 1
    Tree[index].RightPointer ← 0
END FOR
```

* left pointers are used to create link.

Tree[8].LeftPointer ← 0

Tree[8].RightPointer ← 0

# Insertion

(d) A program is to be written to implement the tree ADT. The variables and procedures to be used are listed below:

| Identifier | Data type | Description |
|---|---|---|
| Node | RECORD | Data structure to store node data and associated pointers. |
| LeftPointer | INTEGER | Stores index of start of left subtree. |
| RightPointer | INTEGER | Stores index of start of right subtree. |
| Data | STRING | Data item stored in node. |
| Tree | ARRAY | Array to store nodes. |
| NewDataItem | STRING | Stores data to be added. |
| FreePointer | INTEGER | Stores index of start of free list. |
| RootPointer | INTEGER | Stores index of root node. |
| NewNodePointer | INTEGER | Stores index of node to be added. |
| CreateTree() | | Procedure initialises the root pointer and free pointer and links all nodes together into the free list. |
| AddToTree() | | Procedure to add a new data item in the correct position in the binary tree. |
| FindInsertionPoint() | | Procedure that finds the node where a new node is to be added.<br>Procedure takes the parameter NewDataItem and returns two parameters:<br>• Index, whose value is the index of the node where the new node is to be added<br>• Direction, whose value is the direction of the pointer ("Left" or "Right") |

```
PROCEDURE AddToTree(BYVALUE NewDataItem : STRING)
// if no free node report an error
  IF FreePointer = -1
    THEN
      OUTPUT("No free space left")
    ELSE // add new data item to first node in the free list
      NewNodePointer ← FreePointer
Storing <= Tree[NewNodepointer].Data = NewDataItem
      // adjust free pointer
      FreePointer ← Tree[Freepointer].Leftpointer (increment)
      // clear left pointer
      Tree[NewNodePointer].LeftPointer ← -1
      // is tree currently empty ?
      IF Rootpointer = -1
        THEN // make new node the root node
          Rootpointer = Newnodeptr
        ELSE  // find position where new node is to be added
          Index ← RootPointer
          CALL FindInsertionPoint(NewDataItem, Index, Direction)
          IF Direction = "Left"
            THEN  // add new node on left
              Tree[Index].leftpointer = Newnodepointer
            ELSE  // add new node on right
              Tree[Index].rightpointer = Newnode
          ENDIF
```

```
              Tree[Index].rightpointer = Newnode
          ENDIF
      ENDIF
ENDIF
```

# Searching For The Correct Position

```
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
```

# Finding Node

```
FUNCTION FindNode (SearchItem : STRING) RETURNS INTEGER

        ThisNodePointer ← RootPointer


        WHILE ThisNodePointer <> NullPointer AND Tree[ThisNodePointer].Data <> SearchItem DO

            IF Tree[ThisNodePointer].Data > SearchItem

                THEN
```

```
            ThisNodePointer ← Tree[ThisNodePointer].LeftPointer
        ELSE
            ThisNodePointer ← Tree[ThisNodePointer].RightPointer
        END IF
    END WHILE
    RETURN ThisNodePointer
END FUNCTION
```

# Dictionary

- An abstract data type which stores data in the form of key and value pair

- We use to index other data types / array by numerical index
- Dictionaries are indexed by their keys
- Numbers, Integers, Real, String ] can be used as keys
- Numbers, Integers, Strings, linked List, Stack, queue ] can be used as values
- Record Datatype (Declare key and value)

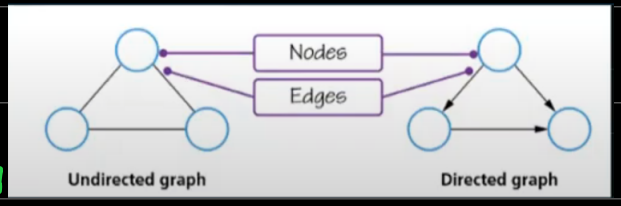
## Restrictions

* key can store multiple values for searching

- Every key can only appear once within the dictionary
- Each key can only have one value.

# Graphs

- A graph is a non-linear data structure consisting of nodes and edges. This is an ADT which is used to implement directed or undirected graphs.




- Edges having direction from one node to another, it is a directed graph.

Q- Describe the uses of graphs

- Bus routes, where nodes are bus stops and edges connect two stops next to each other
- Website where each web-page is a node and edges show the links b/w webpages
- Social Media Networks where a node is a person's information and edges connect two friends.

**Weight:** Each edge may have a weight. Quantity (Distance, time)

**Path:** List of nodes connected by edges b/w two given nodes

**Cycle:** List of nodes that return to the same node.