

Papers Dock

PYTHON

9618

ABSTRACT DATATYPE

Abstract Datatype

An Abstract Datatype is a collection of data and set of operations on that data

Stack

Queue

Linked List

Binary Tree

Papers Dock

PYTHON

9618

STACK

Stacks

A list containing several items operating on the last in first out principle (LIFO).

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

Push("Taha")
Push("Ali")
Push("Amjad")
Push("Bano")
Pop()
Pop()
Push("Qasim")



A stack can be implemented by using the concept of Arrays and Stackpointer

Declare Names : ARRAY [0 : 4] OF STRING

← stackpointer



Push Procedure

```
Names = [""] * 5
stackpointer = 0
def Push(value):
    global stackpointer
    if stackpointer > 4:
        print("Stack Full")
    else:
        Names[stackpointer] = value
        stackpointer = stackpointer + 1
```

Pop Procedure

```
def Pop():  
    global stackpointer  
    if stackpointer == 0:  
        print("The Stack Is Empty")  
    else:  
        stackpointer = stackpointer - 1  
        print(Names[stackpointer])
```


1 A program needs to use a stack data structure. The stack can store up to 10 integer elements.

A 1D array `StackData` is used to store the stack globally. The global variable `StackPointer` points to the next available space in the stack and is initialised to 0.

(a) Write program code to declare the array and pointer as global data structures. Initialise the pointer to 0.

```
#DECLARE stackData : ARRAY [0:9] OF INTEGER
#DECLARE stackpointer : INTEGER
global StackData
global stackpointer
StackData = [0] * 10
StackPointer = 0
```



(b) Write a procedure to output all 10 elements in the stack **and** the value of `StackPointer`.

```
def PrintArray():  
    global StackData  
    global StackPointer  
    print(StackPointer)  
    for x in range(0, 10):  
        print(StackData[x])
```

- (c) The function `Push()` takes an integer parameter and returns `FALSE` if the stack is full. If the stack is not full, it puts the parameter value on the stack, updates the relevant pointer and returns `TRUE`.

Write program code for the function `Push()`.

```
def Push(Number):  
    global StackData  
    global StackPointer  
    if StackPointer > 9:  
        return False  
    else:  
        StackData[StackPointer] = Number  
        StackPointer = StackPointer + 1  
    return True
```

(d) (i) Edit the main program to test the `Push()` function. The main program needs to:

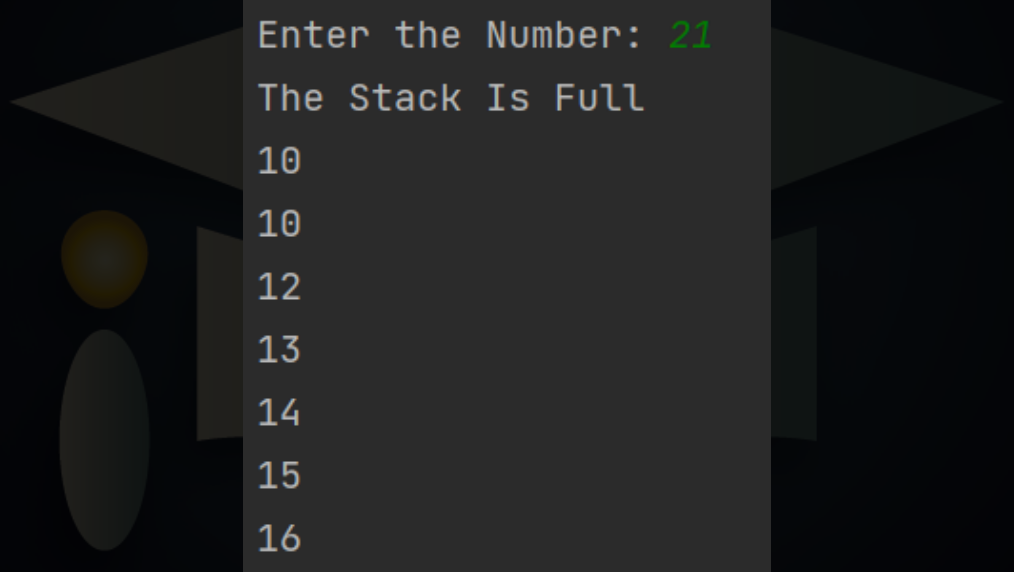
- allow the user to enter 11 numbers and attempt to add these to the stack
- output an appropriate message when a number is added to the stack
- output an appropriate message when a number is not added to the stack if it is full
- output the contents of the stack after attempting to add all 11 numbers.

```
for x in range(0,11):  
    value = int(input("Enter the Number: "))  
    if Push(value) == True:  
        print("Successfully Added")  
    else:  
        print("The Stack Is Full")
```

```
PrintArray()
```


(ii) Test your program from **part 1(d)(i)** with the following 11 inputs:

11 12 13 14 15 16 17 18 19 20 21



```
Enter the Number: 19
Successfully Added
Enter the Number: 20
Successfully Added
Enter the Number: 21
The Stack Is Full
10
10
12
13
14
15
16
17
18
19
20
```

(e) The function `Pop()` returns `-1` if the stack is empty. If the stack is not empty, it returns the element at the top of the stack and updates the relevant pointer.

(i) Write program code for the function `Pop()`.

```
def Pop():  
    global StackData  
    global StackPointer  
    if StackPointer == 0:  
        return -1  
    else:  
        StackPointer = StackPointer - 1  
        temp = StackData[StackPointer]  
        return temp
```

Papers Dock

PYTHON

9618

QUEUE

Queue

A list containing several items operating on the first in first out principle (FIFO).

The first item added is the first item remove from the queue

In queue the data is added from the rear end by using the EndPointer and removed from the front by using the StartPointer



```
Enqueue("Taha")  
Enqueue("Ali")  
Enqueue("Amjad")  
Enqueue("Bano")  
Dequeue()  
Dequeue()  
Enqueue("Qasim")
```

Linear Enqueue

First Condition will check if the Queue is Empty by comparing it with the Max Index value. The item will be inserted by using the TailPointer. After inserting the item we will check if the Headpointer from where the item is removed is still pointing -1 then we have to point to the first value which was added

Linear Enqueue

```
# DECLARE Names : ARRAY [0:9] OF String
Names = [""] * 10
HeadPointer = -1
TailPointer = 0

def Enqueue(Data):
    global Names
    global HeadPointer
    global TailPointer

    if TailPointer < 10:
        Names[TailPointer] = Data
        TailPointer = TailPointer + 1

        if HeadPointer == -1:
            HeadPointer = 0

    else:
        print("Queue Is Full")
```

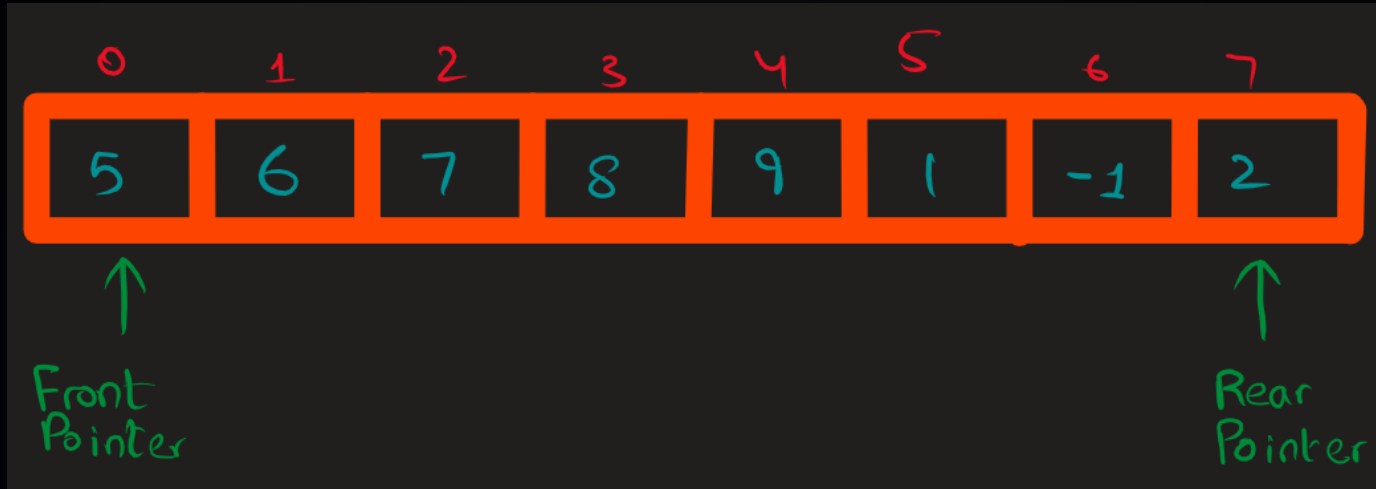
Linear Dequeue

- There are 2 ways that queue is considered to be empty
- 1) If Headpointer is pointing towards -1 this will only happen when there was no value enqueued in the Queue
 - 2) If the Headpointer is equal to the tail pointer which means that all the values which were Enqueued in Queue are now removed.

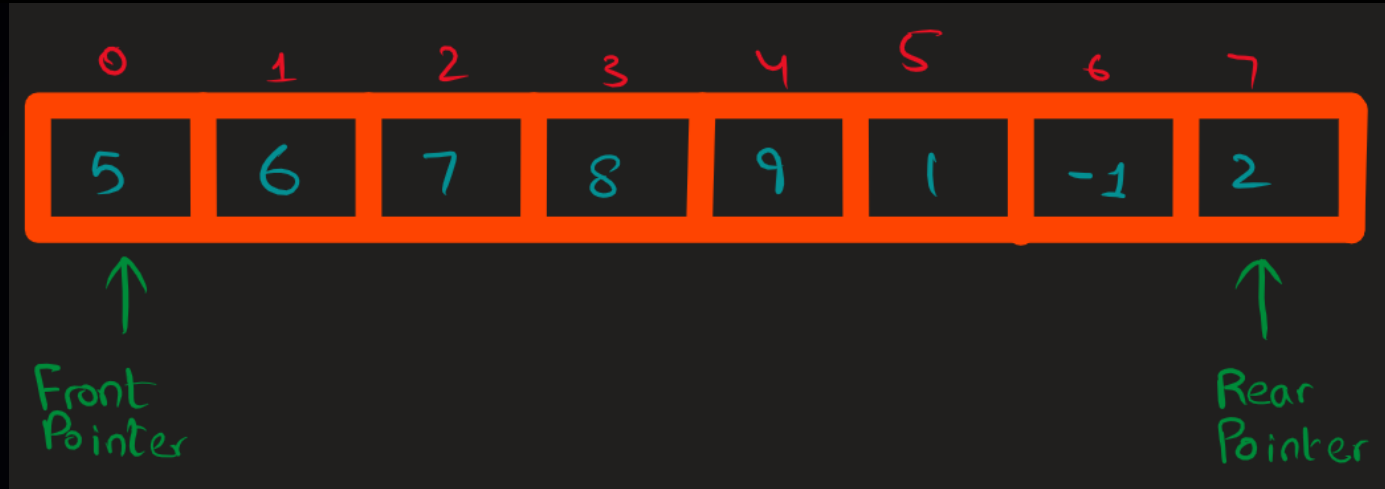
Linear Dequeue

```
def Dequeue():  
    global Names  
    global HeadPointer  
    global TailPointer  
  
    if HeadPointer == -1:  
        print("Queue Is Empty")  
    else:  
        item = Names[HeadPointer]  
        print(item)  
        HeadPointer = HeadPointer + 1  
  
    if HeadPointer == TailPointer:  
        TailPointer = 0  
        HeadPointer = -1
```

Linear Vs Circular



The condition for a linear queue being full is that rearpointer or the endpointer should point towards upperbound or the max index

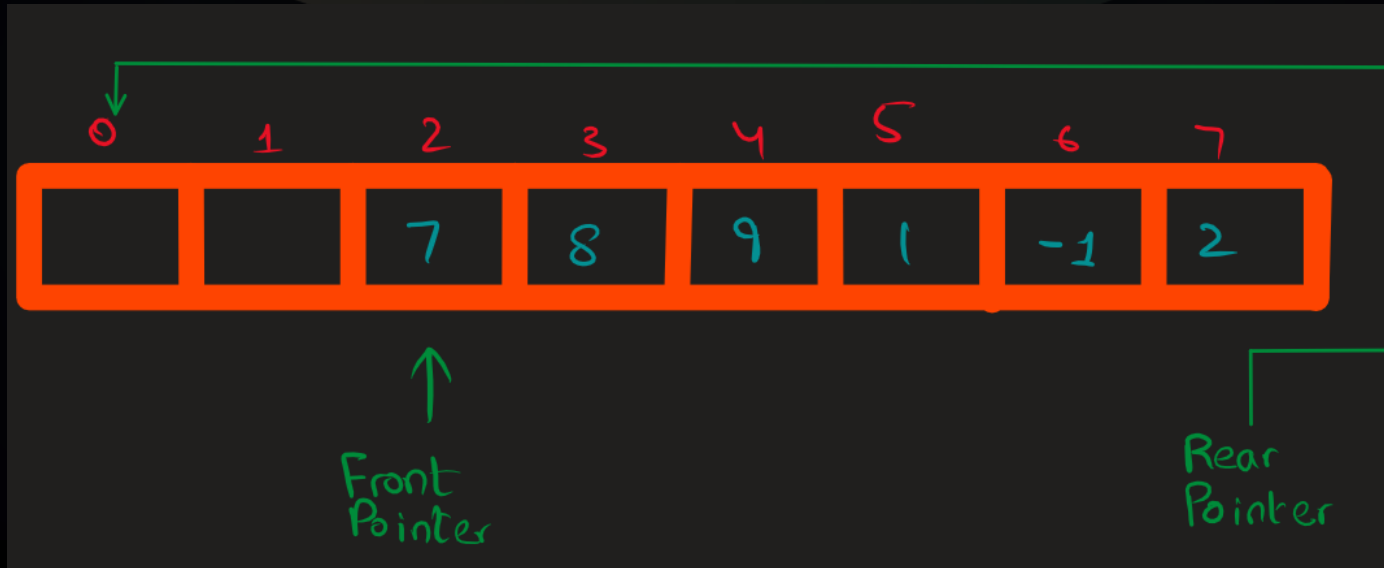


Dequeue()
Dequeue()



According to the condition of queue being full is still true so if you want to enqueue a value it will still print "The Queue Is Full"

Circular Queue



Circular Queue

- 3 A program uses a circular queue to store strings. The queue is created as a 1D array, `QueueArray`, with 10 string items.

The following data is stored about the queue:

- the head pointer initialised to 0
- the tail pointer initialised to 0
- the number of items in the queue initialised to 0.

(a) Declare the array, head pointer, tail pointer and number of items.

If you are writing in Python, include attribute declarations using comments.

- (b) The function `Enqueue` is written in pseudocode. The function adds `DataToAdd` to the queue. It returns `FALSE` if the queue is full and returns `TRUE` if the item is added.

The function is incomplete, there are **five** incomplete statements.

```
FUNCTION Enqueue(BYREF QueueArray[] : STRING, BYREF HeadPointer : INTEGER,
                 BYREF TailPointer : INTEGER, NumberItems : INTEGER,
                 DataToAdd : STRING) RETURNS BOOLEAN

    IF NumberItems = ..... THEN

        RETURN .....

    ENDIF

    QueueArray[.....] ← DataToAdd

    IF TailPointer >= 9 THEN

        TailPointer ← .....

    ELSE

        TailPointer ← TailPointer + 1

    ENDIF

    NumberItems ← NumberItems .....

    RETURN TRUE

ENDFUNCTION
```

Write program code for the function `Enqueue()`.

Circular Enqueue

#Can not passed them as byreference so declared them as global variables

```
def Enqueue(InputData):  
    global QueueArray  
    global HeadPointer  
    global TailPointer  
    global NumberOfItems  
  
    if NumberOfItems >= 10:  
        return False  
  
    QueueArray[TailPointer] = InputData  
  
    if TailPointer >= 9:  
        TailPointer = 0  
    else:  
        TailPointer = TailPointer + 1  
    NumberOfItems = NumberOfItems + 1  
    return True
```

(c) The function `Dequeue ()` returns "FALSE" if the queue is empty, or it returns the next data item in the queue.

Write program code for the function `Dequeue ()`.

Circular Dequeue

```
# Can not pass as by reference, so we will use global variables and array

def Dequeue():
    global QueueArray
    global HeadPointer
    global TailPointer
    global NumberOfItems

    if NumberOfItems == 0:
        return False
    else:
        value = QueueArray[HeadPointer]
        HeadPointer = HeadPointer + 1

        if HeadPointer > 9:
            HeadPointer = 0
        NumberOfItems = NumberOfItems - 1
        return value
```

Papers Dock

PYTHON

9618

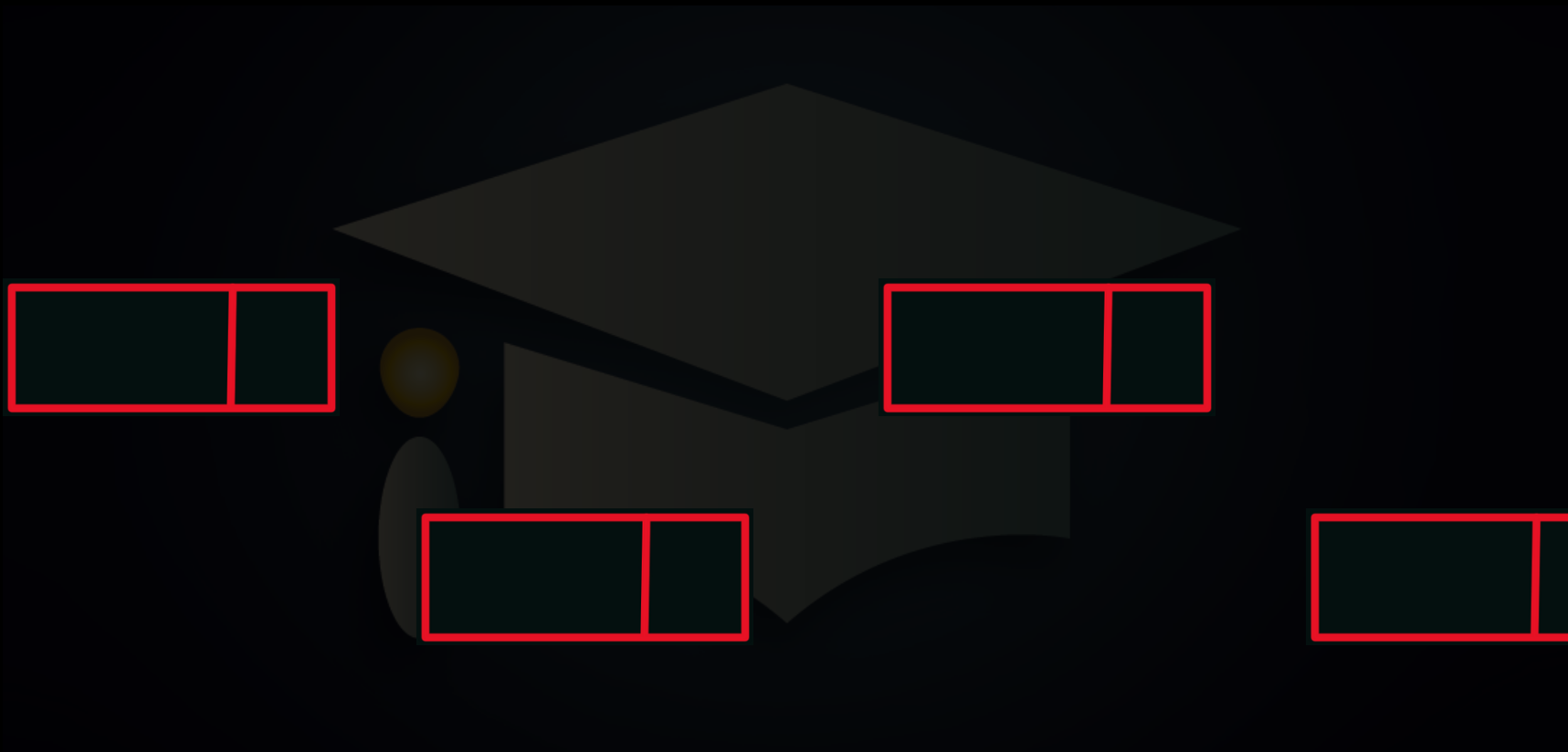
LINKED LIST

Data.

NODE

Pointer





Ordered Linked List And Unordered Linked List

A linked list is a data structure used to store a collection of items where each item is linked to the next one using pointers. There are two types of linked lists: ordered and unordered.

An ordered linked list is a list where the elements are arranged in ascending or descending.

an unordered linked list is a list where the elements are not sorted in any particular order.

Increment In Pointer

S.P = 2

point = List [point] . pointer



2



7



5



9

```
point = S.P
```

```
point = List [ point ] . pointer
```



Point

- 1 An unordered linked list uses a 1D array to store the data.

Each item in the linked list is of a record type, `node`, with a field `data` and a field `nextNode`.

The current contents of the linked list are:

`startPointer` 0

`emptyList` 5

Index	data	nextNode
0	1	1
1	5	4
2	6	7
3	7	-1
4	2	2
5	0	6
6	0	8
7	56	3
8	0	9
9	0	-1

(a) The following is pseudocode for the record type `node`.

```
TYPE node
```

```
    DECLARE data : INTEGER
```

```
    DECLARE nextNode : INTEGER
```

```
ENDTYPE
```

Write program code to declare the record type `node`.

Record DataType For Linked List

```
class node:  
    # Public Data : INTEGER  
    # Public nextNode : INTEGER  
    def __init__(self, DataP, nextNodeP):  
        self.Data = DataP  
        self.nextNode = nextNodeP
```

Index	data	nextNode
0	1	1
1	5	4
2	6	7
3	7	-1
4	2	2
5	0	6
6	0	8
7	56	3
8	0	9
9	0	-1

(b) Write program code for the main program.

Declare a 1D array of type `node` with the identifier `linkedList`, and initialise it with the data shown in the table on page 2. Declare the pointers.

```
# DECLARE linkedList : ARRAY [0:9] OF node
```

```
linkedList = [node(1,1), node(5,4), node(6,7), node(7,-1), node(2,2), node(0,6), node(0,8), node(56,3), node(0,9), node(0,-1)]
```

```
startPointer = 0
```

```
emptyList = 5
```

S.P = 2



2



7



5



9

Unordered Linked List Insertion

The function, `addNode()`, takes the linked list and pointers as parameters, then takes as input the data to be added to the end of the `linkedList`.

The function adds the node in the next available space, updates the pointers and returns `True`. If there are no empty nodes, it returns `False`.

(i) Write program code for the function `addNode()`.

```

def addNode(currentPointer):
    global linkedList
    global emptyList
    data = int(input("Enter the data to add: "))
    # Check If Array FULL or Not
    if emptyList < 0 or emptyList > 9:
        return False
    else:
        freelist = emptyList
        emptyList = linkedList[emptyList].nextNode
        # Create node object
        newNode = node(data, -1)
        # Store it where the freelist is pointing
        linkedList[freelist] = newNode

        previousPointer = 0

        while currentPointer != -1:
            previousPointer = currentPointer
            currentPointer = linkedList[currentPointer].nextNode

        linkedList[previousPointer].nextNode = freelist
        return True

```

Linked List Deletion

S.P = 2



```

def deleteNode():
    global linkedList
    global emptyList
    global startPointer

    currentPointer = startPointer

    data = int(input("Enter the data to delete: "))

    previousPointer = 0
    while currentPointer != -1 and linkedList[currentPointer].data != data:
        previousPointer = currentPointer
        currentPointer = linkedList[currentPointer].nextNode

    if currentPointer == -1:
        return False
    else:
        # Update the pointers to remove the node
        if currentPointer == startPointer:
            startPointer = linkedList[startPointer].nextNode
        else:
            linkedList[previousPointer].nextNode = linkedList[currentPointer].nextNode

        linkedList[currentPointer].data = 0
        linkedList[currentPointer].nextNode = emptyList
        emptyList = currentPointer

    return True

```

The procedure `outputNodes()` takes the array and `startPointer` as parameters. The procedure outputs the data from the linked list by following the `nextNode` values.

(i) Write program code for the procedure `outputNodes()`.

```
def outputNodes():  
    global linkedList  
    currentPointer = startPoint  
    while currentPointer != -1:  
        print(str(linkedList[currentPointer].Data))  
        currentPointer = linkedList[currentPointer].nextNode  
  
outputNodes()
```

Searching In A Linked List

For searching a particular element or value you are supposed to use the concept of increment in a linked list and compare the value of each element in the linked list with the value you are searching and return the current pointer as it will be used to increment

```
def Finditem(currentpointer, Searchvalue):  
    while currentpointer != -1:  
        if linkedList[currentpointer].Data != Searchvalue:  
            currentpointer = linkedList[currentpointer].nextNode  
        else:  
            return currentpointer  
  
    currentpointer = -1  
    return currentpointer
```


Inserting In An Ordered Linked List

Inserting in an ordered link list is to some extent similar to the Insertion In an unordered linked list. But in this you first have to find the correct position and then insert the node

S.P = 2



2



7



5



9

```

def OrderedInsertion(currentpointer):
    global linkedList
    global emptyList
    global startPoint

    datatoininsert = int(input("Enter the data to add: "))

    if emptyList < 0 or emptyList > 9:
        return False
    else:
        freelist = emptyList
        emptyList = linkedList[emptyList].nextNode

        newNode = node(datatoininsert, -1)

        linkedList[freelist] = newNode

        while currentpointer != -1 and linkedList[currentpointer].Data < datatoininsert:
            previousPointer = currentpointer
            currentpointer = linkedList[currentpointer].nextNode

        if currentpointer == startPoint:
            linkedList[freelist].nextNode = startPoint
            startPoint = freelist
        else:
            linkedList[freelist].nextNode = linkedList[previousPointer].nextNode
            linkedList[previousPointer].nextNode = freelist

    return True

```

Papers Dock

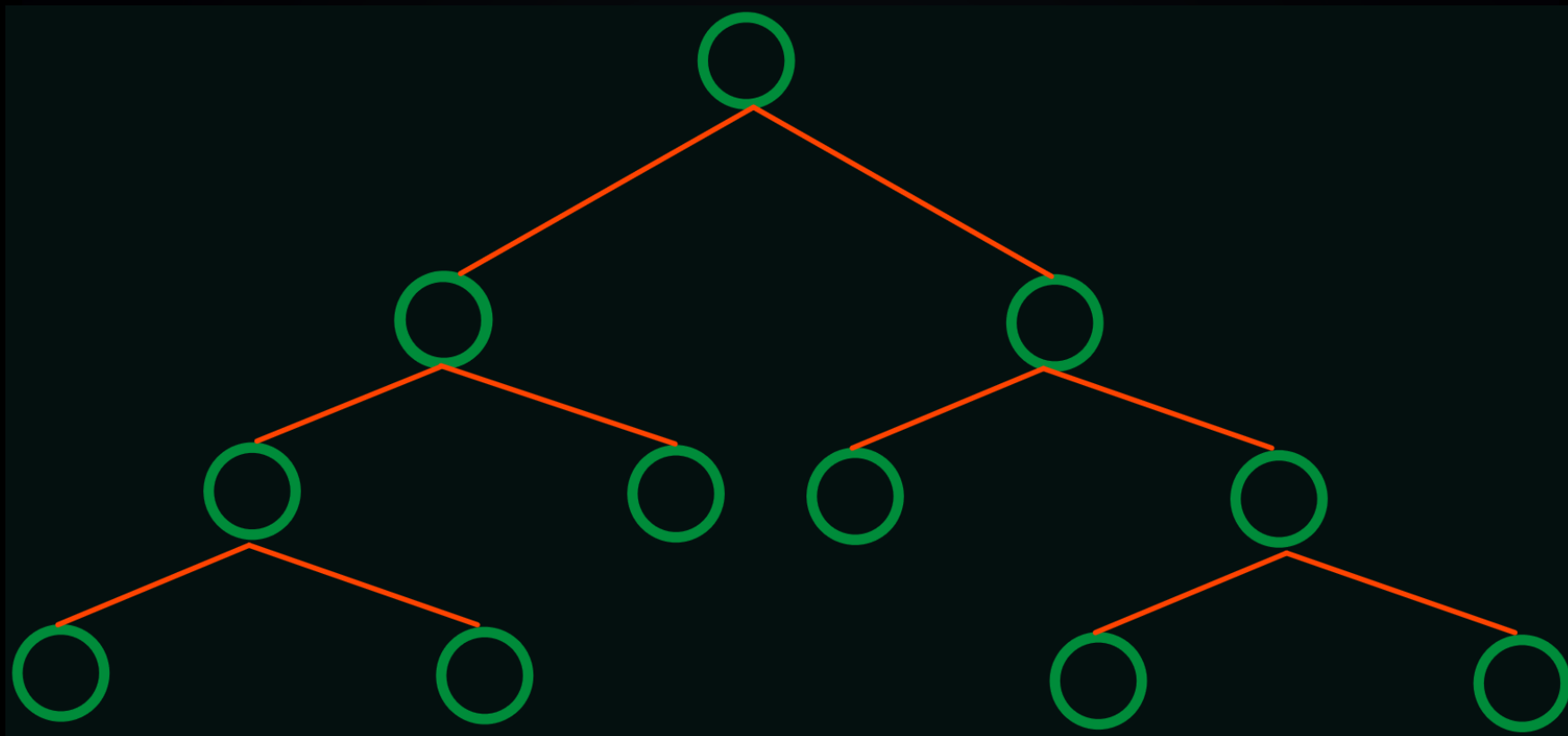
PYTHON

9618

BINARY TREE

Binary Tree

A binary tree is a type of data structure in which each node has at most two children, referred to as the left child and the right child. The nodes are arranged in a hierarchical structure, with a root node at the top and the leaves at the bottom



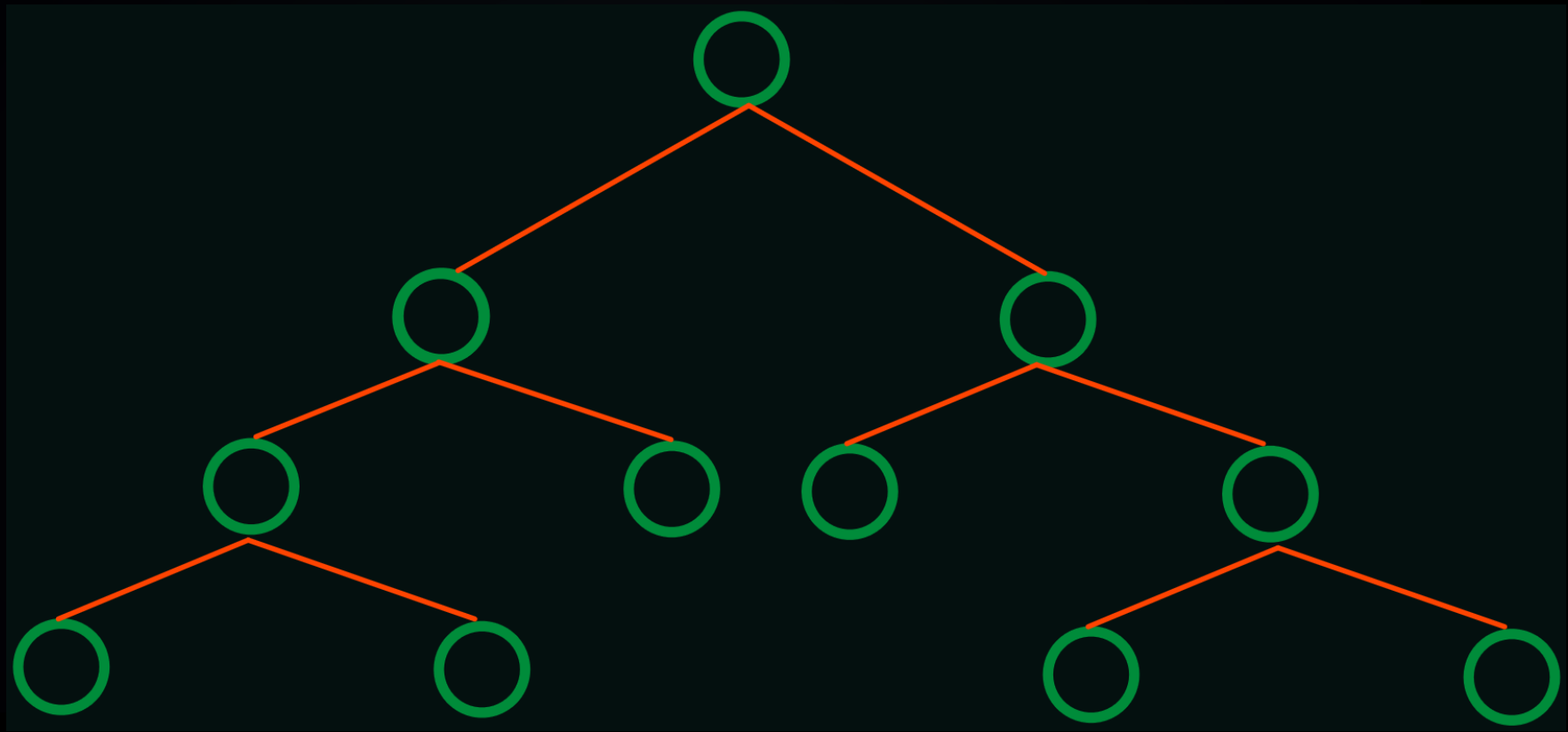
Adding A Value To A Binary Tree

Always start comparing from the root

If the value that you want to insert is bigger than the root then move right

If the value is smaller than the root then move left

10
30
9
25
8
40
4
50

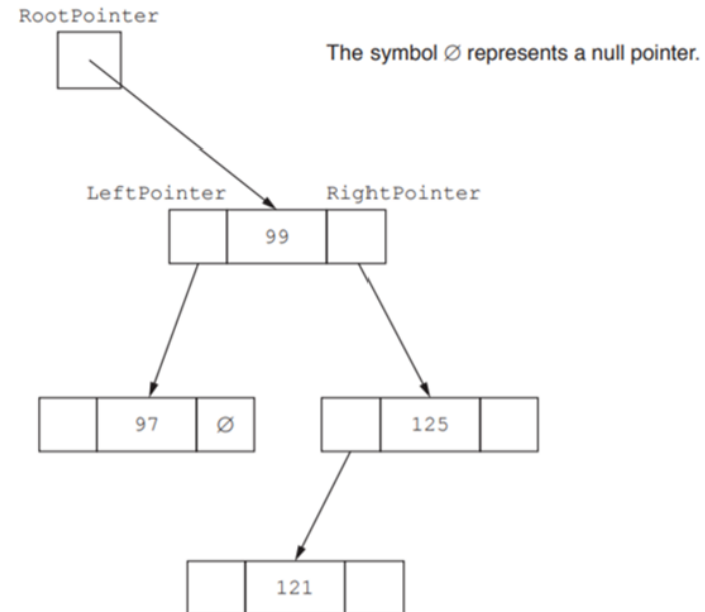


- 2 A computer games club wants to run a competition. The club needs a system to store the scores achieved in the competition.

A selection of score data is as follows:

99, 125, 121, 97, 109, 95, 135, 149

- (a) A linked list of nodes will be used to store the data. Each node consists of the data, a left pointer and a right pointer. The linked list will be organised as a binary tree.
- (i) Complete the binary tree to show how the score data above will be organised.



Tree Traversal

Tree traversal refers to the process of visiting and examining each node in a tree data structure in a specific order.

1) INORDER

2) PREORDER

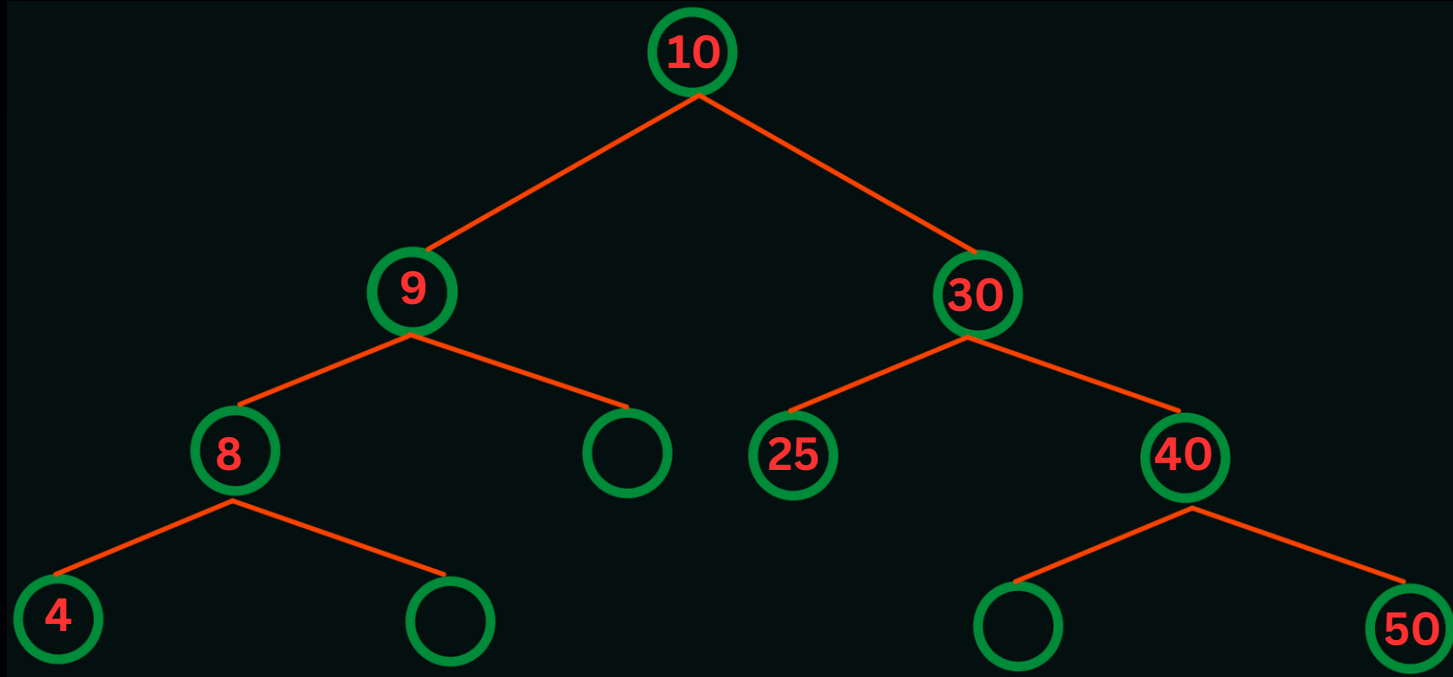
3) POSTORDER



INORDER : LEFT ROOT RIGHT

PREORDER : ROOT LEFT RIGHT

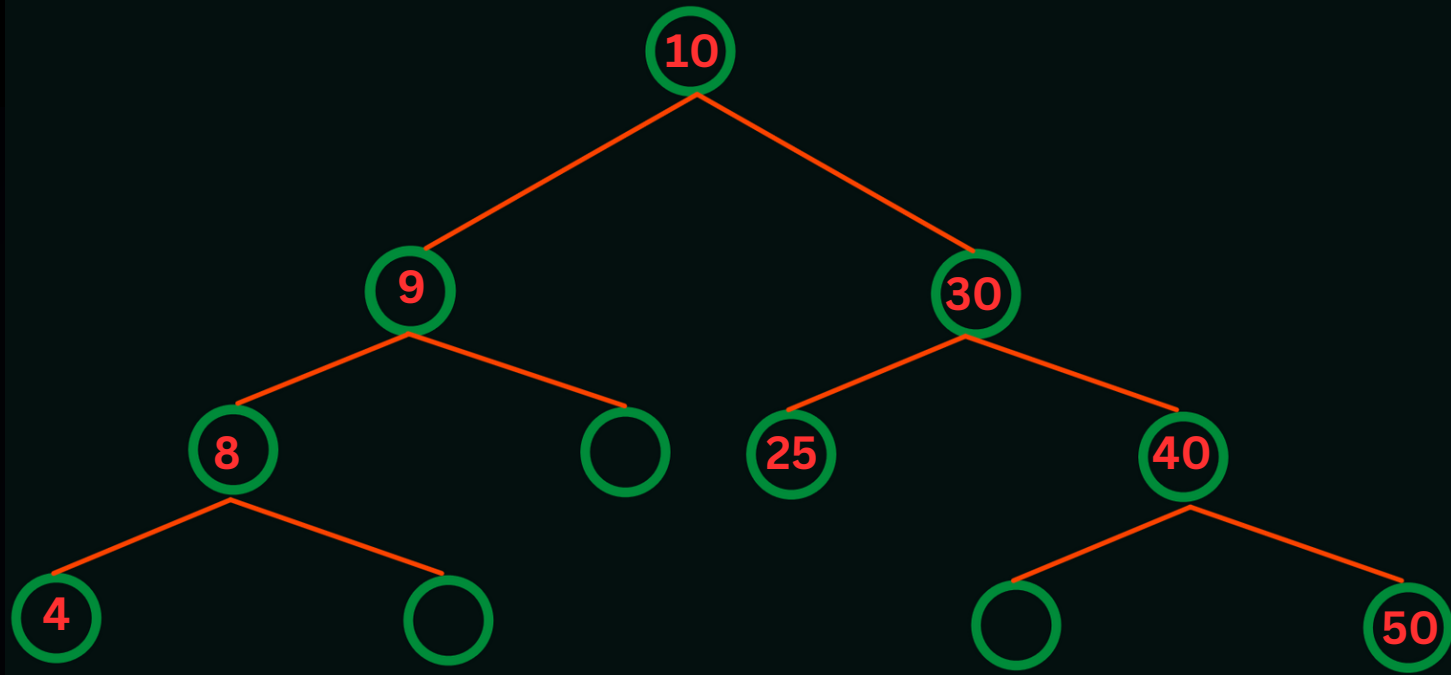
POSTORDER : LEFT RIGHT ROOT



INORDER :

PREORDER :

POSTORDER :



INORDER : 4 8 9 10 25 30 40 50

PREORDER : 10 9 8 4 30 25 40 50

POSTORDER : 4 8 9 25 50 40 30 10

Creating A 2D Array For Binary Tree

An ordered binary tree stores integer data in ascending numerical order.

The data for the binary tree is stored in a 2D array with the following structure:

	LeftPointer	Data	RightPointer
Index	[0]	[1]	[2]
[0]	1	10	2
[1]	-1	5	-1
[2]	-1	16	-1

Each row in the table represents one node on the tree.
The number -1 represents a null pointer.

(a) The 2D array, `ArrayNodes`, is declared with space for 20 nodes.

Each node has a left pointer, data and right pointer.

The program also initialises the:

- `RootPointer` to -1 (null); this points to the first node in the binary tree
- `FreeNode` to 0; this points to the first empty node in the array.

Write program code to declare `ArrayNodes`, `RootPointer` and `FreeNode` in the main program.

```
ArrayNodes = [[0] * 3 for x in range(20)]  
RootPointer = -1  
FreeNode = 0
```


Adding A Node In Binary Tree

First Check If there is space in your binary tree or not by comparing the value of freenode if it has reached the max index then No Space

Then Check if the root pointer is still pointing towards -1 that means that there is no item in the binary tree and just change the value of root pointer to first index which is 0

If the root pointer is not at -1 that means now you have to compare the Data with the root value if the Data you want to insert is less than the value at root then move towards left side and check if the left pointer of the root is pointing towards -1 that means now you can change the value of the left pointer of root to the location where you inserted the data and if it is not at -1 that means you have to increment the left pointer

CurrentPointer = Array[Currentpointer].LeftPOinter



same thing will be done if the Data is greater than the value at Root but you will move towards right

```

def AddNode():
    global ArrayNodes
    global RootPointer
    global FreeNode

    NodeData = int(input("Enter the Data: "))

    if FreeNode <= 19:
        ArrayNodes[FreeNode][0] = -1
        ArrayNodes[FreeNode][1] = NodeData
        ArrayNodes[FreeNode][2] = -1

        # First Check If you want to store it in the first node
        if RootPointer == -1:
            RootPointer = 0
        else:
            Placed = False
            CurrentPointer = RootPointer
            while Placed == False:
                if NodeData < ArrayNodes[CurrentPointer][1]:
                    if ArrayNodes[CurrentPointer][0] == -1:
                        ArrayNodes[CurrentPointer][0] = FreeNode
                        Placed = True
                    else:
                        CurrentPointer = ArrayNodes[CurrentPointer][0]
                else:
                    if ArrayNodes[CurrentPointer][2] == -1:
                        ArrayNodes[CurrentPointer][2] = FreeNode
                        Placed = True
                    else:
                        CurrentPointer = ArrayNodes[CurrentPointer][2]

            FreeNode = FreeNode + 1
    else:
        print("Tree Is Full")

```

Searching In A Binary Tree

```
def FindNode(SearchItem):  
    CurrentPointer = RootPointer  
  
    while CurrentPointer != -1 and ArrayNodes[CurrentPointer][1] != SearchItem:  
        if SearchItem < ArrayNodes[CurrentPointer][1]:  
            CurrentPointer = ArrayNodes[CurrentPointer][0]  
        else:  
            CurrentPointer = ArrayNodes[CurrentPointer][2]  
  
    print(CurrentPointer)
```



INORDER : LEFT ROOT RIGHT

PREORDER : ROOT LEFT RIGHT

POSTORDER : LEFT RIGHT ROOT

Inorder Traversal

```
def InOrder(Array, Root):  
    if Array[Root][0] != -1:  
        InOrder(Array, Array[Root][0])  
    print(str(Array[Root][1]))  
    if Array[Root][2] != -1:  
        InOrder(Array, Array[Root][2])
```

Preorder Traversal

```
def PreOrder(Array, Root):  
    print(str(Array[Root][1]))  
    if Array[Root][0] != -1:  
        PreOrder(Array, Array[Root][0])  
    if Array[Root][2] != -1:  
        PreOrder(Array, Array[Root][2])
```


Postorder Traversal

```
def PostOrder(Array, Root):  
    if Array[Root][0] != -1:  
        PostOrder(Array, Array[Root][0])  
    if Array[Root][2] != -1:  
        PostOrder(Array, Array[Root][2])  
    print(str(Array[Root][1]))
```