Practical - 1

Aim: To search a number from the list using linear unsorted.

Theory: The Process of Identifying or finding a Particular record is called searching.

There are two types of search
↳ Linear Search
↳ Binary search

The Linear search is further classified as:

1) Sorted 2) Unsorted.

Here we will look on the unsorted Linear search.

Linear search also known as sequential search, is a Process that checks every element in the list sequentially until the desired element is found. When the elements to be searched are not specifically arranged in ascending or descending order. They are arranged in random manner. That is what it calls unsorted Linear search.

Scanned with CamScanner

T8

Unsorted Linear search

- The data is entered in random manner.
- User needs to specify the elements to be searched in the entered list.
- Check the condition that whether the entered number matches if it matches then display the location plus increment 1 as data is stored from location zero.
- If all elements are checked one by one and element not found then prompt message number not found

Scanned with CamScanner

```
#Source Code
print("Name:Khushboo Yadav \nRoll No.:1777")
i = 0
arr = [4,6,8,12,15,19,21,61,75,89,99]
print(arr)
search = int(input("Enter No in Search :"))
if(search<arr[0] or search>arr[10]):
        print("Number does not exist")
for j in range(len(arr)):
        if(search == arr[j]):
        print("No. found at :",j)
        print("No. is :",arr[j])

        i += 1
        break
if(i == 1):
        print("No. not found")
```

```
#Output
Name:Khushboo Yadav
Roll No.:1777
[4, 6, 8, 12, 15, 19, 21, 61, 75, 89, 99]
Enter No in Search :100
Number does not exist
```

Scanned with CamScanner

Practical 2

Aim: To search a number from the list using linear sorted method.

Theory: Searching and Sorting are different modes or types of data structure.
Sorting - To basically sort the inputed data in ascending or descending order.
Searching - To search elements and to display in the same

In searching that too in Linear sorted search the data is arranged in ascending to descending or descending to ascending.
That is all what it meant by searching through 'sorted' that is well arranged data.

Scanned with CamScanner

Sorted Linear Search

- The user is supposed to enter data in sorted manner.
- User has to give an element for searching through sorted list.
- If element is found display with an updation as value is stored from location '0'.
- If data or element not found print the same.
- In sorted order list of elements we can check the condition that whether the entered number lies from starting point till the last element if not: then without any processing we can say number not in the list.

Scanned with CamScanner

```
#Source Code
print("Name:Khushboo Yadav \nRoll No.:1777")
i = 0
arr = [1,2,6,14,5,85,26,42,16,48,73,54,46]
print(arr)
search = int(input("Enter No in Search :"))
for j in range(len(arr)):
        if(search == arr[j]):
        print("No. found at :",j)
        print("No is :",arr[j])
        i += 1
        break
if(i != 1):
        print("No not found")
```

```
#Output:-
Name:Khushboo Yadav
Roll No.:1777
[1, 2, 6, 14, 5, 85, 26, 42, 16, 48, 73, 54, 46]
Enter No in Search :26
No. found at : 6
No is : 26
```

Scanned with CamScanner

Practical 3

Aim: To search a number from the given sorted list using binary search.

Theory: A binary search also known as a half-interval search, is an algorithm used in computer science to locate a specified value (key) within an array. For the search to be binary the array must be sorted in either ascending or descending order.

At each step of the algorithm a comparision is made and the procedure branches into one of two directions. Specifically, the key value is compared to the middle element of the array. If the key value is less than or greater than this middle element, the algorithm knows which half of the array to continue searching in because the array is sorted.

This process is repeated on progressively smaller segments of the array untile the value is located.

Because each step in algorithm divides the array size in half a binary search will complete successfully in logarithmic time.

Scanned with CamScanner

```
#Source Code
print("Name : Khushboo Yadav \nRoll No. : 1777")
A = [12,21,24,25,34,67,78,79,85,88,93,99]
print(A)
search = int(input("Enter the number to be searched : "))
l = 0
h = len(A)-1
m = (l+h)//2
if(search<A[l] or search>A[h]):
        print("Number does not exist !")
elif(search == A[l]):
        print("Number found at location : ",l)
elif(search == A[h]):
        print("Number found at location : ",h)
else:
        while(l != h):
        if search == A[m]:
        print("Number found at location : ",m)
        break
        else:
        if search < A[m]:
        h = m
        m = (l+h)//2
        else:
        l = m
        m = (l+h)//2
        if(l == h):
        print("Number found at location : ",l)
```

```
#Output
Name : Khushboo Yadav
Roll No. : 1777
[12, 21, 24, 25, 34, 67, 78, 79, 85, 88, 93, 99]
Enter the number to be searched : 25
Number found at location :  3

Name : Khushboo Yadav
Roll No. : 1777
[12, 21, 24, 25, 34, 67, 78, 79, 85, 88, 93, 99]
Enter the number to be searched : 10
Number does not exist !
```

Scanned with CamScanner

Practical 4

Aim: To demonstrate the use of stack.

Theory: In computer science, a stack is an abstract data type that scrolls as a collection of elements with two principal operations push, which will adds an element to the collection and pop, which removes the most recently added element that was not yet removed.
The order may be LIFO (Last in First out) or FILO (First in Last out).

- PUSH: Adds an item in the if the stack is full then it is said to be over flow condition.
- POP: Removes an item from the stack. The items are poped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an underflow condition.

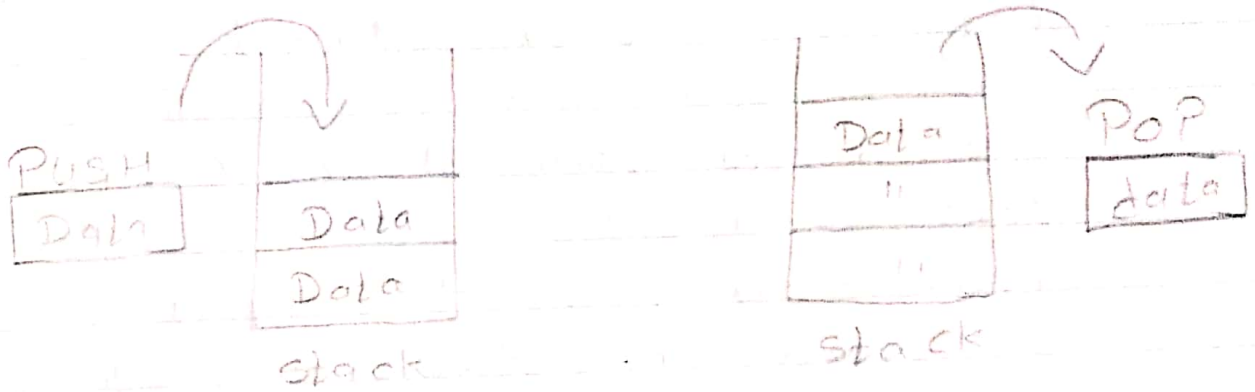
Scanned with CamScanner

```
#Stack
print("Name:Khushboo Yadav\nRoll No:1777")
class Stack():
        def __init__(self):
        self.l = [0,0,0,0,0,0,0]
        self.tos = -1
        def push(self,data):
        n = len(self.l)
        if self.tos == n-1:
        print("Stack is Full")
        else:
        self.tos += 1
        self.l[self.tos] = data
        def pop(self):
        if self.tos < 0:
        print("Stack is Empty")
        else:
        k = self.l[self.tos]
        print("data = ",k)
        self.tos -= 1
s = Stack()
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.push(50)
s.push(60)
s.push(70)
s.push(80)
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()

#Output:
Name:Khushboo Yadav
Roll No:1777
Stack is Full
data =  70
data =  60
data =  50
data =  40
data =  30
data =  20
data =  10
Stack is Empty
```

Scanned with CamScanner

- Peek or Top : Returns top element of stack.

- is Empty: Returns true if stack is empty else false.



Last in First out

4/8

Practical 5

Aim: To demonstrate Queue add and delete

Theory: Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.
Front points to the beginning of the Queue and Rear points to the end of the queue.
Queue follows the FIFO (First in First out) structure.
According to its FIFO structure, element inserted first will also be removed first.
In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue) because queue is open at bot of its ends.
Enqueue() can be termed as add() in queue i.e. adding a element in queue.
Dequeue() can be termed as delete or Remove i.e. deleting or removing of element

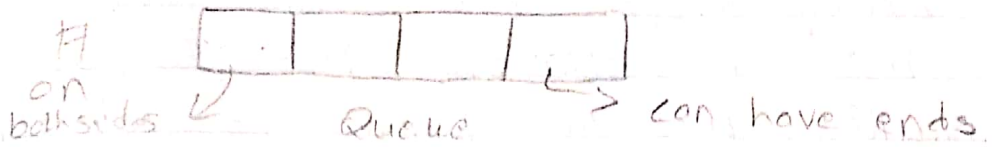
Scanned with CamScanner

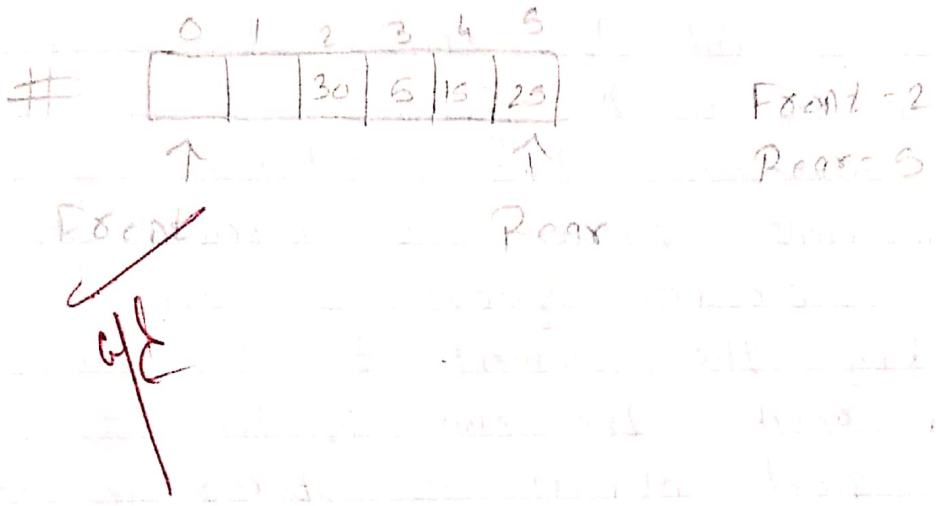
```
#Queue and Delete
print("Name:Khushboo Yadav \nRoll No.:1777")
class Queue:
        global r
        global f
        def __init__(self):
        self.r = 0
        self.f = 0
        self.l = [0,0,0,0,0,0]
        def add(self,data):
        n = len(self.l)
        if self.r < n-1:
        self.l[self.r] = data
        self.r += 1
        else:
        print("Queue is Full")
        def remove(self):
        n = len(self.l)
        if self.f < n-1:
        print(self.l[self.f])
        self.f += 1
        else:
        print("Queue is Empty")
q = Queue()
q.add(30)
q.add(40)
q.add(50)
q.add(60)
q.add(70)
q.add(80)
q.remove()
q.remove()
q.remove()
q.remove()
q.remove()
q.remove()

#Output:
Name:Khushboo Yadav
Roll No.:1777
Queue is Full
30
40
50
60
70
Queue is Empty
```

Scanned with CamScanner

Front end is used to get the front data item from a queue.

Rear is used to get the last item from a queue.





Practical 6

Aim: To demonstrate the use of circular queue in data structure.

Theory: The queue that we implement using an array suffer from one limitation. In that implementation there is a possibility that the queue is reported as full, even though in actuality there might be empty slots at the beginning of the queue.
To overcome this limitation we can implement queue as circular queue. In circular queue we go on adding the element to the queue and reach the end of the array. The next element is stored in the first slot of the array.

Example

Front=1 Rear=3

Scanned with CamScanner

```
#Circular Queue
print("Name:Khushboo Yadav \nRoll No.:1777")
class Queue:
        global r
        global f
        def __init__(self):
        self.r = 0
        self.f = 0
        self.l = [0,0,0,0,0,0]
        def add(self,data):
        n = len(self.l)
        if self.r <= n-1:
        self.l[self.r] = data
        print("data added : ",data)
        self.r += 1
        else:
        s = self.r
        self.r = 0
        if self.r < self.f:
        self.l[self.r] = data
        self.r += 1
        else:
        self.r = s
        print("Queue is Full")
        def remove(self):
        n = len(self.l)
        if self.f <= n-1:
        print("data removed : ",self.l[self.f])
        self.f += 1
        else:
        s = self.f
        self.f = 0
        if self.f < self.r:
        print(self.l[self.f])
        self.f += 1
        else:
        print("Queue is Empty")
        self.f = s
q = Queue()
q.add(44)
q.add(55)
q.add(66)
q.add(77)
q.add(88)
q.add(99)
q.remove()
q.add(66)

#Output:
```

Scanned with CamScanner

Name:Khushboo Yadav
Roll No.:1777
data added : 44
data added : 55
data added : 66
data added : 77
data added : 88
data added : 99
data removed : 44

Scanned with CamScanner

0	1	2	3	4	5
	BB	CC	DD	EE	FF

Front=1 Rear=5

0	1	2	3	4	5
		CC	DD	EE	FF

Front=2 Rear=5

0	1	2	3	4	5
xxx		CC	DD	EE	FF

Front=2 Rear=0

Scanned with CamScanner

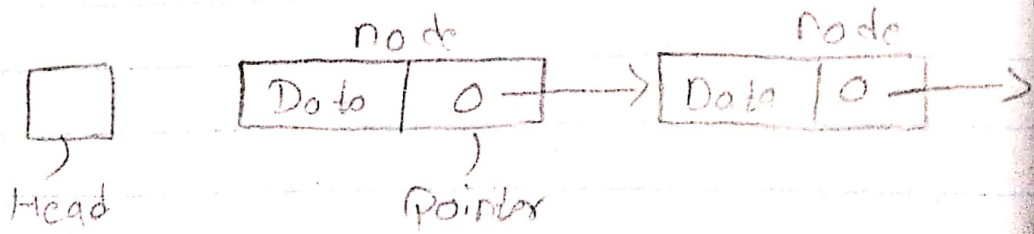
Practical 7

Aim: To demonstrate the use of Linked list in data structure

Theory: A linked list is a sequence of data structures. Linked list is a sequence of links which contains items. Each link contains a connection to another link.

- LINK: Each link of a linked list can store a data called an element.
- NEXT: Each link of a linked list contains a link to the next link called NEXT.
- LINKED LIST: A Linked list contains the connection link to the first link called First.

Linked List Representation:



Scanned with CamScanner

```
#LinkedList
print("Name:Khushboo Yadav \nRoll No.:1777")
class Node:
        global data
        global next
        def __init__(self,item):
        self.data = item
        self.next = None
class LinkedList:
        global s
        def __init__(self):
        self.s = None
        def addL(self,item):
        newnode = Node(item)
        if self.s == None:
        self.s = newnode
        else:
        head = self.s
        while head.next != None:
        head = head.next
        head.next = newnode
        def addB(self,item):
        newnode = Node(item)
        if self.s == None:
        self.s = newnode
        else:
        newnode.next = self.s
        self.s = newnode
        def display(self):
        head = self.s
        while head.next != None:
        print(head.data)
        head = head.next
        print(head.data)
start = LinkedList()
start.addL(50)
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
```

```
start.addB(30)
start.addB(20)
start.display()

#Output:
Name:Khushboo Yadav
Roll No.:1777
20
30
40
50
60
70
80
```

Scanned with CamScanner

Types of Linked List:

- simple
- Doubly
- Circular

Basic operations

- Insertion
- Deletion
- Display
- Search
- Delete.

Scanned with CamScanner

Practical 2

Aim: To evaluate postfix expression using stack.

Theory: Stack is an (ADT) and works on LIFO (Last in first out) i.e. PUSH and POP operations.
A postfix expression is a collection of operators and operands in which the operator is placed after the operands.

Steps to be followed:

1. Read all the symbols one by one from left to right in the given postfix expression.
2. If the reading symbol is operand then push it on to the stack.
3. If the reading symbol is operator (+, -, *, /, etc). then perform two pop operations and store the two popped operands in two different variables (operand 1 and operand 2) Then perform reading symbol operation using operand 1 and operand 2 and push result back on to the stack.
4. Finally! perform a pop operation and

```
#PostFix Evaluation
print("Name:Khushboo Yadav \nRoll No.:1777")
def evaluate(s):
        k = s.split()
        n = len(k)
        stack = []
        for i in range(n):
        if k[i].isdigit():
        stack.append(int(k[i]))
        elif k[i] == '+':
        a = stack.pop()
        b = stack.pop()
        stack.append(int(b)+int(a))
        elif k[i] == '-':
        a = stack.pop()
        b = stack.pop()
        stack.append(int(b)-int(a))
        elif k[i] == '*':
        a = stack.pop()
        b = stack.pop()
        stack.append(int(b)*int(a))
        else:
        a = stack.pop()
        b = stack.pop()
        stack.append(int(b)/int(a))
        return stack.pop()
s = "5 6 4 2 * - +"
r = evaluate(s)
print("The Evaluated Value is ",r)

#Output:
Name:Khushboo Yadav
Roll No.:1777
The Evaluated Value is  3
```

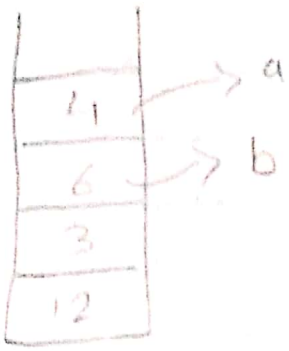
Scanned with CamScanner

display the popped value as final result.

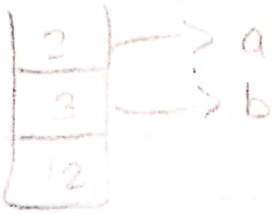
value of postfix expression:

S = 12 3 6 4 − + *

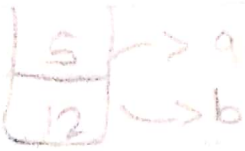
stack



b − a = 6 − 4 = 2 // store again in stack



b + a = 3 + 2 = 5 // store again in stack



b * a = 12 * 5 = 60

Practical 9

Aim: To sort given random data by using bubble sort.

Theory: Sorting is type in which any random data is sorted i.e. arranged in ascending or descending order.

Bubble sort sometimes referred to as sinking sort.

Is a simple sorting algorithm that repeatedly steps through the lists, compares adjacent elements and swap them if they are in wrong order.

The pass through the list is repeated until the list is sorted.

The Algorithm which is a comparision sort is named for the way smaller or larger elements "bubble" to the top of the list.

Although the Algorithm is simple, it is too slow and it compares one element checks if condition fails then only swaps otherwise goes on.

Scanned with CamScanner

```
#Bubble Sort
print("Name:Khushboo Yadav \nRoll No.:1777")
a = [7,3,8,9,5,6,2]
for p in range(len(a)-1):
        for c in range(len(a)-1-p):
        if (a[c]>a[c+1]):
        a[c],a[c+1] = a[c+1],a[c]
print(a)
```

Output:
Name:Khushboo Yadav
Roll No.:1777
[2, 3, 5, 6, 7, 8, 9]

Scanned with CamScanner

Example:

First Pass

(51428) → (15428). Here algorithm compare the first two elements and swaps since 5>1

(15428) → (14528) swaps since 5>4

(14528) → (14258) swap since 5>2

(14258) → (14259) Now since these elements are already in order (8>5) algorithm does not swap them.

Second Pass:

(14258) → (14258)

(14258) → (12458) swap sin 4>2

(12458) → (12458)

Third Pass

(12458) It checks and gives the data in sorted order.

Practical -10

Aim: To 'evaluate' i.e to sort the given data in Quick sort.

Theory: Quicksort is an efficient sorting algorithm. Type of a Divide and Conquer algorithm. It Picks an elements as Pivot and Partitions the given array around the picked Pivot. There are many different versions of quick sort that Pick Pivot in different ways.

1) Always Pick first element as Pivot.
2) Always Pick last element as Pivot.
3) Pick a random element as Pivot.
4) Pick median as Pivot.

The key Process in quicksort is, Partition(). Target of Partitions is, given as array and an element x of array as Pivot, Put x at its correct Position in sorted array and Put all smaller elements (smaller than x) before x and Put all greater elements (greater than x) after x. All this should be done in linear Time.

Scanned with CamScanner

```
#QuickSort
print("Name:Khushboo Yadav \nRoll No.:1777")

def quickSort(alist):
        quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):

        if first<last:
                splitpoint = partition(alist,first,last)
                quickSortHelper(alist,first,splitpoint-1)
                quickSortHelper(alist,splitpoint+1,last)

def partition(alist,first,last):
        pivotvalue = alist[first]
        leftmark = first + 1
        rightmark = last
        done = False
        while not done:
        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
                leftmark += 1
        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
                rightmark -= 1
        if rightmark < leftmark:
                done = True
        else:
                temp = alist[leftmark]
                alist[leftmark] = alist[rightmark]
                alist[rightmark] = temp

        temp = alist[first]
        alist[first] = alist[rightmark]
        alist[rightmark] = temp
        return rightmark

alist = [42,54,45,67,89,66,55,80,100]
quickSort(alist)
print(alist)
```

Output:
Name:Khushboo Yadav
Roll No.:1777
[42, 45, 54, 55, 66, 89, 67, 80, 100]

Scanned with CamScanner

Practical 11

Aim: selection sort

Theory:

In computer science, selection sort is an in place comparsion sorting algorithm. It has an O (n²) time complexity, which makes it inefficient on large lists and generally performs worst than the similar insertion sort.

Selection sort is noted for its simplicity and has performance advantages over more complicated algorithm in certain situations particularly where auxilary memory is limited.

Scanned with CamScanner

```
#Selection sort
print("Name: Khushboo Yadav \nRoll No.: 1777")
a = [12,10,13,16,19,11,20]
print(a)
for i in range(len(a)-1):
        for j in range(len(a)-1):
                if a[j] > a[j+1]:
                        a[j],a[j+1] = a[j+1],a[j]
print(a)
```

```
#Output:
Name: Khushboo Yadav
Roll No.: 1777
[12, 10, 13, 16, 19, 11, 20]
[10, 11, 12, 13, 16, 19, 20]
```

Scanned with CamScanner

Practical 12

Aim: Merge sort.

Theory: Merge sort is the algorithm which follows divide and conquer approach. Consider an array if of n no. of elements. The algorithm processes the element in 3 steps.

1) If it contains 0 or 1 elements then it is already sorted. otherwise, Divide it into two sub-array of equal no. of elements.
2) Conquer means sort the two sub-arrays recursively using the merge sort.
3) Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

The main idea behind merge sort is that, the short list takes less time to be sorted.

Complexity	Best case	Average case	Worst case
Time complexity	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Space complexity			$O(n)$

Scanned with CamScanner

```
#Merge Sort
print("Name:Khushboo Yadav \nRoll No.:1777")
def sort(arr,l,m,r):
        n1=m-l+1
        n2=r-m
        L=[0]*(n1)
        R=[0]*(n2)
        for i in range(0,n1):
                L[i]=arr[l+i]
        for j in range(0,n2):
                R[j]=arr[m+1+j]
        i=0
        j=0
        k=l
        while i<n1 and j<n2:
                if L[i]<=R[j]:
                        arr[k]=L[i]
                        i+=l
                        j+=l
                        k+=l
                else:
                        arr[k]=R[j]
                        j+=1
                        k+=1
                while i<n1:
                        arr[k]=L[i]
                        i+=1
                        k+=1
                while j<n2:
                        arr[k]=R[j]
                        j+=1
                        k+=1
def mergesort(arr,l,r):
        if l<r:
                m=int((l+(r-1))/2)
                mergesort(arr,l,m)
                mergesort(arr,m+1,r)
                sort(arr,l,m,r)
arr=[12,23,34,56,78,45,86,98,42]
n=len(arr)
mergesort(arr,0,n-1)
print(arr)
```

Output:

Name:Khushboo Yadav

Roll No.:1777

[12, 23, 34, 56, 42, 45, 78, 86, 98]

Scanned with CamScanner

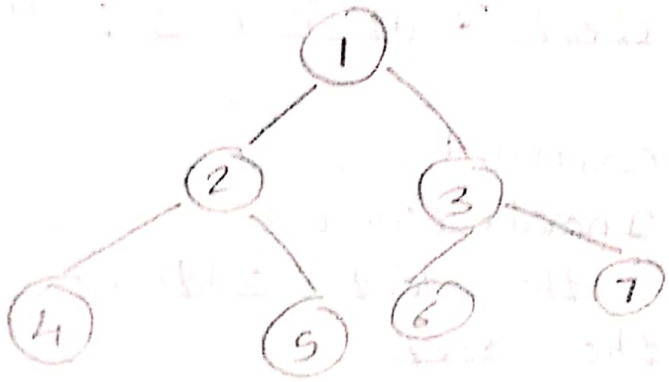
Practical 13

Aim: Binary Tree and Traversals.

Theory:

* Binary Tree:

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary can have only 2 children. We typically name them the left and right child.

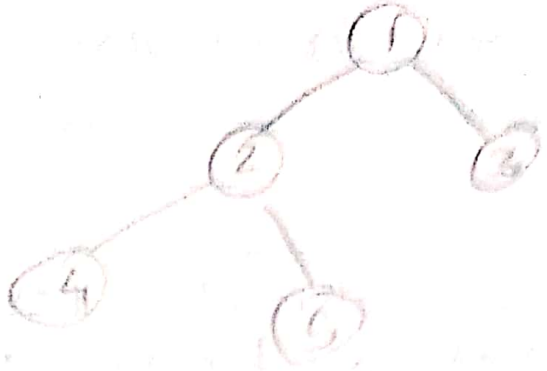


A Binary Tree node contains following parts:

1) Data
2) Pointer to left child
3) Pointer to right child.

* Binary Tree Traversals:

Unlike linear data structures (Array, Linked List, Queues, stacks, etc) which have only one logical way to tranverse them, trees can be traversed in different way. Following are the generally used ways for tranverse trees



Traversals:

a] Inorder (Left, Root, Right): 4 2 5 1 3

b] Preorder (Root, Left, Right): 1 2 4 5 3

c] Postorder (Left, Right, Root): 4 5 2 3 1

* Inorder Traversal:

Algorithm Inorder (Tree)

1) Traverse the left subtree
2) Visit the root
3) Traverse the right subtree

* Preorder Traversal:

Algorithm Preorder (Tree)

1) Visit the root
2) Traverse the left subtree
3) Traverse the right subtree

* Postorder Traversal:

Algorithm Postorder (Tree)

1) Traverse the left subtree
2) Traverse the right subtree
3) Visit the root

Scanned with CamScanner

```
#Binary Tree and Traversal
print("Name:Khushboo Yadav \nRoll No.:1777")
class Node:
        global r
        global l
        global data
        def __init__(self,l):
                self.l=None
                self.data=l
                self.r=None
class Tree:
        global root
        def __init__(self):
                self.root=None
        def add(self,val):
                if self.root==None:
                        self.root=Node(val)
                else:
                        newnode=Node(val)
                        h=self.root
                        while True:
                                if newnode.data < h.data:
                                        if h.l!=None:
                                                h=h.l
                                        else:
```

Scanned with CamScanner

```
                    h.l=newnode
                    print(newnode.data,"added on left of",h.data)
                    break
            else:
                if h.r!=None:
                    h=h.r
                else:
                    h.r=newnode
                    print(newnode.data,"added on right of",h.data)
                    break
    def preorder(self,start):
        if start!=None:
            print(start.data)
            self.preorder(start.l)
            self.preorder(start.r)
    def inorder(self,start):
        if start!=None:
            self.inorder(start.l)
            print(start.data)
            self.inorder(start.r)
    def postorder(self,start):
        if start!=None:
            self.inorder(start.l)
            self.inorder(start.r)
            print(start.data)
T=Tree()
T.add(100)
T.add(80)
T.add(70)
T.add(85)
T.add(10)
T.add(78)
T.add(60)
T.add(88)
T.add(15)
T.add(12)
print("Preorder")
T.preorder(T.root)
print("Inorder")
T.inorder(T.root)
print("Postorder")
T.postorder(T.root)
```

Scanned with CamScanner

Output:
Name:Khushboo Yadav
Roll No.:1777
80 added on left of 100
70 added on left of 80
85 added on right of 80
10 added on left of 70
78 added on right of 70
60 added on right of 10
88 added on right of 85
15 added on left of 60
12 added on left of 15
Preorder
100
80
70
10
60
15
12
78
85
88
Inorder
10
12
15
60
70
78
80
85
88
100
Postorder
10
12
15
60
70
78
80
85
88
100

Scanned with CamScanner