☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
1	To search a number from the list using linear unsorted →	41-42	21-12-19	
2	To search a number from the list using linear sorted method →	43-44	21-12-19	
3	To search a number from the given sorted list using binary search →	45	21-12-19	
☆	Impetus (Issue Date) → ←		03-1-20	
4	To demonstrate the use of stack →	46	18-1-20	
5	To demonstrate Queue add and delete →	47-48	18-1-20	
6	To demonstrate the use of circular queue in data-structure	49-50	18-1-20	

☆☆ INDEX ☆☆

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 -

1) LINEAR SEARCH
2) 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 elements 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.

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.

Source code:

```
print("Khushi Singh \n 1768")
a=[3,33,4,45,23,79,38]
j=0
print(a)
search=int(input("Enter no to be searched: "))
for i in range(len(a)):
    if(search==a[i]):
        print("Number found at: ",i+1)
        j=1
        break
if(j==0):
    print("Number NOT FOUND!")
```

Output:

Case 1:

```
Khushi Singh
 1768
[3, 33, 4, 45, 23, 79, 38]
Enter no to be searched: 33
Number found at:  2
```

Case 2:

```
Khushi Singh
 1768
[3, 33, 4, 45, 23, 79, 38]
Enter no to be searched: 67
Number NOT FOUND!
```

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 manner.

SEARCHING → To search elements and to display 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.

SORTED LINEAR SEARCH :

⇒ The user is supposed to enter data in sorted manner.

⇒ User has to given an element for searching through sorted list.

⇒ If element is found display with an updation

as value is sorted 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 the without any processing we can say number not in the list.

Source code:

```
print("Khushi Singh \n1768")
a=[3,10,21,59,69,74,80]
j=0
print(a)
search=int(input("Enter the number to be searched: "))
if((search<a[0]) or (search>a[6])):
  print("Number doesnt exist!")
else:
  for i in range (len(a)):
    if(search==a[i]):
      print("Number found at: ",i+1)
      j=1
      break
  if(j==0):
    print("Number NOT found!")
```

Output:

Case 1:

```
Khushi Singh
1768
[3, 10, 21, 59, 69, 74, 80]
Enter the number to be searched: 80
Number found at:  7
```

Case 2:

```
Khushi Singh
1768
[3, 10, 21, 59, 69, 74, 80]
Enter the number to be searched: 1
Number doesnt exist!
```

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 comparison 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 until the value is located. Because each step in the algorithm divides the array size is half a binary search will complete successfully in logarithmic time.

PRACTICAL-3

SOURCE CODE:

```
a=[3,4,21,24,25,29]
print("Khushi Singh \n 1768")
search=int(input("Enter the number to be searched: "))
l=0
h=len(a)-1
m=int((l+h)/2)
if((search<a[l])or(search>a[h])):
   print("Does not exist")
else:
   while(l!=h):
      if(search==a[m]):
         print("Number is found at: ",m)
         break
      else:
         if(search<a[m]):
            h=m-1
            m=int((l+h)/2)
         else:
            l=m+1
            m=int((l+h)/2)
   if(search==a[l]):
      print("Number is found at location: ",l)
   else:
      print("Number not found")
```

OUTPUT:

CASE1:

```
Khushi Singh
 1768
Enter the number to be searched: 4
Number is found at location:  1
```

CASE2:

```
Khushi Singh
 1768
Enter the number to be searched: 28
Number not found
```

CASE3:

```
Khushi Singh
 1768
Enter the number to be searched: 33
Does not exist
```

PRACTICAL - 4

AIM : To demonstrate the use of stack.

THEORY :

In computer science, a stack is an abstract data type that serves as a collection of elements with two principal operations push, which 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).

There Basic Operations are performed in the stack.

⇒ PUSH : Adds an item in the stack. If the stack is full then it is said to be over flow condition.

⇒ POP : Removes an item from the stack. The items are popped in the reversed order in which they are present. If the stack is empty, then it is said to be underflow condition.

⇒ PEEK OR TOP : Returns top elements of stack.

⇒ EMPTY : Returns true if stack is empty else false.

SOURCE CODE:

```
print("Khushi Singh \n 1768")
class stack:
  global tos
  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=self.tos+1
      self.l[self.tos]=data
  def pop(self):
    if self.tos<0:
      print("stack empty")
    else:
      k=self.l[self.tos]
      print("data=",k)
      self.tos=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:

```
Khushi Singh
1768
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
```

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 both of its ends.

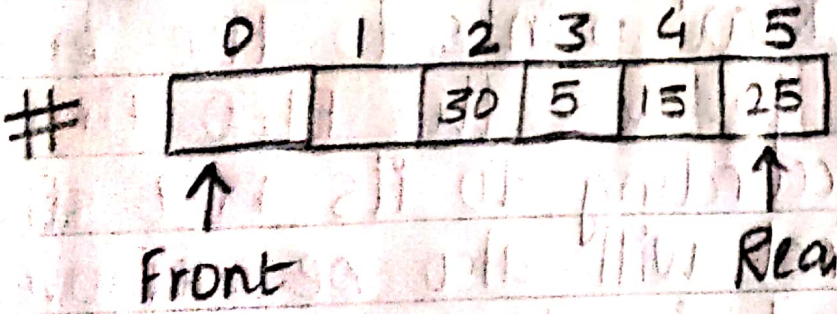
ENQUEUE : Enqueue() can be termed as add() in queue i.e adding a element in queue.

DEQUEUE : Dequeue() can be termed as delete or remove. i.e deleting or removing of element.

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

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

Queue can have ends on both sides



Front = 2
Rear = 5

SOURCE CODE:

```
Print("Khushi Singh \n 1768")

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=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=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:

```
Khushi Singh
1768
Queue is full
30
40
50
60
70
Queue is empty
```

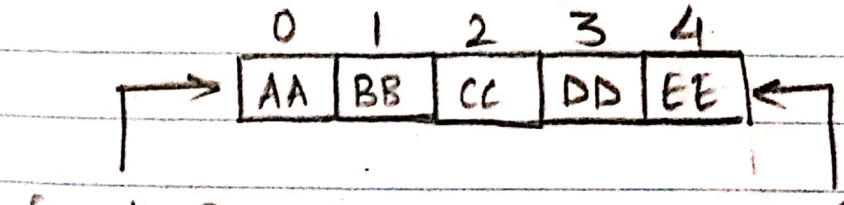
PRACTICAL - 4

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

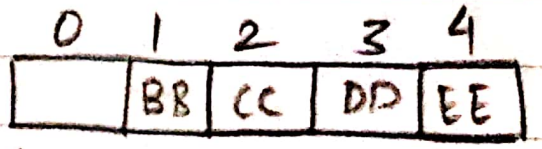
THEORY :

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 is or 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~~.

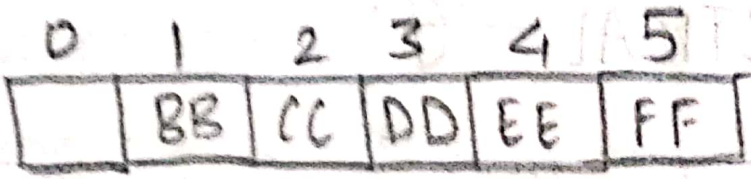
EXAMPLE :



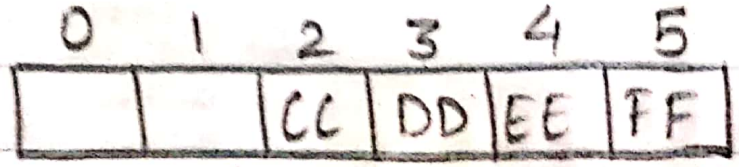
Front = 0 Rear = 4



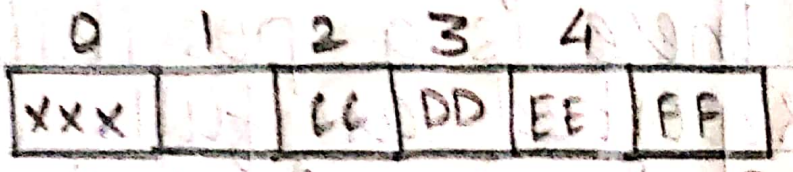
Front = 1 Rear = 4



Front = 1 Real = 5



Front = 2 Real = 5



Front = 2 Real = 0

SOURCE CODE:

```
Print("Khushi Singh\n 1768")

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=self.r+1
     else:
      s=self.r
      self.r=0
      if self.r<self.f:
        self.l[self.r]=data
        self.r=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=self.f+1
    else:
      s=self.f
      self.f=0
      if self.f<self.r:
        print(self.l[self.f])
        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 :

```
Khushi Singh

1768

data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
data removed: 44
```

PRACTICAL – 7

AIM : To demonstrate the use of Linked list in data structure.

THEORY :

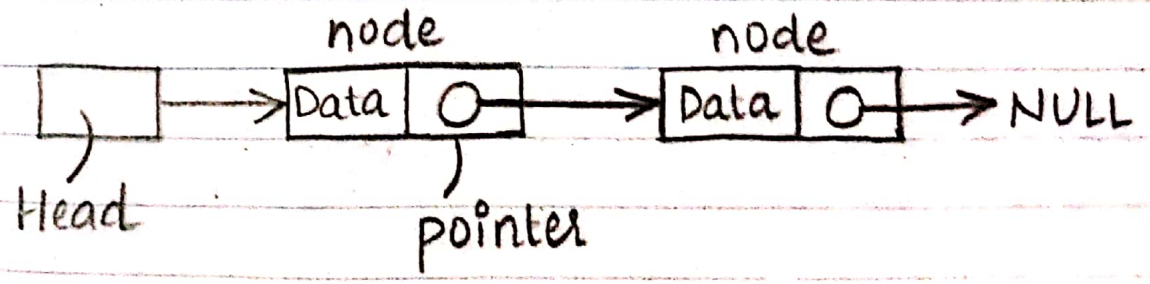
A linked list is a sequence of data structures. Linked list is a sequence of links which contains items. Each link contains a connection to another link.

* 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 :

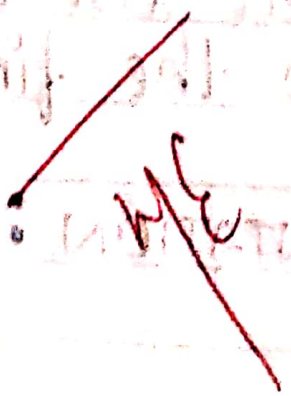


— TYPES OF LINKED LIST :

1) Simple
2) Doubly
3) Circular.

— BASIC OPERATIONS :

1) Insertion
2) Deletion
3) Display
4) Search
5) Delete.



```
print("KHUSHI SINGH 1768")
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:

```
KHUSHI SINGH 1768
20
30
40
50
60
70
80
```

PRACTICAL – 8

AIM : To evaluate postfix expression using stack.

THEORY :

Stack is an (ADT) and works on LIFO (Last-in First-one) i.e – PUSH and POP operations. A postfix expression is a collection of operations 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.

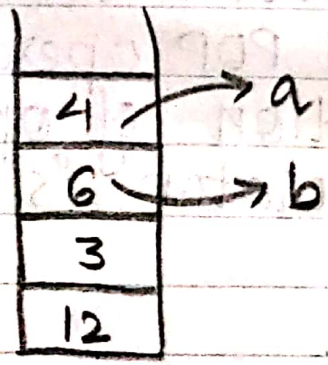
3) If the reading symbol is operator (+, −, *, /, etc) then perform TWO pop operations and store the two popped operands in two different variables (operand 1 and operand 2). Then perform reading symbol operation using operand 1 and operand 2 and push result back on to the stack.

4) Finally ! Perform a pop operation and display the popped value as final result.

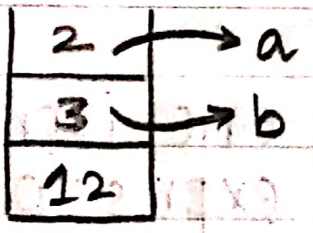
- Value of postfix expression :

S = 12 3 6 4 − + *

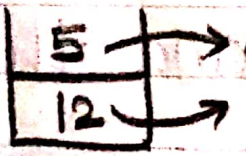
Stack :



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



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



b * a = 12 * 5 = 60

```
print("KHUSHI SINGH 1768")
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="23 9 8 5 - + * "
r=evaluate(s)
print("The evaluated value is:",r)
```

OUTPUT:

```
KHUSHI SINGH 1768
The evaluated value is: 276
```

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 swaps 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 as it compares one element checks if condition fails then only swaps otherwise goes on.

EXAMPLE:

First pass

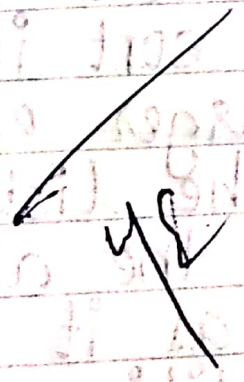
(5 1 4 2 8) → (1 5 4 2 8)

Here algorithm compares the first two elements

and swaps 5>1
(1 5 4 2 8) → (1 4 5 2 8) swap since 5>4
(1 4 5 2 8) → (1 4 2 5 8) swap since 5>2
(1 4 2 5 8) → (1 4 2 5 8) Now since these elements are already in order (8>5) algori-thm does not swap them.

Second pass :
(1 4 2 5 8) → (1 4 2 5 8)
(1 4 2 5 8) → (1 2 4 5 8) swap since 4>2
(1 2 4 5 8) → (1 2 4 5 8)

Third pass :
(1 2 4 5 8). It checks and gives the data in sorted order.



SOURCE CODE:

```
a=[72,24,67,38,11,99]
print("Khushi Singh \n 1768")
print("Array before sorting:\n",a)
for p in range (len(a)-1):
    for c in range (len(a)-1-p):
        if(a[c]>a[c+1]):
            t=a[c]
            a[c]=a[c+1]
            a[c+1]=t
print("Array after sorting: \n",a)
```

OUTPUT:

```
Khushi Singh
1768
Array before sorting:
[72, 24, 67, 38, 11, 99]
Array after sorting:
[11, 24, 38, 67, 72, 99]
```

PRACTICAL – 10

AIM : To sort given random data by using Selection sort.

THEORY :

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part unsorted part and putting it at the beginning in a given array. The algorithm maintains two arrays i.e- subarrays in a given array -

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

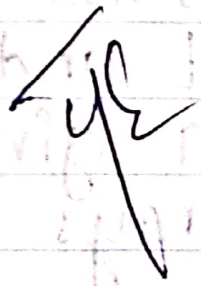
⇒ TIME COMPLEXITY :

$O(n^2)$ as there are two nested loops.

⇒ AUXILIARY SPACE : $O(1)$

The good thing about selection sort is it never makes more than $O(n)$ swaps and

can be useful when memory writes is a costly operation.



```
print( "Khushi")
a=[10,11,12,16,14,15]
print(a)
for i in range (len(a)-1):
    for j in range (len(a)-1):
        if (a[j]>a[i+1]):
            t=a[j]
            a[j]=a[i+1]
            a[i+1]=t  print(a)
```

Output

Khushi

[10,11,12,16,14,15]

[10,11,12,14,15,16]

PRACTICAL – 11

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

THEORY :

Quicksort is an efficient sorting algo-rithm. Type of a divide and conquer algo-rithm. It picks an element 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 an 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

SOURCE CODE:

```
print( "Khushi\n1768")
def qsort(alist):
   qsorthelper(alist,0,len(alist)-1)
def qsorthelper(alist,first,last):
   if first<last:
      splitpoint=partition(alist,first,last)
      qsorthelper(alist,first,splitpoint-1)
      qsorthelper(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=leftmark+1
      while alist[rightmark]>=pivotvalue and rightmark>=leftmark:
         rightmark=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]
qsort(alist)
print(alist)
```

OUTPUT:

Khushi

1768

[42, 45, 54, 55, 66, 67, 80, 89, 100]

PRACTICAL – 12

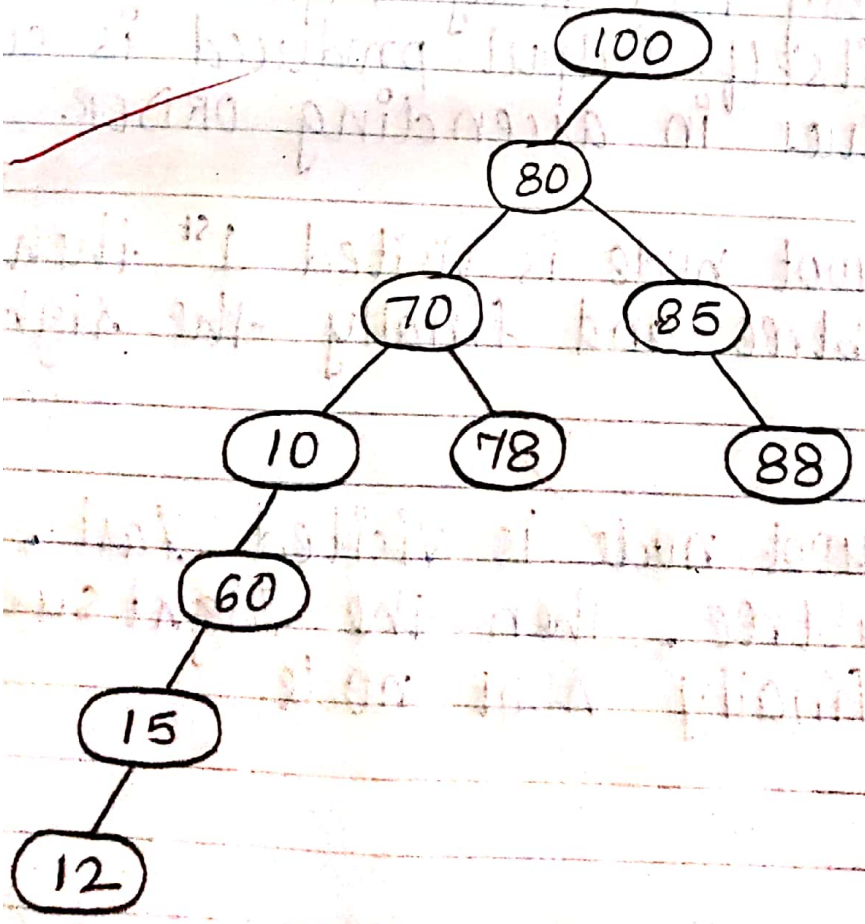
AIM : Binary Tree and Transversal.

THEORY :

A binary tree is a special type of tree in which every node or vertex has either no child or one child node or two child nodes.

A binary tree is an important class of a tree data structure in which a node can have at most two children.

Diagrammatic Representation of BINARY SEARCH TREE :



Transversal is a process to visit all the no of a tree and may print their values too.

There are 3 ways which we use to transverse a tree -

a) INORDER
b) PREORDER
c) POSTORDER

IN-ORDER : The left-subtree is visited 1st then the root and later the right sub tree. we should always remember that every node may represent a subtree itself. Output produced is sorte key values in ascending ORDER.

PRE-ORDER : The root node is visited 1st then th left subtree and finally the right subtree.

POST-ORDER : The root node is visited last, left subtree, then the right subtree and finally root node.

```
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:
                        h.l=newnode
                        print(newnode.data,"added
                        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)
```

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
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
```

PRACTICAL - 13

AIM : Merge Sort.

THEORY :

Merge Sort is a sorting technique based on divide and conquer techniques with worst - case time complexity being O(n log n), it one of the most respected algorithm.

Merge Sort first divides the array into equal halves and then combines them in a sorted manner.

It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging 2 halves. The merge (arr, l, m, r) is key process that assumes that arr [l....m] and arr [m+1...r] are sorted and merges the two sorted sub - array into one.

```
#Merge Sort
print("Name:Khushi  \nRoll No.:1768")
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:Khushi
Roll No.:1768
[12, 23, 34, 56, 42, 45, 78, 86, 98]
```