THAKUR COLLEGE OF SCIENCE & COMMERCE

NAAC Accredited with Grade "A" (3rd Cycle)

THAKUR TRUSTS

ISO 9001 : 2015 Certified

Degree College

Computer Journal

CERTIFICATE

SEMESTER II **UID No.** _____

Class FY. BSC CS **Roll No.** 1753 **Year** 2015-20

This is to certify that the work entered in this journal is the work of Mst. / Ms. MISHRA ANUSHKA ABHAY who has worked for the year 2019-20 in the Computer Laboratory.

Teacher In-Charge

Head of Department

Date : _____

Examiner

INDEX

Source code:

```
print("Anushka Mishra \n 1753")
a=[4,29,21,64,25,24,3]
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:

```
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 ==========
>>>
Anushka Mishra
 1753
[4, 29, 21, 64, 25, 24, 3]
Enter no to be searched: 21
Number found at:  3
>>> ========== RESTART ==========
>>>
Anushka Mishra
 1753
[4, 29, 21, 64, 25, 24, 3]
Enter no to be searched: 6
Number not found!
>>>
```

Practical No. 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

Unsorted Linear search

→ The data is entered in random manner.
→ User needs to specify the ~~random~~ 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("Anushka Mishra \n1753")
a=[3,4,21,24,25,29,64]
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:

```
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 (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Anushka Mishra
1753
[3, 4, 21, 24, 25, 29, 64]
Enter the number to be searched: 3
Number found at:  1
>>> ========== RESTART ==========
>>>
Anushka Mishra
1753
[3, 4, 21, 24, 25, 29, 64]
Enter the number to be searched: 65
Number doesnt exist!
>>> ========== RESTART ==========
>>>
Anushka Mishra
1753
[3, 4, 21, 24, 25, 29, 64]
Enter the number to be searched: 32
Number NOT found
>>>
```

Practical No. 2

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

Theory : Searching and sorting are different modes or types of data - structures 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 search through 'sorted' that is well arranged data.

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 fand display with an updation as value is stored from location 'o'.

→ 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.

Source code:

```
print("ANUSHKA MISHRA\n1753")
a=[3,4,21,24,25,29,64]
print(a)
search=int(input("ENter number to be searched from the list:"))
l=0
h=len(a)-1
m=int((l+h)/2)
if((search<a[l]) or (search>a[h])):
  print("Number not in RANGE!")
elif(search==a[h]):
  print("number found at location :",h+1)
elif(search==a[l]):
  print("number found at location:",l+1)
else:
  while(l!=h):
    if(search==a[m]):
      print("Number found at location:",m+1)
      break
    else:
      if(search<a[m]):
        h=m
        m=int((l+h)/2)
      else:
        l=m
        m=int((l+h)/2)
  if(search!=a[m]):
    print("Number not in given list!")
    break
```

Output:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (I
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
ANUSHKA MISHRA
1753
[3, 4, 21, 24, 25, 29, 64]
ENter number to be searched from the list:25
Number found at location: 5
>>> ========== RESTART ==========
>>>
ANUSHKA MISHRA
1753
[3, 4, 21, 24, 25, 29, 64]
ENter number to be searched from the list:65
Number not in RANGE!
>>> ========== RESTART ==========
>>>
ANUSHKA MISHRA
1753
[3, 4, 21, 24, 25, 29, 64]
ENter number to be searched from the list:10
Number not in given list!
>>>
```

Practical No. 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 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 grater than this middle element, the algorithm knows which half of array to continue searching in because the array is sorted.

This process is repeated on progressing smaller segments of array until the value is located.

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

Practical : 4

Aim: To demonstrate the use of stack

Theory:

In computer science, a stack is a abstract data type that serves as a collection of elements with two principal operation push, which adds an elements to the collection & pop, which removes the most recently added element that was not get removes. The order may be LIFO (Last in first out) or FILO (First in Last out)

The Basic operation 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 item are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an underflow condition.

- Peer or Top: Returns top elements of stack.
- is Empty: Returns true if stack is empty else false

Source code:

```
print("ANUSHKA MISHRA\n1753")
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=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=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:

```
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 ==========
>>>
ANUSHKA MISHRA
1753
Stack is full!
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
Stack is empty!
```

Sorce code:

```
print("ANUSHKA MISHRA\n1753")
class queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[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()
```

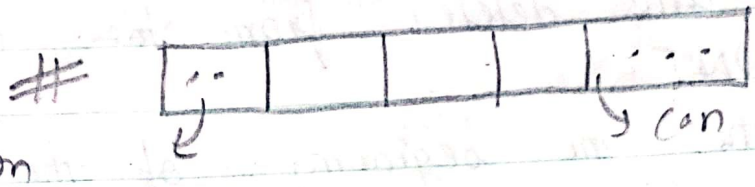
Practical No. 65

Aim : To demonstrate Queue add and delete.

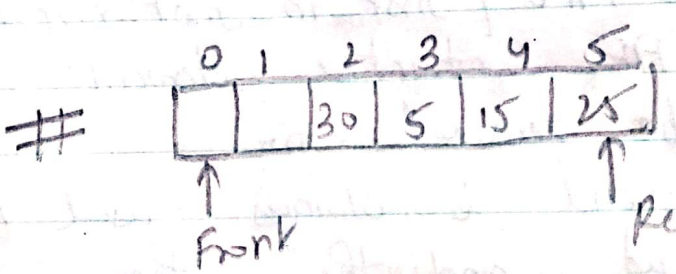
Theory : Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.
Front points to the beginning of the queue and Rear points to the end of the queue.
Queue follows the FIFO (First_in_First_out) structure.
According to 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() can be termed as add() in queue i.e. adding an element in queue.
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.



can have endy on both sides



Front = 2
Rear = 5

```
q.remove()
q.remove()
q.remove()
q.remove()
q.remove()
```

Output:

```
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 ==========
>>>
ANUSHKA MISHRA
1753
Queue is full!
Queue is full!
30
40
50
60
Queue is Empty!
Queue is Empty!
>>>
```

Sorce code:

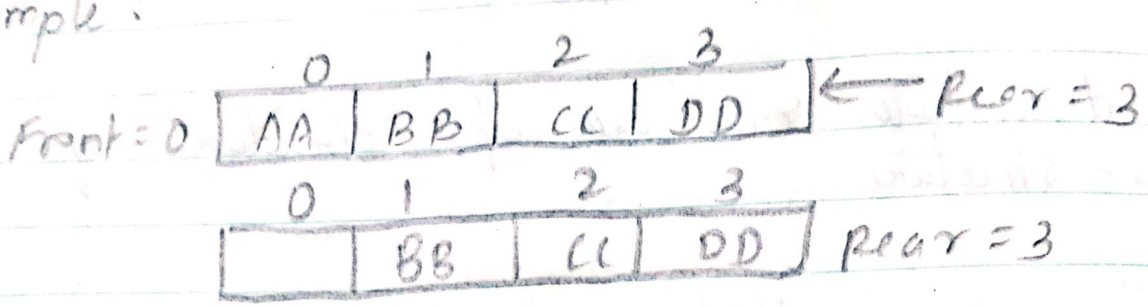
```
print("ANUSHKA MISHRA\n 1753")
class queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if(self.r<=n-1):
      self.l[self.r]=data
      print("data added:",data)
      self.r=self.r+1
    else:
      s=self.r
      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:
      print("Queue is empty")
```

Practical No.: 6

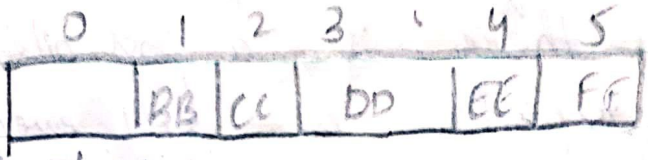
Aim : To demonstrate the use of circular queue is data-structure.

Theory : The queue that we implement using an array suffer from one limitations. 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.

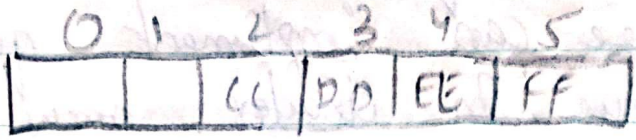
Example:



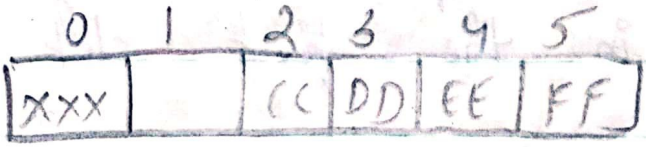
Front = 1



Front = 1



Front = 2



front = 2

```
        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:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
ANUSHKA MISHRA
 1753
data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
Data removed: 44
>>>
```

SOURCE CODE:

```
print("ANUSHKA MISHRA 1753")
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:
```

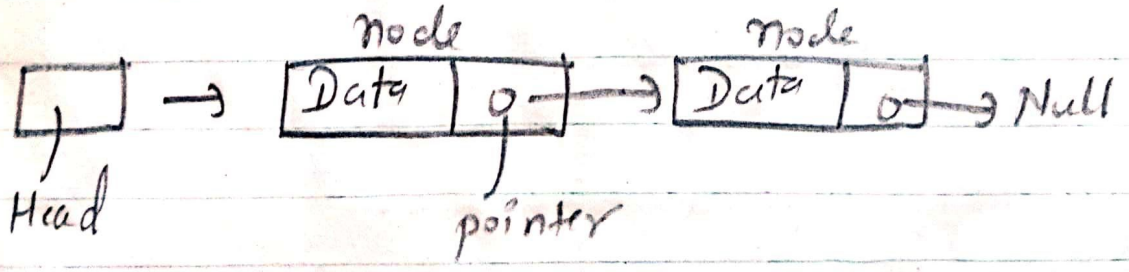
Practical No. 7.

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 :

→ simple
→ doubly
→ circular

Basic operations.

→ Insertion
→ Deletion.
→ Display
→ Search
→ Delete.

```
        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:

```
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 ==========
>>>
ANUSHKA MISHRA 1753
20
30
30
40
40
50
50
60
60
70
70
80
>>> ========== RESTART ==========
>>>
```

SOURCE CODE:

```
print("ANUSHKA MISHRA 1753")
def evaluate(s):
  k=s.split()
  n=len(k)
  stack=[]
  for i in range(n):
    if k[i].isdigit():
      print(int(k[i]))
      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="8 6 9 - +"
r=evaluate(s)
print("The evaluated value is :",r)
```

OUTPUT:

Practical No. 8.

Aim: To evaluate postfix expression using stack

Theory: Stack is an (ADT) & works on LIFO (Last-in First-out) i.e. PUSH & 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:

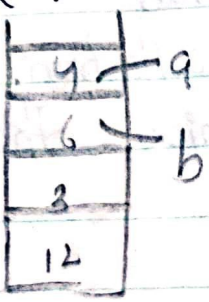
1. Read all the symbols one by one from left to right in the given postfix expression
2. If the reading symbol is operand then push it on to the stack
3. If the reading symbol is operator (+, -, *, /, etc) then perform two pop operations & store the two popped operands on two different variables (operand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

TAO

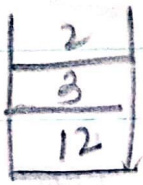
Value of postfix expression:

5 = 1 2 3 6 4 − + *

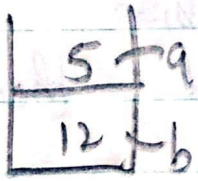
stack:



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



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



b * a − 12 * 5 = 50

```
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 ==========
>>>
>>>
ANUSHKA MISHRA 1753
8
6
9
The evaluated value is : 5
>>>
```

SOURCE CODE:

```
print("ANUSHKA MISHRA \n1753")
a=[10,9,8,7,1]
print("Before BUBBLE SORT elements list: \n",a)
for passes in range(len(a)-1):
  for compare in range(len(a)-1-passes):
    if(a[compare]>a[compare+1]):
      temp=a[compare]
      a[compare]=a[compare+1]
      a[compare+1]=temp
print("After BUBBLE SORT elements list: \n",a)
```

OUTPUT:

```
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 ==========
>>>
ANUSHKA MISHRA
1753
Before BUBBLE SORT elements list:
 [10, 9, 8, 7, 1]
After BUBBLE SORT elements list:
 [1, 7, 8, 9, 10]
>>>
```

PRACTICAL No : 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.

CDO

Example:

First pass
(5 1 4 2 8) → (1 5 4 2 8) Here algorithm compare the first two elements & swaps since 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) swap, Now since these elements are already in order (8>5) algorithm 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 & gives the data in sorted order.

```
# Quick Sort Practical
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=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=[49,54,45,99,89,64,55,82,100]
print("Unsorted: ",alist)
quickSort(alist)
print("Quicksort: ",alist)
print("\nAnushka\nFYBSC_CS 1753")
```

PRACTICAL No. 10.

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

Theory: Quicksort is an efficient sorting algorithm. Type of a Divide & conquer algorithm. It picks an elements as pivot & partition 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 elements as pivot.
2) Always pick last elements as pivot.
3) Pick a random element as pivot
4) Pick median as pivot.

The key process in quicksort is partition(). Target of partition 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, & put all greater elements (greater than x) after x. All this should be done in linear time.

PRACTICAL No. 11

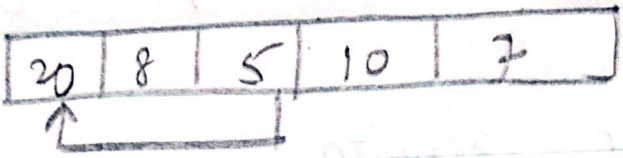
Aim : To sort given random data by using selection sort.

Theory : The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part & putting it at the beginning. The algorithm maintains two sub arrays in a given array. The subarray which is already sorted.

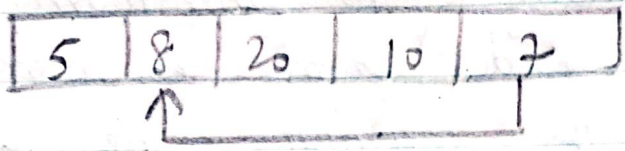
The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the smallest value as it makes a pass and after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the smallest item is in the correct place.

After the second pass, the next smallest is in place. This process continues and requires $n-1$ passes to sort n items, since the final item must be place after the $(n-1)$st pass.

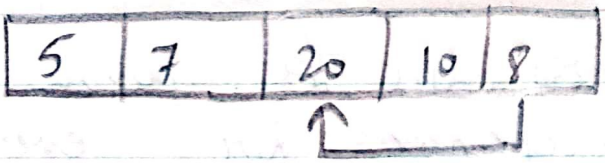
eg. On each pass the smallest remaining item is selected and then placed in its proper location.



5 is smallest

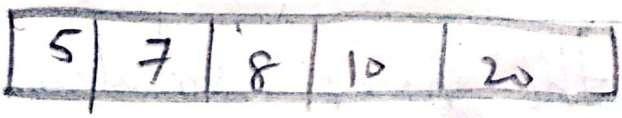


7 is smallest



8 is smallest

10 OK list is sorted



20 OK list is sorted

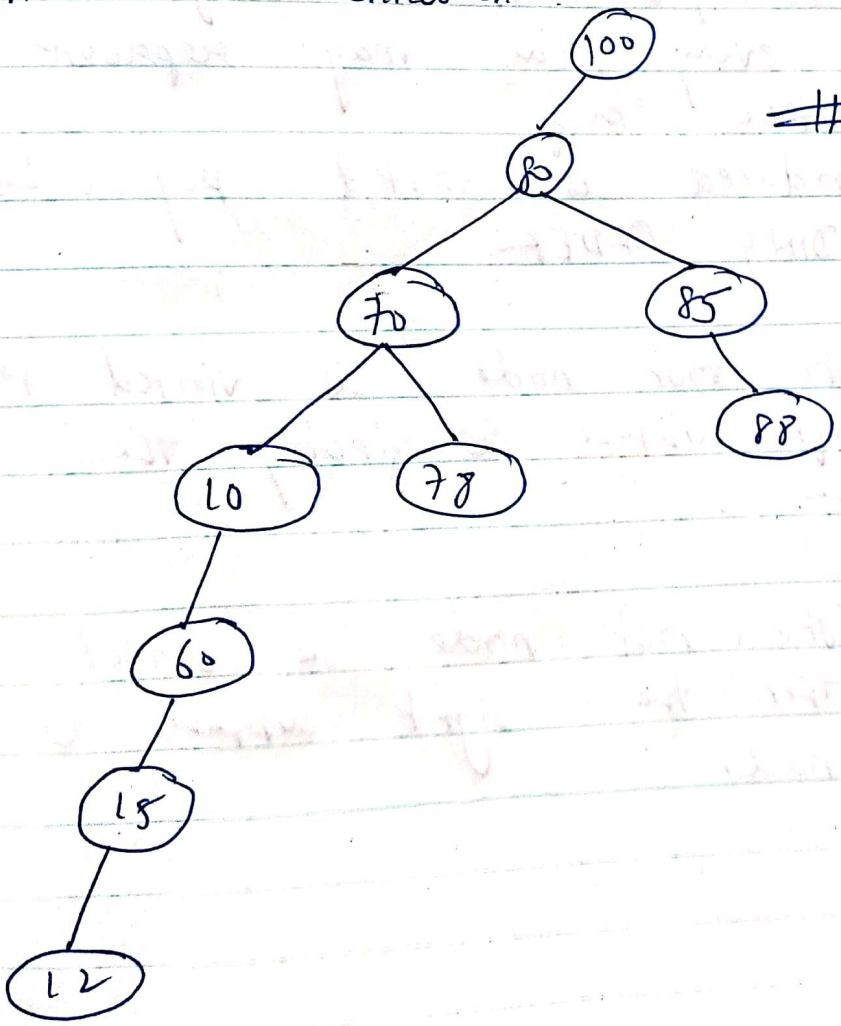
```
class Node:
    global r
    global l
    global data
    def __init__(self,l):
        self.l=None
        self.data=l
        self.r=None
class Tree:
    global root
    def __init__(self):
        self.root=None
    def add(self,val):
        if self.root==None:
            self.root=Node(val)
        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 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.postorder(start.l)
            self.postorder(start.r)
            print(start.data)
T=Tree()
T.add(100)
T.add(99)
T.add(75)
T.add(82)
T.add(9)
T.add(78)
T.add(63)
```

PRACTICAL No. 12.

Aim Binary Tree and Traversal

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.

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



Diagrometic representation of Binary search tree

Traversal: Traversal is a process to visit all the node of a tree and may print their value too.

There are 3 ways we use to traverse a tree.

(1) In-ORDER: The left-subtree is visited 1st then the root & later the right sub tree, we should always remember that every node may represent a subtree itself.
Output produced is sorted key values in ASCENDING ORDER

(2) PRE-ORDER: The root node is visited 1st then the left subtree & finally the right subtree.

(3) POST-ORDER: The root node is visited last, left subtree, then the right subtree & finally root node.

```
T.add(80)
T.add(13)
T.add(18)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
print("Anushka Mishra\nFY1753")
```

```
#Merge Sort
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+=1
            #k+=1
        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=[14,22,33,53,79,45,86,99,45]
print(arr)
n=len(arr)
mergesort(arr,0,n-1)
print(arr)
print("Anushka Mishra \nFY1753")
```

PRACTICAL NO. 13

AIM : MERGE SORT

Theory : Merge sort is a sorting technique based on divide & conquer technique with worst-case time complexity being O (n log n), is 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 & 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