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.
Searching can be carried out in two types Linear Search and Binary search.
Linear search is further classified as : Sorted and Unsorted.
In unsorted linear search the element to be found is arranged in random manner.
Unsorted Linear Search is carried out by following manner:

↳ The data is entered in random manner.

↳ User need to specify the element 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.

```
found = False
x=[27,32,44,12,6]
search=int(input("Enter a no. to be stored"))
for i in range (len(x)):
  if(search==x[i]):
    print("The no. found at index:",i)
    found=True
    break;
if (found==False):
    print("The no. is not present in their list")

print("Name:-Sushant Poojary,"
  "roll no. - 1767")
```

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 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 can be carried out by:

↳ The User is supposed to enter data in sorted manner.

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

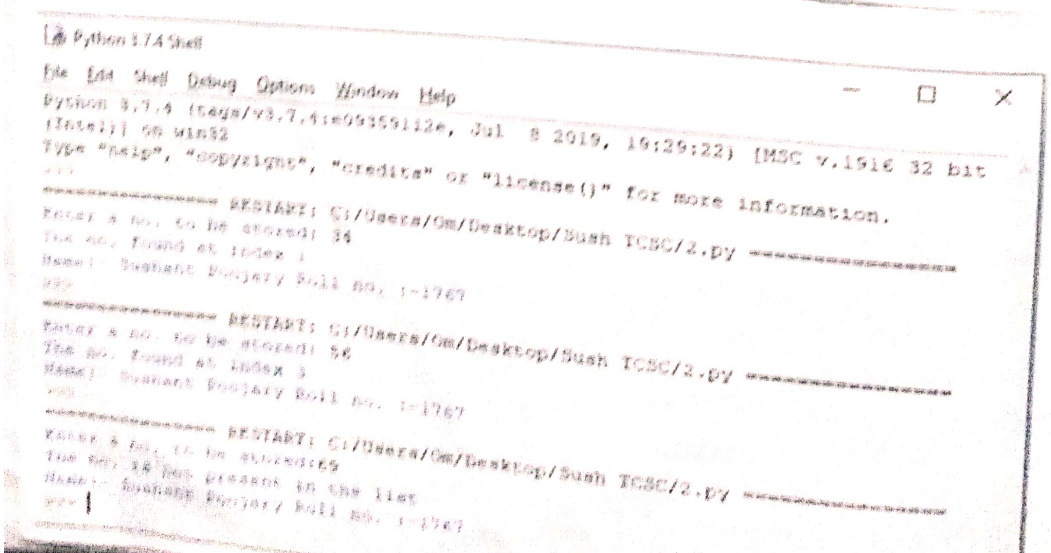
↳ If element is found display with an updation as value is stored from location '0'.

↳ 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 the not then without any processing we can say number not in the list.

```
found=False
z=[12,34,45,56,76]
search=int(input("Enter a no. to be stored:"))
if(search<z[0] or search>z[len(z)-1]):
  print("No. doesn't exist")
else:
  for i in range(len(z)):
    if (search ==z[i]):
      print("The no. found at index",i)
      found=True
      break;
  if(found==False):
    print("The no. is not present in the list")

print("Name:- Sushant Poojary",
  "Roll no. :-1767")
```

Output:



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 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.

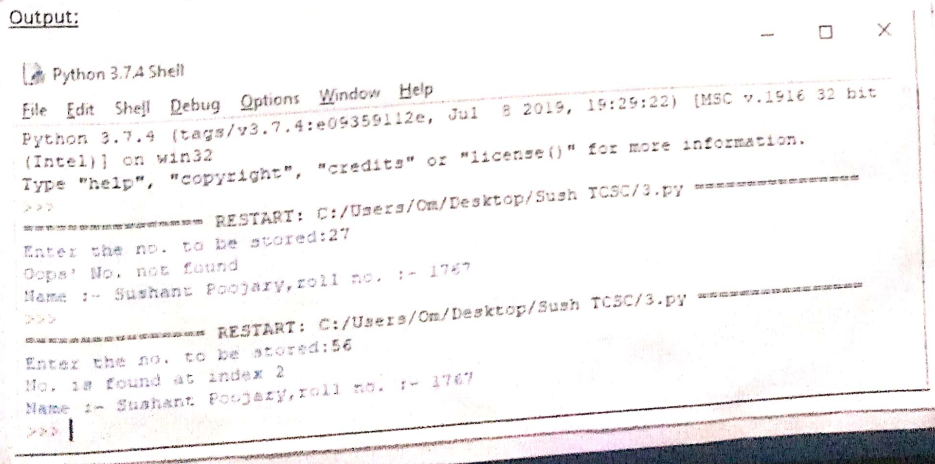
If the key value is less than or greater than this middle element, the algorithm knows which half of the array to continue searching in because the array is sorted.

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 in half a binary search will completely succesful in logarithmic time.

```
a=[7,34,56,78,89]
search=int(input("Enter the no. to be stored:"))
l=0
r=len(a)-1

while(True):
  m=int((r+1)/2)
  if(l>r):
    print("Oops! No. not found")
    break;
  if(search==a[m]):
    print("No. is found at index",m)
    break;
  else:
    if (search<a[m]):
      r=m-1
    else:
      l=m+1

print("Name :- Sushant Poojary,"
  "roll no. :- 1767")
```

Output:



Practical - 5

Aim : To demonstrate the use of stack

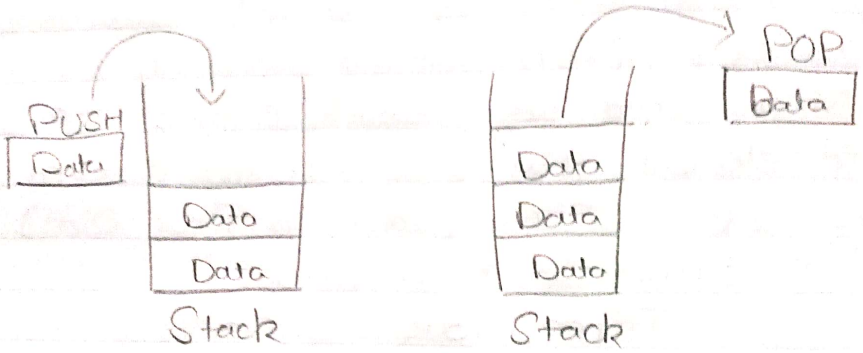
Theory : In Computer science, a stack is always an abstract data type that serves as a collection of element with two principal operations Push, which adds an element to the collection and pop, which removes the most recently added element that was not yet removed. The order may be LIFO (Last in First Out) or FIFO (First In First Out)

Three 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 pushed. If the stack is empty, then it is said to be an underflow condition
- Peek or Top : Returns top element of stack.

```
##Stack###
print("SUSHANT POOJARY","1767")
class stack:
    global tos
    def __init__(self):
        self.l=[0,0,0,0,0,0,0,0]
        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()
```

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



Last In - First - Out

OUTPUT:

```
SUSHANT POOJARY 1767
stack is full
data = 70
data = 60
data = 50
data = 40
data = 30
data = 20
data = 10
stack empty
```

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 given 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 (deque) because queue is open at both of its ends.

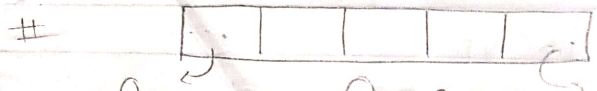
enqueue() can be termed as add() in queue i.e. adding a element in queue.

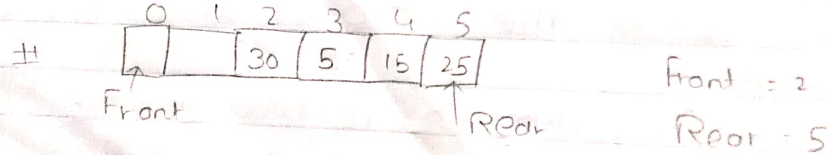
Dequeue() can be termed as delete or Remove. i.e. deleting or removing of element.

```
##Queue and add Delete##
print("SUSHANT POOJARY","1767")
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()
```

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

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





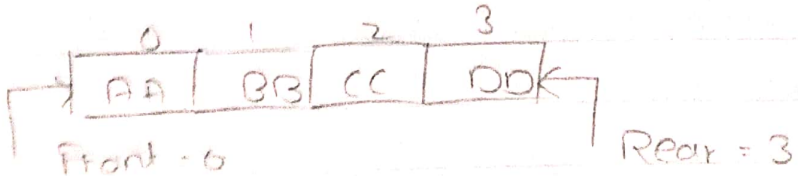

```
SUSHANT POOJARY 1767
Queue is full
30
40
50
60
70
Queue is empty
```

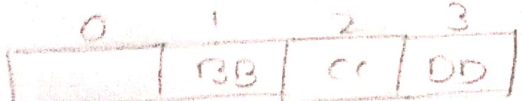
Practical - 7

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

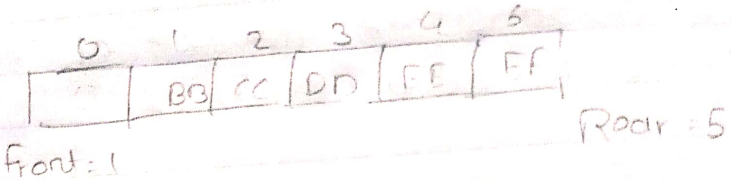
Theory : The queue that we implement using an array suffer from one limitation. In that implementation there is a possibility that the queue is reported as full, even though in actuality there might be empty slots at the beginning of the queue
To overcome this limitation we can implement queue as circular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

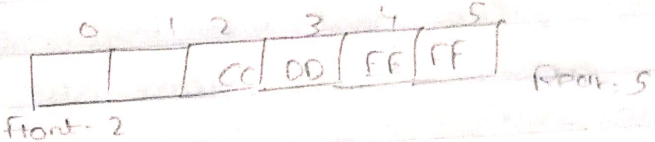
Example :

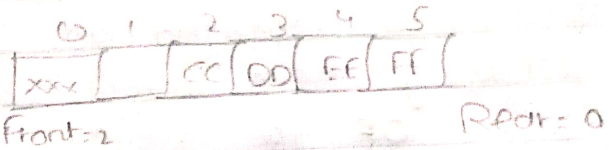




```
#(circular queue)
print("SUSHANT POOJARY","1757")

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:

```
SUSHANT POOJARY 1767
data added :  44
data added :  55
data added :  66
data added :  77
data added :  88
data added :  99
data removed :  44
>>>
```

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 as sinking sort.
Is a simple sorting algorithm that repeatedly steps through the lists, compares adjacent ~~rotes~~ 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 if compares one element checks if condition fails then only swaps otherwise goes on.

```
print( "Sushant")
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[j+1]):
            t=a[j]
            a[j]=a[j+1]
            a[j+1]=t
print(a)
```

Output

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

Example:

First pass

(5 1 4 2 8) → (1 5 4 2 8) Here algorithm compares the first two elements and 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) Now since these elements are already in order (8 > 5) algorithm does not swap them.

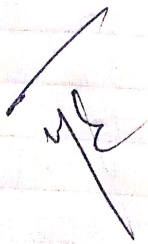
Second pass:

(1 4 2 5 8) → (1 4 2 5 8)

(1 4 2 5 8) → (1 2 4 5 8) swap sin 4 > 2

Third Pass:

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



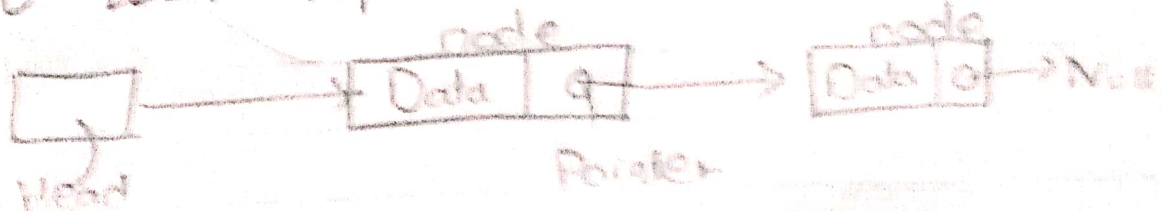
Practical - 8

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.

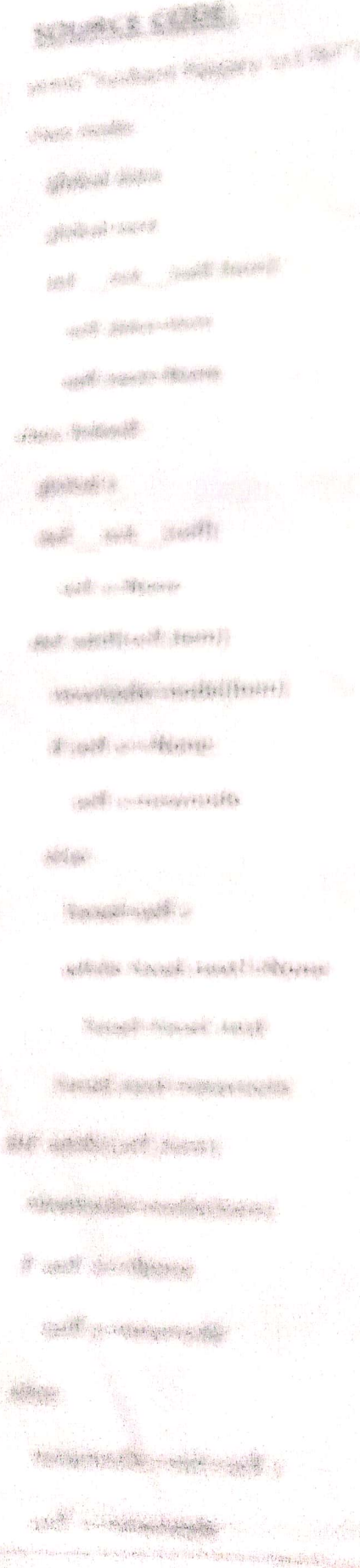
- LINK - Each link of a linked list can store a data called an element.
- NEXT - Each link of a linked list contains a link to the next link called NEXT.
- LINKED LIST - A linked list contains the connection link to the first link called first.

LINKED ~~LIST~~ Representation:



PRACTICAL-8

SOURCE CODE:



Types of Linked List:

→ Simple
→ Doubly
→ Circular.

Basic operations.

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

```
def display(self):
    head=self.s
    while head.next!=None:
        print(head.data)
        head=head.next
    print(head.data)
start=linkedl()
start.addl(50)
start.addl(60)
start.addl(70)
start.addl(80)
start.addb(40)
start.addb(30)
start.addb(20)
start.display()
```

OUTPUT:

```
Sushant Poojary
1767
20
30
40
50
60
70
80
```

PRACTICAL-9

SOURCE CODE:

```
print("Sushant Poojary\n1767")
def eva(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))
```

Practical - 8

Aim : To evaluate postfix expression using stack.

Theory : Stack is an (ADT) and 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.

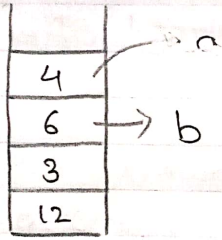
Steps to be followed:
1.) Read all the symbols one by one from left to right in the given postfix expression.
2.) If the reading symbol is operand then push it on the stack
3) If the reading symbol is operator (+, -, *, /, etc) then perform TWO pop, operations and store the two popped operands in two different variables (operand1 & operand2) Then program reading symbol operation using operand 1 & operand 2 and push result back on to the stack.
4.) Finally! Perform a pop operation and display the popped valueas

final result.

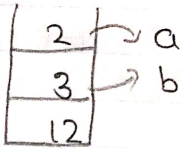
Value of postfix expression:

S = 1 2 3 6 4 - + *

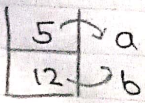
Stack:



b - a = 6 - 4 = 2 // Store again in Stack



b + a = 3 + 2 = 5 // Store result in Stack

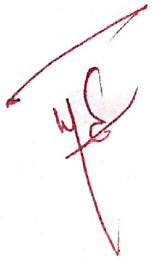


b * a - 12 * 5 = 60

```
    return stack.pop()
s="5 3 + 8 2 - *"
r=eva(s)
print("The evaluated value is:",r)
```

OUTPUT:

```
Sushant Poojary
1767
The evaluated value is: 48
```



```
print( "Sushant")
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

Sushant

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

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

Practical - 10

Selection Sort

Aim : To demonstrate working of Selection Sort in data structure.

Theory :

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. It has an $O(n^2)$ time complexity, which makes it inefficient on target lists, and generally performs worse than the similar insertion sort.

Worst complexity :- n^2

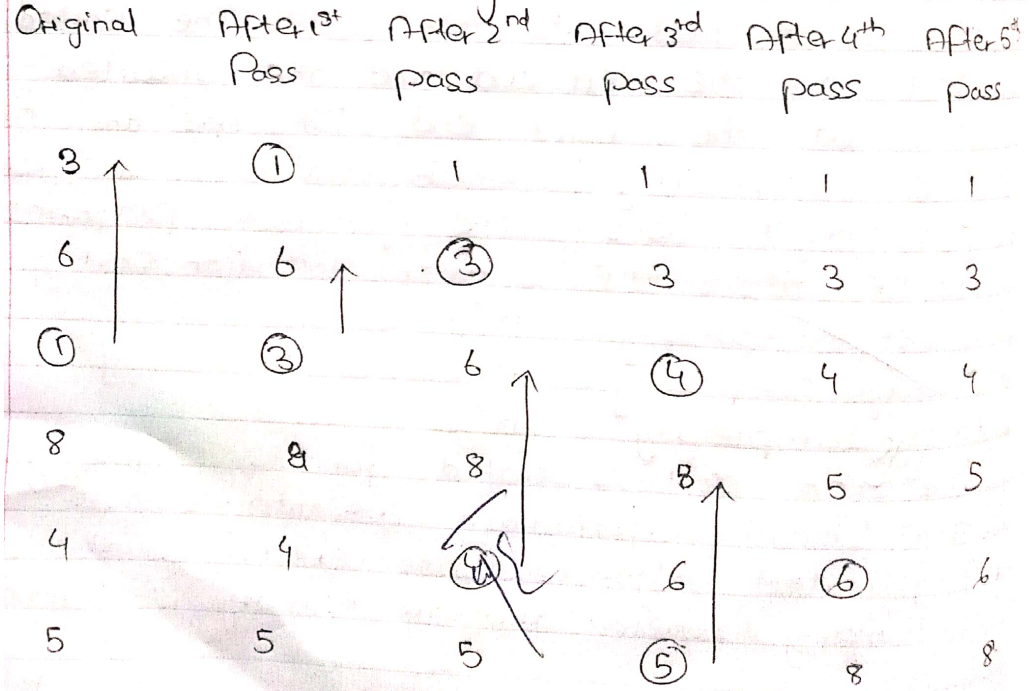
Average complexity :- n^2

Best complexity :- n^2

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

The time efficiency of selection sort is quadratic, so there are a number of sorting techniques which have better time complexity than selection sort. One thing which distinguishes selection sort from other sorting algorithms is that it makes the minimum possible number of swaps, n-1 in the worst case. It compares a element with every element of list and sorts accordingly.

Working.



Practical - 11

Quick Sort

Aim :- To demonstrate use of Quick Sort in data structure.

Theory :-

Quicksort is an efficient sorting algorithm developed by British Computer Scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competition, merge sort and heap sort.

Worst complexity :- n^2

Average complexity :- $n * \log(n)$

Best complexity :- $n * \log(n)$

Method :- Partitioning Quicksort is a divide and conquer algorithm.

It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot.

SOURCE CODE:

```
print( " Sushant\n1767")
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
```

The sub arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Efficient implementations of quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.

Working

(a) 2 8 7 1 3 5 6 [4] — Pivot

(b) 2 1 [3] 4 7 5 6 8 — Pivot

(c) 2 [1] 3 4 7 5 [6] 8 — Pivot

(d) 1 2 3 4 [5] 6 7 8 — Pivot

(e) 1 2 3 4 5 6 7 8

```
    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:

```
Sushant
1767
[42, 45, 54, 55, 66, 67, 80, 89, 100]
```

Practical 12

Topic :- Binary Tree. and Traversal.

Procedure :- It is a tree which supports maximum of 2 children for any node within the tree and thus any kinds of trees are available with different features. The binary tree which is a finite set of elements that is either empty or further divided into sub trees. There are two ways to represent binary trees

- Using arrays
- Using linked lists

A binary tree is a special type of tree in which every node or vertex has either no child node or more child node or two child nodes. A binary tree is an important class of tree data structure in which a node can have at most two children.

- A binary tree is either an empty tree.
- Or a binary tree consists of a node called the root node, a left subtree and a right subtree, both of which will act as a binary tree once again.

Inorder Traversal:

In this traversal method the left subtree is visited first, then the root and later the right sub-tree. If a binary tree is traversal in-order, the output will produce sorted key value in an ascending order.

Pre-order Traversal:-

In this traversal method, the root node is visited first, then the left sub tree and finally the right subtree.

Postorder Traversal:

In this traversal method, the root is visited last, hence the name first we traverse the left sub-tree, then the right subtree and finally the root node.