Practical - 1

Aim:- To search a number from the 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.

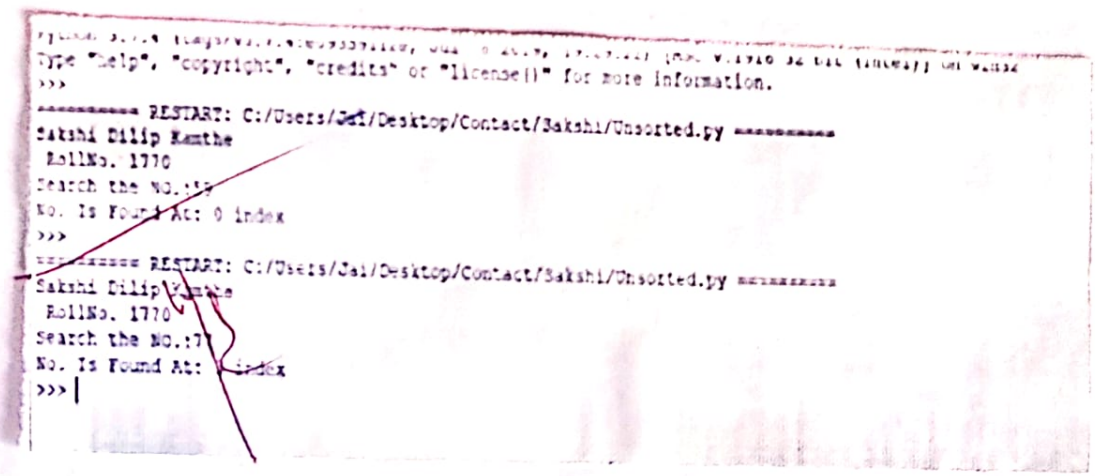
The linear search is further classified as

a. Sorted
b. Unsorted

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 element to be searched are not specifically arranged in ascending or descending order. They are arranged in random manner. This is what called as unsorted linear search. For unsorted or random search of mart the unsorted linear search method

```
found=False
A=[59,70,61,77,54]
print("Sakshi DilipKamthe \n RollNo. 1770")
search=int(input("Search the NO.:"))
for i in range(len(A)):
   if(search==A[i]):
print("No. Is Found At:",i,"index")
      found=True
      break;
if (found==False):
print("Not Found")
```



Unsorted linear search

1. The data is entered in random manner
2. User needs to specify the element to be searched in the entered list
3. check the condition that whether the entered number matches if matches then display the location plus increment i as data is stored from location 2010.
4. If all elements are checked one by one & element not found then prompt message number not found.

16 CS

Practical - 2

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

Theory: Searching & sorting are different modes or types of data structures.
Sorting - To basically sort the inputs data in ascending or descending manner.
Searching - To search elements & to display the same.

In searching the two in linear sorted search, the data is arranged in ascending to descending or descending to ascending that is all what is meant by searching through 'sorted' that it will arrange data.

Sorted linear search

1. The user is supposed to enter data in sorted manner.
2. User has to give an element for searching through sorted list.
3. If element is found display with an updation as value is stored from location 'o'.
4. If data or element not found Print the same.

```
found=False
print("Sakshi DilipKamthe \n RollNo. 1770")
A=[59,61,70,77,99]
search=int(input("Search the No.:"))
if(search<A[0] or search>A[len(A)-1]):
print("Not Found")
else:
   for i in range(len(A)):
      if(search==A[i]):
print("No. Is Found At:",i,"index")
         found=True
         break;
   if (found==False):
print("Not Found")
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/Jai/Desktop/Contact/Sakshi/Sorted.py ==========
Sakshi Dilip Kamthe
 RollNo. 1770
Search the No.:78
Not Found
>>>
========== RESTART: C:/Users/Jai/Desktop/Contact/Sakshi/Sorted.py ==========
Sakshi Dilip Kamthe
 RollNo. 1770
Search the No.:77
No. Is Found At: 3 index
>>>
```

In sorted order list of element user can check the condition that whether the entered number lies from starting point to the last element of not then without any processing user can say number not in the list.

Practical-3

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

Theory: A binary search also known as a half-internal 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.

Each step of algorithm a comparition is made & Procedure branches into one of the two directions specifically the Key value is ~~less~~ + compared to the middle elements 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 segment of the array until the value is located. Because each step in the algorithm divides the array size half a binary search will complete successfully in logarithmic.

```
A=[59,61,70,77,99]
print("Sakshi DilipKamthe \n RollNo. 1770")
search=int(input("Search the No.:"))
l=0
r=len(A)-1
while True:
   m=(l+r)//2
   if(l>r):
print("Not Found")
      break;
   if(search==A[m]):
print("Number Is Found At:",m,"index")
      break;
   else:
      if(search<A[m]):
         r=m-1
      else:
       r=m+1
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/Saf/Desktop/Contact/Sakshi/Binary.py ==========
Sakshi Dilip Kamthe
 RollNo. 1770
Search the No.:46
Not Found
>>>
========== RESTART: C:/Users/Saf/Desktop/Contact/Sakshi/Binary.py ==========
Sakshi Dilip Kamthe
 RollNo. 1770
Search the No.:70
Number Is Found At: 2 index
>>>
```

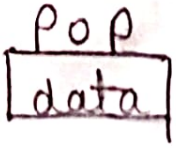
```
print("Sakshi  \n1770")
class stack:
    global tos
    def __init__ (self):
        self.l=[0,0,0,0,0,0,0]
        self.tos=-1
    def push(self,data):
        n=len(self.l)
        if(self.tos==n-1):
            print("Stack is full")
        else:
            self.tos=self.tos+1
            self.l[self.tos]=data
    def pop(self):
        if(self.tos<0):
                print("stack empty")
        else:
                k=self.l[self.tos]
                print("data=",k)
                self.tos=self.tos-1
```

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).
Three Basic operations are Performed in the stack.

* 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.
* PUSH:- Adds an item in the stack if it is full then it is said to be over flow condition.
* PEEK OR TOP: Returns top element of stack
* is Empty: Returns true if stack is empty else false



stack

Last - in - First - out

```
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:

```
Sakshi
1770
Stack is full
data= 70
data= 60
data= 50
data= 40
data= 30s
data= 20
data= 10
stack empty
```

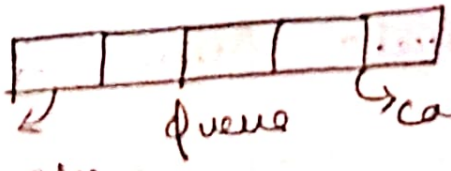
SOURCE CODE:

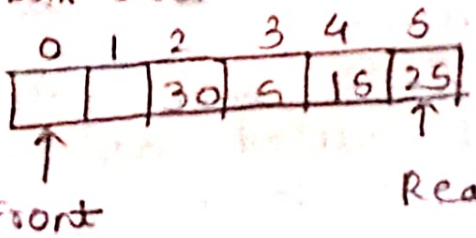
```
print("Sakshi \n 1770")
class Queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if (self.r<n-1):
      self.l[self.r]=data
      self.r=self.r+1
    else:
      print("Queue is full")
  def remove(self):
    n=len(self.l)
    if (self.f<n-1):
      print(self.l[self.f])
      self.f=self.f+1
    else:
      print("Queue is empty")
```

:: To demonstrate queue add and delete

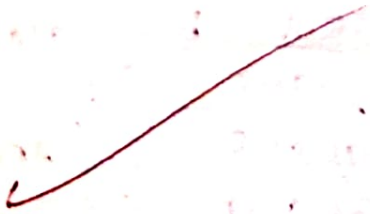
THEORY: Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT
Front Points to the beginning of the queue & Rear Points to the end of the queue.
Queue follows the FIFO (First-in-First-out) structure.
According to its FIFO structure element inserted first will also be removed first.
In a queue, one end is always used to insert data (enqueue) & 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 a 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.





Front = 2
Rear = 5



```
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:

```
Sakshi
1770
Queue is full
30
40
50
60
70
Queue is empty
```

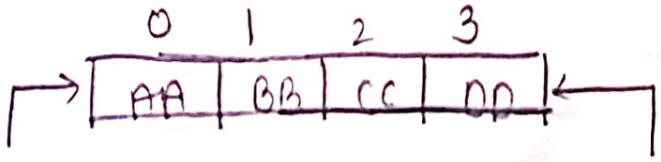
SOURCE CODE:

```
print("Sakshi \n 1770")
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)
```

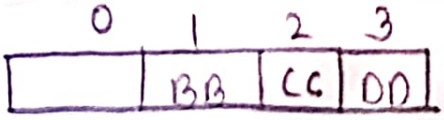
:- To demonstrate the use of circular queue as data structure.

: 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 be the queue & reach the end of the array. The next element is stored in the first slot of the array.

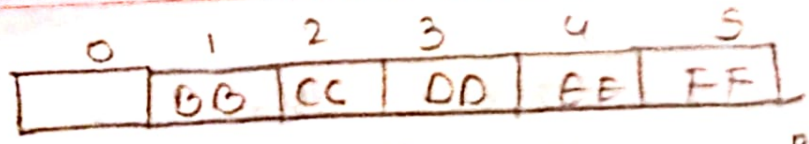
Example:



Front = 0 Rear = 3

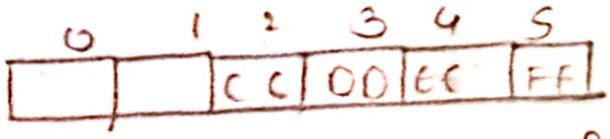


Front = 1 Rear = 3



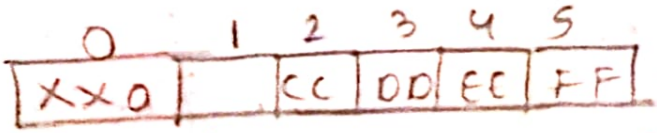
Front = 1

Rear = 5



Front = 2

Rear = 5



Front = 2

Rear = 0

```
            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:

```
Sakshi
 1770
data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
Queue is full
Data removed: 44
```

SOURCE CODE:

```
print("Sakshi \n1770")
class node:
    global data
    global next
    def __init__(self,item):
        self.data=item
        self.next=None
class linkedl:
    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
```

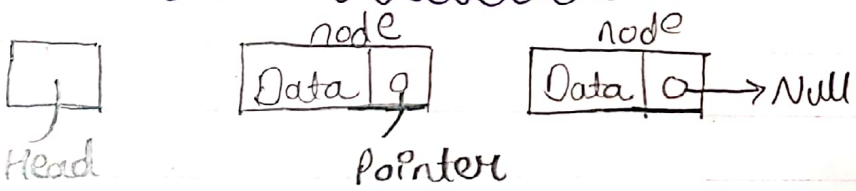
Practical - 8

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 :



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:

```
Sakshi
1770
20
30
40
50
60
70
80
```

SOURCE CODE:

```
print("Sakshi\n1770")
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))
```

:- 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 & 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.

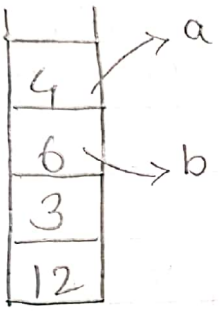
3] If the reading symbol is operator (+, -, *, /, etc) then Perform TWO Pop operations & store the two Popped operands in two different variables (operand 1 & operand 2). Then perform reading symbol operation using operand 1 & operand 2 & Push result back on to the stack.

4] Finally! Perform a Pop operations & display the popped value as final result.

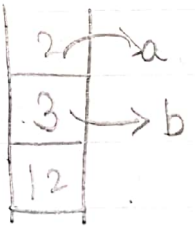
value of postfix expression:

S = 12 3 6 4 − + *

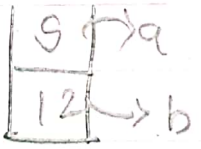
stack:



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



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



b * a − 12 * 5 = 60 //.

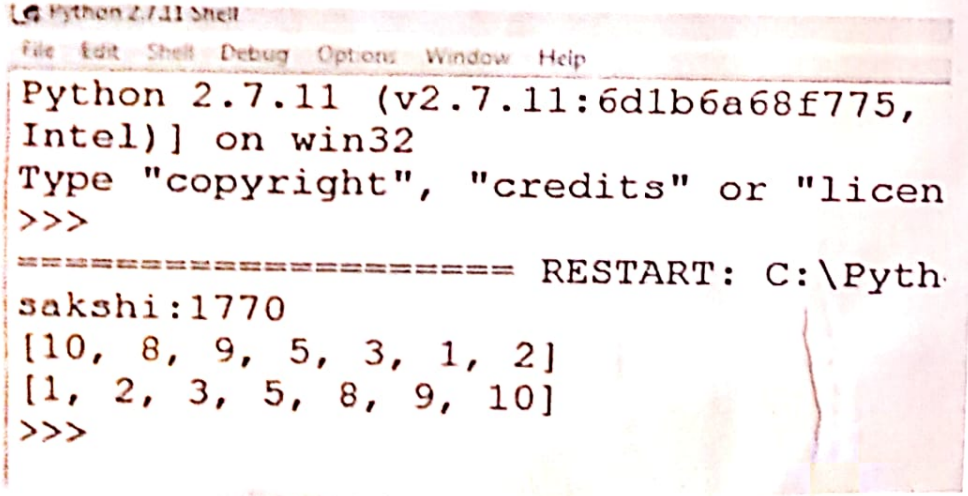
```
    return stack.pop()
s="5 3 + 8 2 - *"
r=eva(s)
print("The evaluated value is:",r)
```

OUTPUT:

```
Sakshi
1770
The evaluated value is: 48
```

```
print("sakshi:1770")
a=[10,8,9,5,3,1,2]
print(a)
for i in range(len(a)-1):
for j in range(len(a)-1-i):
if(a[j]>a[j+1]):
        t=a[j]
a[j]=a[j+1]
a[j+1]=t
print(a)
```

OUTPUT:



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

Theory:

SORTING

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

Bubble sort sometimes referred as sinking sort.

It is a simple sorting algorithm that repeatedly steps through the lists, compares adjacent element & 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 comparison sort in named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too. as it compares one element checks it condition fails then only swaps otherwise goes on.

Practical-10

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

Theory:

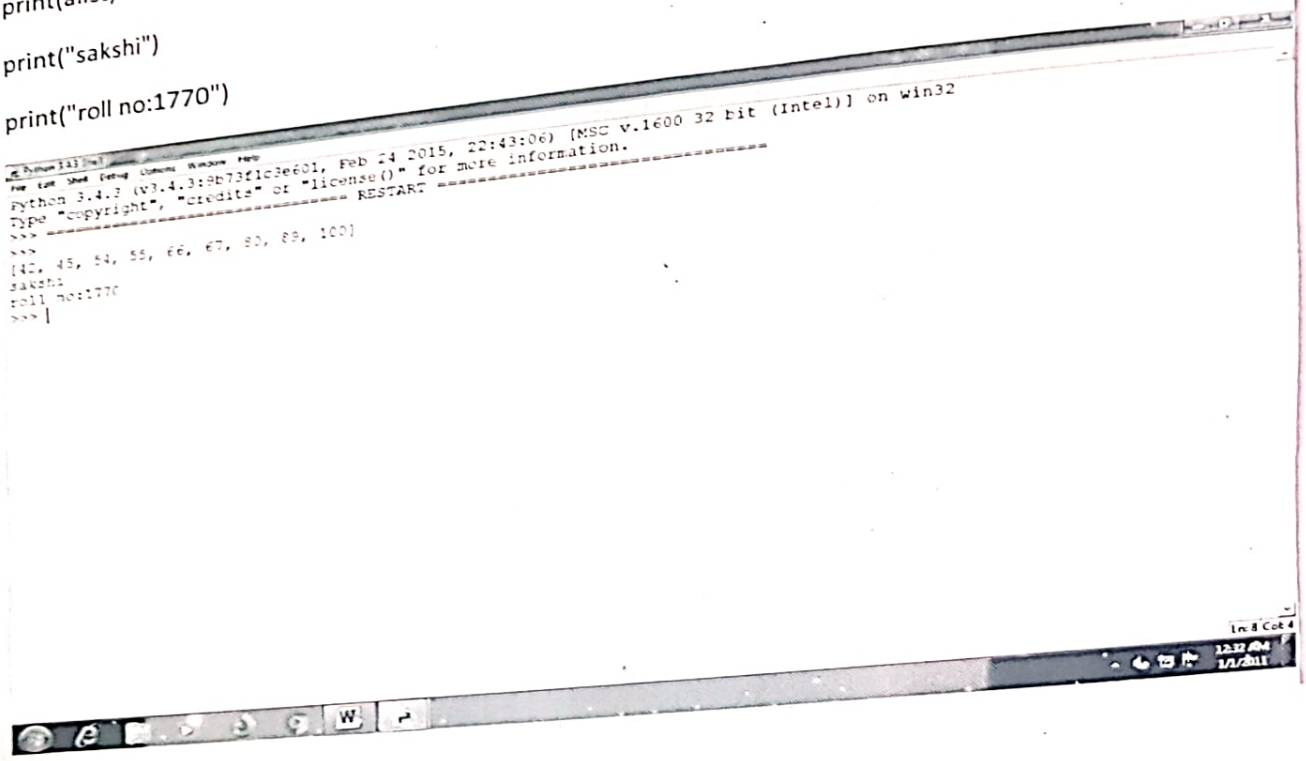
Quicksort is an efficient sorting algorithm type of a divide & conquer algorithm. It picks an element as Piror & Partitions the given array around the picked Pirot.

There are many different versions of quick sort that Pick Pirot in different ways.

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

```
def quicksort(alist):
quicksortHelper(alist,0,len(alist)-1)
defquicksortHelper(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:
whileleftmark<=rightmark and alist[leftmark]<=pivotvalue:
leftmark=leftmark+1
whilealist[rightmark]>=pivotvalue and rightmark>=leftmark:
rightmark=rightmark-1
ifrightmark<leftmark:
done=True
else:
temp=alist[leftmark]
alist[leftmark]=alist[rightmark]
alist[rightmark]=temp
temp=alist[first]
alist[first]=alist[rightmark]
```

```
alist[rightmark]=temp

returnrightmark

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

quicksort(alist)

print(alist)

print("sakshi")

print("roll no:1770")
```



The Key Process is quicksort is Partition().
Target of Partitions is given as array & an element x of array as Pivot, Put X as its correct position in sorted array & than ~~x bef~~. 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.

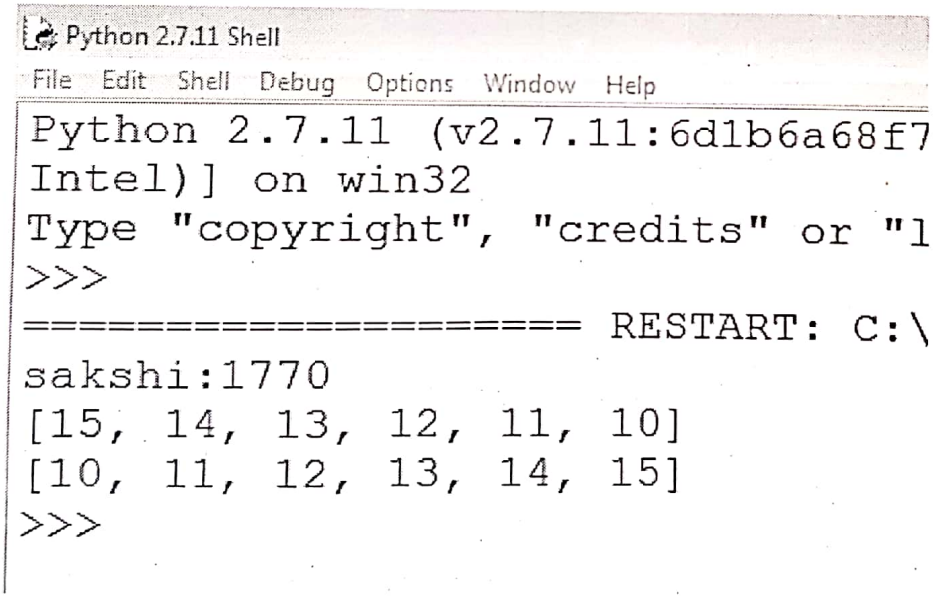
Theory:

Selection sort is a simple sorting algorithm. This sorting algorithm is an inplace comparison algorithm in which the list is divided into two parts, the sorted part at the left & the unsorted part at the right. Initially, the sorted part is empty and the unsorted part is the entire list.

PROGRAM 11:

```
print("sakshi:1770")
b=[15,14,13,12,11,10]
print(b)
for i in range(len(b)-1):
for j in range(len(b)-1):
if (b[j]>b[i+1]):
        t=b[j]
b[j]=b[i+1]
b[i+1]=t
print(b)
```

OUTPUT:



```
print("sakshi:1770")
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):
ifself.root==None:
self.root=Node(val)
else:
newnode=Node(val)
        h=self.root
while True:
ifnewnode.data<h.data:
ifh.l!=None:
            h=h.l
else:
h.l=newnode
print(newnode.data,"added on left of",h.data)
```

Practical No-12

Aim Binary Tree & Traversal.

Theory:

A Binary Tree is a special type of tree in which every node or vertex has either child or one child node or two child.

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

Traversal is a process to visit all the nodes of a tree and may print their values.

There are 3 ways in which we use to traverse a tree.

(A) inorder.

(B) Preorder.

(C) Postorder.

(A) inorder

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.

(B) PREORDER

The root node is visited 1st then the left subtree & finally the right subtree

```
else:
ifh.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)
definorder(self,start):
if start!=None:
self.inorder(start.l)
print(start.data)
self.inorder(start.r)
defpostorder(self,start):
if start!=None:
self.postorder(start.l)
self.postorder(start.r)
print(start.data)
T=Tree()
T.add(100)
T.add(35)
T.add(70)
```

```
T.add(15)
T.add(95)
T.add(97)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
```

OUTPUT:

```
Python 2.7.11 Shell
File  Edit  Shell  Debug  Options  Window  Help
Type "copyright", "credits" or "license()" fo
>>>
========== RESTART: C:\Python27\pra
sakshi:1770
(35, 'added on left of', 100)
(70, 'added on right of', 35)
(15, 'added on left of', 35)
(95, 'added on right of', 70)
(97, 'added on right of', 95)
preorder
100
35
15
70
95
97
inorder
15
35
70
95
97
100
postorder
15
97
95
70
35
100
```

(c) POST ORDER.

The root node is visited last left subtree, then the right subtree & finally the root node.

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( ). it one of the most respected Algorithm.

Merge sort first divides the array into equal values & then combines them in a sorted manner.

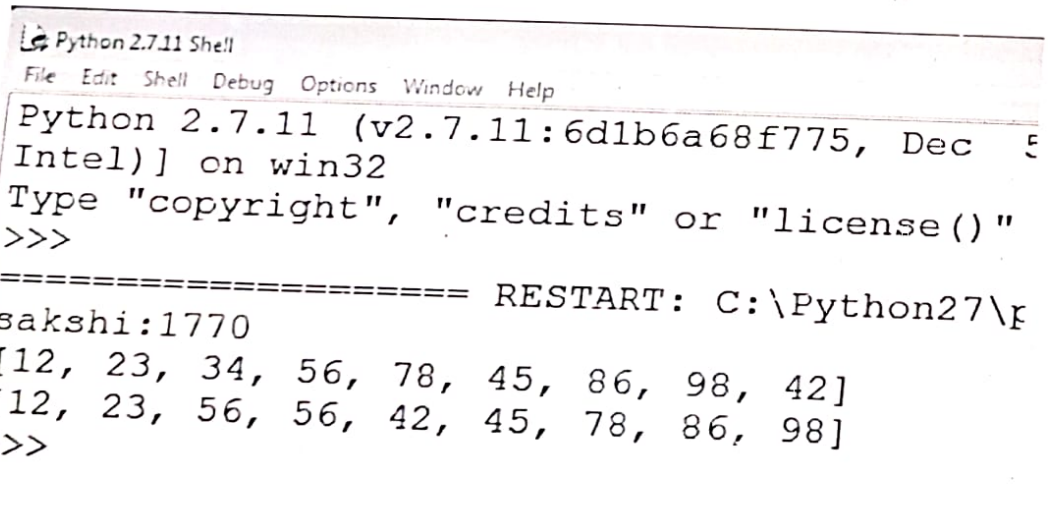
If divides input array into two values & calls itself for the two values & then merge the two sorted values.

The merge() function is used for merging two values.

```
print("sakshi:1770")
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
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
        defmergesort(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]
    print(arr)
   n=len(arr)
  mergesort(arr,0,n-1)
 print(arr)
```

OUTPUT:



The merge Process (arr, l, r, m) is key Process that assumes that arr [l ... m] and arr [m+1 ... r] are sorted & merges the two sorted sub-array into one.