Source code:

```
print("Yashvi Shah \n 1706")
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:

Case1:
Yashvi Shah
1706
[3, 33, 4, 45, 23, 79, 38]
Enter no to be searched: 38
Number found at: 7
Case 2:
Yashvi Shah
1706
[3, 33, 4, 45, 23, 79, 38]
Enter no to be searched: 99
Number NOT FOUND!

Practical - 1

AIM:- To search a number from the list using Linear unsorted.

THEORY: The process of identifying or finding a particular record is called searching. There are two types of search
- Linear search
- Binary search

The Linear search is further classified as
- sorted
- unsorted.

Here we will see about the unsorted Linear search.

Linear search also known as sequential search is a process that checks every element in the list is sequentially until the desired element is found.

Unsorted data means when the data is not arranged randomly it is unsorted.

The element to be searched is entered by the user. After which the element is compared with the data. When the match is found the location of the element is displayed accordingly.

If element is found at 0th location it is best case. If the element is found at nth loc, it is the worst case.

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 order 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 a logical way.

It is a technique to compare each and every element with the key element to be found, if both of them matches, the algorithm returns that element along with its position.

If the data is found at the 0th position it is the best case

If the data is found at the nth position it is the worst case.

Source code:

```
print("Yashvi Shah \n1706")
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:

Case1:
Yashvi Shah
1706
[3, 10, 21, 59, 69, 74, 80]
Enter the number to be searched: 21
Number found at: 3
Case2:
Yashvi Shah
1706
[3, 10, 21, 59, 69, 74, 80]
Enter the number to be searched: 99
Number doesnt exist!
Case3:
Yashvi Shah
1706
[3, 10, 21, 59, 69, 74, 80]
Enter the number to be searched: 11
Number NOT found!

```
#Source code:
print("Yashvi shah\n1706")
a=[3,12,23,36,49,66,78]
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:

Case1:

Yashvi shah
1706
[3, 12, 23, 36, 49, 66, 78]
Enter number to be searched from the list:36
Number found at location: 4

Case2:

Yashvi shah
1706
[3, 12, 23, 36, 49, 66, 78]
Enter number to be searched from the list:99
Number not in RANGE!

Case3:

Yashvi shah
1706
[3, 12, 23, 36, 49, 66, 78]
Enter number to be searched from the list:11
Number not in given 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-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 in half a binary search will complete successfully in Logarithmic terms.

Source code:

[illegible]

Output:

[illegible]

Practical - 4

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. It is a simple sorting algorithm that repeatedly steps through the list, 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 comparison 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 only then it swaps otherwise it continous

example:

(1 4 5 2 8) → (1 4 2 5 8) swaps since 5 > 2
(1 4 2 5 8) → (1 2 4 2 8) swaps since 5 > 1
(1 2 4 5 8) → (1 4 5 2 8) swaps since 5 > 4
(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 since 4 > 2
(1 2 4 5 8) → (1 2 4 5 8)

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

PRACTICAL-5

SOURCE CODE:

```
print"Yashvi Shah \n1706"
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()
```


Practical 5

Aim:- Demonstrate the use of stack

Theory:- A stack is a container of objects that are inserted and removed according to the LIFO (last in first out) principle. In the pushdown stacks only two operations are allowed: push the items into the stack and pop the item out of the stack. push adds an item to the top of the stack, pop removes the item from the top. A stack is a recursive data structure. Applications: • The simplest application of a stack is to reverse a word. • Another application is an 'undo' mechanism in text editors. • Language processing • Space for parameters and local variables is created internally using stack, Compiler's syntax check for matching braces is implemented by using stack, support for recursion. In the standard library of classes, the data type stack is an adapter class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, list, linked list or any other collection.

Regardless of the type of the underlying data structure, a stack must implement the same functionality. Stack is a linear data structure which follows a particular order in which the operations are performed. Implementing stack using an array is easy to implement. Memory is saved as pointers are not involved. Though it is not dynamic. And doesn't grow and shrink depending on needs at runtime.

OUTPUT:

Yashvi Shah
1706
Stack is full
data= 70
data= 60
data= 50
data= 40
data= 30s
data= 20
data= 10
stack empty

PRACTICAL-6

SOURCE CODE:

```
print("Yashvi Shah \n 1706")
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()
```


Practical 6

Aim: To demonstrate add and delete in queue

Theory:

Queue is also an abstract data structure or a linear data structure, just like data structure here the first element is inserted from one end called REAR and removal of existing element takes place from the other end called as FRONT. This makes queue as FIFO (first in first out) data structure, which means that element inserted first will be removed first. The process to add an element into queue is called enqueue and the process of removal of an element from queue is called Dequeue.

Applications: Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios.

- Serving request on a single shared resource like a printer. • CPU task scheduling etc.
- In real life scenario, call center phone systems uses queues to hold people calling them in an order, until a service representative is free.

• Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e first come first serve.
• Queue can be implemented using an array, stack or linked list. The easiest way of implementing a queue is by using an array. Initially the head and the tail of the queue point at the first index of the array.
• As we add elements to the queue the rear keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

OUTPUT:

```
Yashvi Shah
1706
Queue is full
30
40
50
60
70
Queue is empty
```

SOURCE CODE:

```
print("Yashvi Shah \n 1706")
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:

```
Yashvi Shah
```


Practical 7

Aim:- To demonstrate use of circular tube

Theory:

In a linear queue, once the queue is completely full, it is not possible to insert more elements. Even if we dequeue the queue to remove some of the elements until the queue is reset, no new elements can be inserted.

When we dequeue any element to remove it from the queue, we are actually moving the front of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the rear pointer is still at the end of the queue. The only way is to reset the linear queue for a fresh start.

Circular Queue is also a linear data structure, which follows the principle of FIFO, but instead of ending the queue at the last option, it again starts from the first position after the last, hence making the queue behave like a circular data structure. In case of a circular queue, head pointer will always point to the front of the queue and tail pointer will always point to the end of the queue

Initially, the head and the tail pointer will be pointing to the same location this would mean that the queue is empty. New data is always added to the location pointed by the tail pointer is incremented to point to the next avaible location.

Applications: Below we have some common real world examples where circular queues are used:

1. Computer controlled traffic signal system uses circular queue.
2. CPU scheduling and memory management.

1706
data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
Queue is full
Data removed: 44

PRACTICAL-8

SOURCE CODE:

```
print("Yashvi Shah \n1706")
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 add(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

Theory: A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

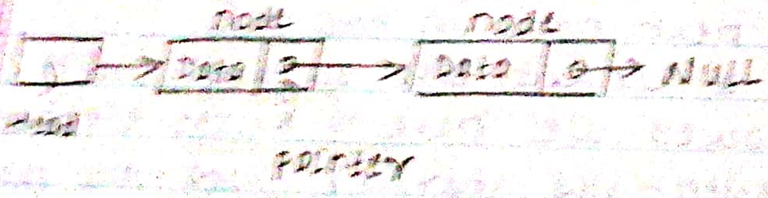
- In a linked list random access is not allowed we have to access elements sequentially starting from the first node so we cannot do binary search with linked lists efficiently with its default implementation.
- A linked list is represented by a pointer to the first node of the linked list. The first node is called the head If the linked list is empty, then the value of the head is null. Each node in a list consists of at least two parts: (1) Data (2) Pointer to the next node

Following are the basic operations

- Insertion: Adds an element at the beginning of the list.

- DelFron: Deletes an element at the beginning of the list.
- Display: Displays the complete list
- Search: searches an element using the given key.
- Delete:- Deletes an element using the given key.

LINKED LIST REPRESENTATION:-



FOLDER

```
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:

```
Keshvi Shah
1706
20
30
40
50
60
70
80
```

PRACTICAL-9

SOURCE CODE:

```
print("Yashvi Shah\n1706")
def eval(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 9

Aim: To evaluate postfix expression using stack

Theory:

- The Postfix notation is used to represent the algebric expressions. The infix notation as parenthesis are not required in postfix. We have discussed infix to postfix conversion. In this post, evaluation of postfix expressions is discussed

- Following is algorithm for evaluation postfix expressions

(1) Create a stack to store operands (or values)

(2) Scan the given expression and do following for every scanned element.

...a) If the element is a number, push it into the stack

...b) If the element is a operator, pop operands for the operator from the stack.
Evaluate the operator and push the result back to the stack.

13) When the expression is ended the number in the stack is the final answer.

14) Finally perform a pop operation and display the popped value as final result.

* Value of postfix expression:-

5 3 6 4 - + *

Stack:

6 - 4 = 6 - 4 = 2 // store again in stack

6 + 2 = 3 + 2 = 5 // store result in stack

5 * 5 = 12 * 5 = 60


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

OUTPUT:

```
Yashvi Shah
1706
The evaluated value is: 48
```

PRACTICAL-10

SOURCE CODE:

```
print("Yashvi Shah\n1706")
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
```


Practical-10

Aim: To sort an array using Quick Sort

Theory:-

Like merge sort, Quick sort is a Divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

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

The key process in quick sort is partition()
Target of partitions is, given on 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.

There can be many ways to do partition
The logic is simple we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. while traversing.

```
        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:

Yashvi Shah

1706

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

78

If we find a smaller element, we swap current element with a[i] otherwise we ignore current element. The worst case occurs when the partition process always picks greatest or smallest element as pivot. The best case occurs when the partition process always picks the middle element as pivot.

```
print ("Yashvi Shah\nFYBSC CS 1706")

#MERGE SORT METHOD
print ("\n* MERGE SORT METHOD\n")

def mergeSort(arr):

    if len(arr)>1:
        mid = len(arr)//2
        lefthalf = arr[:mid]
        righthalf = arr[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=j=k=0

        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                arr[k]=lefthalf[i]
                i=i+1

            else:
                arr[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            arr[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            arr[k]=righthalf[j]
            j=j+1
            k=k+1


arr = [27,89,70,55,62,99,45,14,10]
print("RANDOM LIST : ", arr)
mergeSort(arr)
print("\nMERGESORTED LIST : ",arr)
```


Practical - 11

Aim: To sort a list using merge sort

Theory:-

Like QuickSort, MergeSort is a Divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merge the two sorted halves. The merge() function is used for merging two 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-arrays into one.

The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged

APPLICATIONS:

- Merge sort is useful for sorting linked lists in O(n log n) Hon.
- Merg sort access data sequentially and the need of random access is low
- Inversion count problem.
- Used in External sorting.

Mergesort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular where sequentially accessed data structure are very common.


```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32                                    IDLE_tmp_4be126sh
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================ RESTART: C:\Users\DINESH\Desktop\DS P11.py ================
Yashvi Shah
FYBSC CS 1706

* MERGE SORT METHOD

RANDOM LIST :  [27, 89, 70, 55, 62, 99, 45, 14, 10]

MERGESORTED LIST :  [10, 14, 27, 45, 55, 62, 70, 89, 99]
>>>
```

```
print ("Yashvi Shah \nFYBSC CS 1706")

#BINARY TREE METHOD
print ("\n* BINARY TREE\n")

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):
```


Practical - 12

Aim:- Binary Tree and Traversal

Theory:- Tree is one of the most important non-linear data structures. Various kinds of trees are avaible 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 one 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 subtree.
If a binary tree is tranversed in-order, the output will produce sorted key values in an ascending order.
- Pre-order Tranversal:-
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 subtree, then the right subtree and finally the root node.

```
        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(10)
T.add(80)
T.add(40)
T.add(42)
T.add(20)
T.add(70)
T.add(50)
T.add(90)
T.add(80)

print("Preorder Numbers : ")
T.preorder(T.root)
print("Inorder Numbers : ")
T.inorder(T.root)
print("Postorder Numbers : ")
T.postorder(T.root)
```



```
----------
Yashvi Shah
FYBSC CS 1706

* BINARY TREE

80 Added on Right of 10
40 Added on Left of 80
42 Added on Right of 40
20 Added on Left of 40
70 Added on Right of 42
50 Added on Left of 70
90 Added on Right of 80
80 Added on Left of 90
Preorder Numbers :
10
80
40
20
42
70
50
90
80
Inorder Numbers :
10
20
40
42
50
70
80
80
90
Postorder Numbers :
20
40
42
50
70
80
80
90
10
```