Practical-1

Source code:

ds1.py - C:\Users\DELL\AppData\Local\Programs\Python\Python37-32\ds1.py (3.7.3)

File Edit Format Run Options Window Help

```
print("Sanjana Alwe \n 1709")
a=[5,28,22,65,24,26,2]
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 3.7.3 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:\Users\DELL\AppData\Local\Programs\Python\Python37-32\ds1.py ==
Sanjana Alwe
 1709
[5, 28, 22, 65, 24, 26, 2]
Enter no to be searched: 28
Number found at:  2
>>>
== RESTART: C:\Users\DELL\AppData\Local\Programs\Python\Python37-32\ds1.py ==
Sanjana Alwe
 1709
[5, 28, 22, 65, 24, 26, 2]
Enter no to be searched: 13
Number NOT FOUND!
>>>
```

Ln: 17 Col: 4

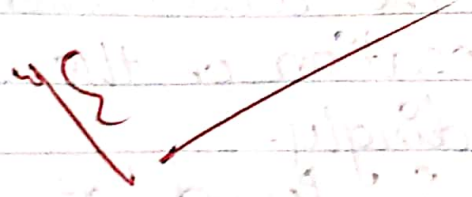
PRACTICAL - 1

Aim: To search an element using linear search method.

Theory: Linear search: The process of identifying or finding a particular record is called searching. Linear search is classified as:
- Sorted
- Unsorted

Linear search also known as sequential search is a process that checks every element in the list sequentially until the desired element is found.
Unsorted data means when the data is not arranged in a logical manner like ascending or descending order. When the data is 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 the element is found at the 0th location, it is the best case.
If the element is fount at the nth location, it is the worst case.

The indexing of the array starts from 0. Therefore while displaying the location it should be incremented by 1.
Suppose the array has n elements then the location of the n^{th} element is n+1



Practical-2

Source code:

ds2.py - C:\Users\DELL\AppData\Local\Programs\Python\Python37-32

File Edit Format Run Options Window Help

```
print("Sanjana Alwe \n1709")
a=[4,6,24,28,30,40,65]
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:

Python 3.7.3 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:\Users\DELL\AppData\Local\Programs\Python\Python37-32\ds2.py ==
Sanjana Alwe
1709
[4, 6, 24, 28, 30, 40, 65]
Enter the number to be searched: 66
Number doesnt exist!
>>>
== RESTART: C:\Users\DELL\AppData\Local\Programs\Python\Python37-32\ds2.py ==
Sanjana Alwe
1709
[4, 6, 24, 28, 30, 40, 65]
Enter the number to be searched: 28
Number found at:  4
>>>
== RESTART: C:\Users\DELL\AppData\Local\Programs\Python\Python37-32\ds2.py ==
Sanjana Alwe
1709
[4, 6, 24, 28, 30, 40, 65]
Enter the number to be searched: 31
Number NOT found!
>>>
```

PRACTICAL - 2

Aim: To search an element using linear sort method.

Theory: Linear search is one of the simplest searching method in which targeted item is sequentially matched with each item in the list.
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 is position.
Linear sort method means the data is sorted is ascending or descending order. The data is not taken randomly rather it is arranged in a logical way.
If the data is found at the 0^{th} position it is the best case.
If the data is found at the n^{th} position it is the worst case.

SE

PRACTICAL- 3

Aim: To search an element using binary search method.

Theory: Binary search also known as half-interval search is a search algorithm that finds the position of a target value within a sorted array.

Binary search compares the target value value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found.

If the search ends with the remaining half being empty, the target is not in the array.

Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search.

PRACTICAL-3

SOURCE CODE:

```
a=[3,4,21,24,25,29]
print("Sanjana Alwe \n 1709")
search=int(input("Enter the number to be searched: "))
l=0
h=len(a)-1
m=int((l+h)/2)
if((search<a[l])or(search>a[h])):
  print("Does not exist")
else:
  while(l!=h):
    if(search==a[m]):
      print("Number is found at: ",m)
      break
    else:
      if(search<a[m]):
        h=m-1
        m=int((l+h)/2)
      else:
        l=m+1
        m=int((l+h)/2)
if(search==a[l]):
  print("Number is found at location: ",l)
else:
  print("Number not found")
```

OUTPUT:

CASE1:

Sanjana Alwe

1709

Enter the number to be searched: 4

Number is found at location: 1

CASE2:

Sanjana Alwe

1709

Enter the number to be searched: 28

Number not found

CASE3:

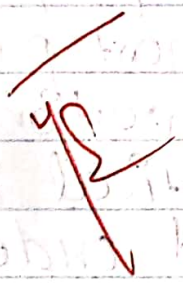
Sanjana Alwe

1709

Enter the number to be searched: 33

Does not exist

Binary search can be used to solve a wider range of problems, such as finding an element.
When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.



PRACTICAL - 4

Aim : To sort an array using bubble sort.

Theory: Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. Bubble sort can be practical if the input is in mostly sorted order with some out-of-order elements nearly in position.

A bubble sort, also called as sinking sort or exchange sort, is a sorting algorithm that compares adjacent pairs and swaps them if necessary, causing the items to "bubble" up toward their proper position. The process continues until no swaps are necessary.

PRACTICAL- 4

SOURCE CODE:

```
a=[72,24,67,38,11,99]
print("Sanjana Alwe \n 1709")
print("Array before sorting:\n",a)
for p in range (len(a)-1):
    for c in range (len(a)-1-p):
        if(a[c]>a[c+1]):
            t=a[c]
            a[c]=a[c+1]
            a[c+1]=t
print("Array after sorting: \n",a)
```

OUTPUT:

```
Sanjana Alwe
 1709
Array before sorting:
 [72, 24, 67, 38, 11, 99]
Array after sorting:
 [11, 24, 38, 67, 72, 99]
```

PRACTICAL-5

SOURCE CODE:

```
print("Sanjana Alwe \n1709")
class stack:
    global tos
    def __init__ (self):
        self.l=[0,0,0,0,0,0,0]
        self.tos=-1
    def push(self,data):
        n=len(self.l)
        if(self.tos==n-1):
            print("Stack is full")
        else:
            self.tos=self.tos+1
            self.l[self.tos]=data
    def pop(self):
        if(self.tos<0):
                print("stack empty")
        else:
                k=self.l[self.tos]
                print("data=",k)
                self.tos=self.tos-1
s=stack()
s.push(10)
s.push(20)
```

PRACTICAL-5

Aim: To 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 item into the stack and pop the item out of 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 is 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, an array, 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.

The linked list implementation of stack can grow and shrink according to the needs at run-time. But requires extra memory due to involvement of pointers.

```
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:

```
Sanjana Alwe
1709
Stack is full
data= 70
data= 60
data= 50
data= 40
data= 30s
data= 20
data= 10
stack empty
```

PRACTICAL-6

SOURCE CODE:

```
print("Sanjana Alwe \n 1709")
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")
```

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 requests 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 points 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

```
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:

```
Sanjana Alwe
1709
Queue is full
30
40
50
60
70
Queue is empty
```

PRACTICAL-7

SOURCE CODE:

```
print("Sanjana Alwe \n 1709")
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)
```

PRACTICAL - 7

Aim: To demonstrate the use of circular queue

Theory: In a linear queue, once the queue is completely full, it's 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, and once the data is added, tail pointer is incremented to point to the next available 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

```
        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)
```

data added: 88
Queue is full
Data removed: 44

OUTPUT:

```
Sanjana Alwe
1709
data added: 44
data added: 55
data added: 66
data added: 77
```

PRACTICAL-8

SOURCE CODE:

```
print("Sanjana Alwe \n1709")
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

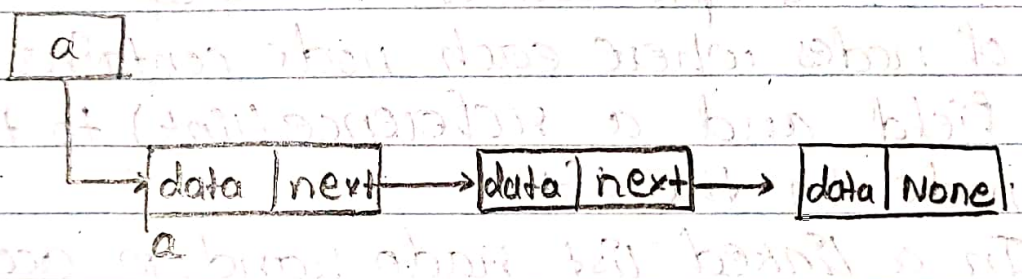
Theory: A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers ~~as~~ In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next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

In a linked list ~~nodo~~ 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 operation
- Insertion - Adds an element at the beginning of the list.

- Deletion - 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 ~~array~~ element using the given key.



```
    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:

```
Sanjana Alwe
1709
20
30
40
50
60
70
80
```

PRACTICAL-9

SOURCE CODE:

```
print("Sanjana Alwe\n1709")
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-9

Aim: To evaluate a postfix expression using stack

Theory: The postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as paranthesis are not required in postfix.

Following is algorithm for evaluation of postfix expressions:

1. Create a stack to store operands.
2. Scan the given expression and do the following for every scanned element
   - If the element is a number, push it into the stack
   - If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack.
3. When the expression is ended, the number in the stack in the final answer.

Let the expression be "53+82-*"

- Scan 5, its a number so push it back in stack
- Scan 3, its a number so push it to stack

Now the stack contains "5 3"

- Scan '+' its an operator pop two operands from stack, apply + operator, result is 8 so push it to stack
- Scan 8, again number push it in stack
- Scan 2, its a number so push it in stack
- Scan '-', pop two operands apply - operator result is 6 push it back
- Scan '*' pop two operators i.e 8 and 6 apply '*' operator result is 48 push it back in stack
- So now the stack only contains the result.

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

OUTPUT:

Sanjana Alwe

1709

The evaluated value is: 48

PRACTICAL-10

SOURCE CODE:

```
print("Sanjana Alwe \n1709")
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

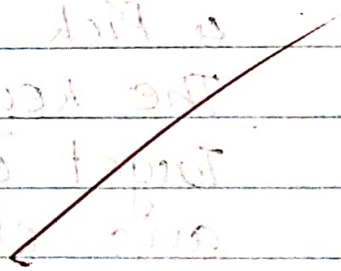
Theory: Like Merge Sort, Quick Sort is a Divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

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

The key process in quicksort is partition() Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

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

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.



```
        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:

```
Sanjana Alwe
1709
[42, 45, 54, 55, 66, 67, 80, 89, 100]
```

IDLE_tmp_txrua160

```
print ("Sanjana\nFYBSC CS 1709")

#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 merges 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:

1. Merge Sort is useful for sorting linked lists in O(n log n) time.
Merge sort accesses data sequentially and the need of random access is low.
2. Inversion Count Problem
3. Used in External Sorting

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

Sanjana
FYBSC CS 1709

* 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]
>>>

IDLE_tmp_zj_cfzav

```
print ("Sanjana \nFYBSC CS 1709")

#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: Trees is one of the most important non-linear data structures. Various 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 one child node or two child nodes. A binary tree is an important class of a tree data structure in which a node can have at most two children.

- 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

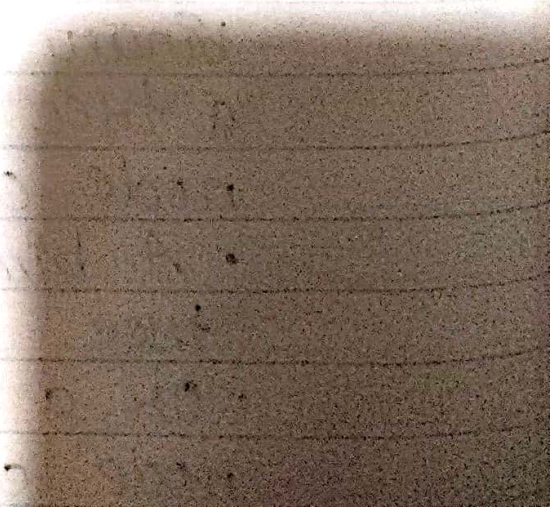
In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

- Pre-order Traversal

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

- Postorder Traversal

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

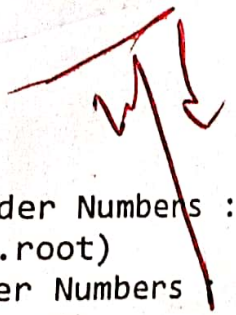


```
        if start!=None:                    IDLE_tmp_zj_cfzav
            self.inorder(start.l)
            print(start.data)
            self.inorder(start.r)

    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)
```



Sanjana
FYBSC CS 1709

* 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
```