PRACTICAL - 1

Aim :- Implement linear search to find an item in the list

Theory :-

Linear search

linear search is one of the simplest algorithm in which targeted item is sequentially matched with each item in the list.

It is worst searching algorithm with worst case time complexity. It is a force approach on the other hand incase of an ordered list, instead of searching the list in sequence. A binary search is used which will start by examining the middle item.

Linear search is a technique to compare each and every element with the key element to be found, if both of item matches, the algorithm returns that element found and its position is also found.

1) Unsorted
Algorithm:

Step1: Create an empty list and assign it to a variable

Step2: Accept the total no. of elements to be inserted into the list from the user say 'n'

Step3: Use for loop for adding the elements into the list.

Step4: Print the list

Step5: Accept an element from the user that is to be searched into the list.

Step6: Use for loop in a range from '0' to the total no. of elements to search the elements from the list.

Step7: Use of loop that the elements in the list is equal to element accepted from user

Step8: If the element is found then print the statement statement that the element is found along with the element position

* Unsorted program

```
x = [4, 3, 5, 8, 9, 2]
a = int(input("Enter a number:"))
for i in range(len(x)):
    if (a == x[i]):
        print("Element found at:", i)
        break
if (a != x[i]):
    print("No of elemet found")
```

Output

Enter a number : 3
Element found AT : 1

Enter a Number : 9
Element found at : 4

Enter a number : 10
No element found.

31 08

```
# Sorted Program
s = list (input ("Enter the Number : "))
s.sort()
print (s)
a = int (input ("Enter the number to be search"))
for i in range (len (s)):
    if (a == s[i]):
        print ("Number found in position", i)
        break
if (a != s[i]):
    print ("Number not found")
```

Output

Enter the number = 2, 5, 8, 9, 7

[1, 2, 3, 8, 9]

Enter the numbers to be searched: 5

Number found in position 2

Enter the number : 2, 3, 9, 6, 1

[1, 2, 3, 5, 9]

Enter the numbers to searched: 7

Number not found.

Step 10:- Draw the output of given algorithm

2] Sorted linear search :-

Sorting means to arrange the element in increasing or decreasing order.

- Algorithm :

Step 1 :- Create empty list and assign it to variable.

Step 2 :- Accept total no. of element to be inserted into the list from user, say 'n'

Step 3 :- Use for loop for using append() method to all element in the list

Step 4 :- Use sort() method to sort the accepted the element and assign in increasing order the list then print the list.

Step 5 :- Use of if statement to find the range in which element.

Step 6:- Use if loop that the elements in list equal to the element accepted from user.

Step 7:- Attach the input and output in above algorithm.

Source code :-

```
a = list(input("Element list : "))
a.sort()
c = len(a)
s = int(input("Enter list : "))
if (s>a[c-1] or s<a[0]):
    print("not in a list")

else :
    first, last = 0, c-1
    for i in range(0,c):
        m = int((first + last) / 2)
        print("found at", m)
        if s == a[m]:
            print("Number found")
            break
        else:
            if s<a[m]:
                last = m-1
            else:
                first = m+1
```


PRACTICAL-2

Aim :- Implement Binary search to find an searched no. in the list

Theory :

Binary Search

Binary search is also known as Half-interval search, logarithmic search or binary chop is a search algorithm, search to be ~~set~~ finds a position of a target value within a sorted array. If you are looking for the number which is at the end of the list then you need to search entire this can be avoided by using Binary Position search.

Algorithm :

Step 1 :- Create Empty list and ~~using~~ assign it to a variable

Step 2 :- Using input method, accept the range of given list.

Step 3 :- Use for loop, add elements in list, using append() method.

Step 4:- Use sort() to sort the accepted element and assign it in increasing to ordered list print the list after sorting.

Step 5:- Use if loop to give the range in which element is found in given range then display message "Element not found".

Step 6:- Then use the else statement, if statement is found in range then satisfy the below condition.

Step 7:- Accept an argument and by of the element that element has to be sorted.

Step 8:- Initialize first to 0 and last to last element of list a array is starting from 0 if it initialized less than the total count

Step 9:- Use for loop and assign to the given loop.

Step 10 :- Repeat till you found the element pick the input and output of above algorithm.

```
# Bubble sort
a = list(input("Enter Numbers : "))
for i in range(len(a)-1):
    for c in range(len(a)-1-i):
        if a[c]>a[c+1]:
            t = a[c]
            a[c] = a[c+1]
            a[c+1] = t
print(a)
```

Output :-

Enter Numbers : 2, 4, 8, 1, 9, 5, 6

[1, 2, 4, 5, 6, 8, 9]

Enter Numbers : 0.48, 0.32, 0.78, 0.19

[0.19, 0.32, 0.43, 0.78]

Enter Numbers : 232, 543, 123, 901

[123, 232, 543, 401]

PRACTICAL NO. 43

Bubble sort

- Aim :- Implementation of bubble sort program on given list.
- Theory :- Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their position if the exist in the wrong order, this is the simplest form of sorting available

Step 1 :- Bubble sort starts by comparing the first two element of an array and swapping if necessary.

Step 2 :- If we want to sort the elements of array in ascending order then first element is greater than second then i we need to swap element

Step 3 :- If the element is smaller than second then we do not swap the elements.

Step 4 :- Again second and third elements are compared and swapped if it is necessary and this process goes until last and second last element, is compared and swapped.

Step 5:- There are n elements to be sorted then the process mentioned above should be repeated n-1 to get the required result.

Step 6:- Stop the output and input of above algorithm of above labeled sort stepwise

38

```
print ("quick sort")
def quick (a list):
    help (a)
def help (a list, first, last):
    if first < last:
        split = part (a list, first, last)
        help (a list, first, split - 1)
        help (a list, split + 1, last)
def part (a list, first, last):
    pivot = a list [first]
    l = first + 1
    r = last
    done = False
    while not done:
        while l <= r and a list [l] <= pivot:
            l = l + 1
        while a list [r] >= pivot and r >= l:
            r = r - 1
        if r < l:
            done = True
        else:
            t = a list [l]
            a list [l] = a list [r]
            a list [r] = t
        n = len (a list)
    quick (a list)
    print (a list)
```


Quick sort

- Aim :- Implement quick sort to sort the given list.
- Theory :- The quick sort is a recursive algo based on the divide and conquer technique.
- Algorithm :-

Step 1 :- Quick sort first selects a value, which is called pivot value, first element serve as our first pivot value since we know that first will eventually end up as last in that list.

Step 2 :- The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than pivot value.

Step 3 :- Partitioning begins by locating two position marks let's call them left mark and right mark at the beginning and end of remaining items in the list.

Step 5:- At the point where rightmark becomes less than leftmark we stop the position of rightmark is now split point.

Step 6:- The pivot value can be exchanged with the contents of split point and now (pivot value is now in place.

Step 7:- The quick sort function passes invokes a recursive function quicksort.

Step 9:- quicksort begin with zero and last as range.

Step 10:- If length of list is less than equals to, it is already sorted.

Output:-

```
Enter range for list : 5
Enter element : 4
Enter element : 3
Enter element : 2
Enter element : 1
Enter elements : 9
[1, 2, 3, 4, 9]
```

```
OP
# stack
print ("Harish Chavan")
class stack :
    global tos
    def __init__(self):
        self.l = [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.l[self.tos] = 0
        self.tos = self.tos - 1
def peek(self):
    if self.tos < 0:
        print("stack empty")
    else:
        a = self.l[self.tos]
s = stack()
```


PRACTICAL NO:5

Aim :- Implementation of stack using python list

Theory :- A stack is a linear data structure that can be represented in the real world in the form of physical stack or a pile. The elements in the stack are added or removed only from one position i.e the topmost position.

Algorithm :-

Step 1 :- Create a class stack with instance variable items.

Step 2 :- Define the init method with self argument and initialize the initial value and then initialize to an empty list.

Step 3 :- Define method push and pop under the class stack.

Step 4:- Use the statement to give the condition that if length of given list is greater than the range of the list, stack is full.

Step 5:- Or else print statement as insert the element into the stack and initialize the value.

Step 6:- Push method used to insert the elements and pop method used to delete the element from the stack.

Step 7:- Assign the elements value in push method to add and print the given value is poped or not.

Output

Avish Chavan

```
>>> s.push(20)
>>> s.d
[20, 0, 0, 0, 0]
>>> s.pop()
Data = 20
>>> s.d
[0, 0, 0, 0]
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.push(50)
>>> s.pop()
>>> s.d
[10, 20, 30, 40, 50]
```

```
program :-
class queue :
    global r
    global f
    global a
    def __init__(self) :
        self.f = 0
        self.r = 0
        self.a = [0, 0, 0, 0, 0]
    def enqueue(self, value) :
        self.n = len(self.a)
        if (self.r == self.n) :
            print ("Queue is full")
        else :
            self.a[self.r] = value
            self.r += 1
            print("Queue element inserted", value)
    def dequeue(self) :
        if (self.f == len(self.a)) :
            print("Queue is empty")
        else :
            value = self.a[self.f]
            self.a[self.f] = 0
            print("Queue element deleted", value)
t: Queue() self.f += 1
```


PRACTICAL NO. 6

- Aim: Implementing a queue using python list
- Theory :- Queue is linear data structure which has 2 reference front and rear. Implementing a queue using python list is the simplest as the python list provides in built functions to perform the specified operations of the queue. It is based on the principal that a new element is inserted for rear and element of queue is deleted which is at front.
- Queue() : creates a new empty queue
- Enqueue(): Insert an element at the rear of the queue and similar to that of insertion of linked using stack.
- Dequeue(): Retrieves the element which was at the front. The front is moved to the successive element. It dequeue operation cannot remove element of the queue is empty.

Algorithm :-

Step 1: Define a class of queue and assign global variable then define init() method with self argument in init() assign or initialize the initial value with the help of self argument.

Step 2: Define a empty list and define enqueue() method with 2 argument, assign the length of empty list.

Step 3: Use if statement that length is equal to rear then queue is full or else insert the element in empty list and display that queue element added successfully and increment by 1.

Step 4: Define dequeue() with self-argument order this, use if statement that front is equal to length of list then display queue is empty or else, give that front is at zero and using that delete the increment from front side and increment is by 1.

Output :-

```
>>> b.enqueue(4)
  queue element inserted 4
>>> b.enqueue(5)
  queue element inserted 5
>>> b.enQueue(6)
  queue element inserted 6
>>> b.enqueue(7)
  queue element inserted 7
>>> b.enqueue(8)
  queue element inserted 8
>>> b.enqueue(9)
  queue is full
>>> b.o
  [4, 5, 6, 7, 8]
>>> b.dequeue()
  queue element deleted 4
>>> b.dequeue()
  queue element deleted 5
>>> b.dequeue()
  queue element element 6
>>> b.dequeue()
  queue is empty
>>> b.o
  [0, 0, 0, 0, 0]
```

Source code :-

```
def evaluate(s):
    k = s.split()
    n = len(k)
    stack = []
    for i in range(n):
        if n[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))
    return stack.pop()
s = "869 X +"
s = evaluate(s)
print("the value is:", s)
```


Aim :- program of evaluation of given string by using stack in python environment i.e postfix.

Theory :-

The post fix expression is free of any parentheses. Further we need not care of the priorities of the operators in the program. A given postfix expression can easily be evaluated using stack. Reading the expression is always as from left to right in postfix.

Algorithm :-

Step 1 :- Define evaluate as function, then create an empty stack in python.

Step 2 :- convert the string to a list by using the string method 'split'.

Step 3 :- Calculate the length of string and print it.

Step 4 :- Use for loop to assign the range of string then give condition using if statement.

Step 5 :- To perform the Arithmetic operation pop two elements from the stack and push the result back on the stack.

Step 6 :- Print the result of stack after the evaluation of postfix.

Step 7 :- Attach output and input of above algorithm.

Output :-
The evaluated value is : 62

```
Source code
# linked list
class node:
    global data
    global next
    def __init__(self, item):
        self.data = item
        self.next = None
class linked list:
    global s
    def __init__(self):
        self.s = None
    def addL(self, item):
        new node = node(item)
        if self.s = = none:
            self.s = new node
        else:
            head = self.s
            while head.next != none:
                head = head = next
                head.next = new node
    def display(self):
        head = self.s
        while head.next != None:
            print(head.data)
            head = head.next
        Print(head.data)
```


PRACTICAL-8

- Aim :- Implementation of single linked list by adding the nodes from last position.

- Theory :- A linked list is a linear data structure which stores the element in a node in a linear fashion but no necessarily contiguous. The individual element of the linked list called a node. Node consists of 2 parts. ① Data ② Next. Data stores all the information w.r.t the element for example roll number name, address, etc whereas next refers to the next node.

Algorithm :-

Step 1 :- Traversing of a linked list means visiting all the nodes in the linked list in order to perform some operation on them.

Step 2 :- The entire list can be accessed using the first node of linked list. The first Node of the linked list in turn is returned by the head pointer of linked list.

Step 3:- Thus the entire linked list can be traversed using the node which is referred by the head pointer of linked list.

Step 4:- Now that we know that we can traverse the entire linked list using the head pointer we should only use it to the first node of list only.

Step 5:- We should not use the head pointer to traverse the entire linked list. Because the head pointer is only reference the head pointer can lead to changes we cannot target.

Step 6:- We may lose the reference to the 1st node in our linked list and hence most of our linked list. So in order to avoid making some unwanted changes to the 1st node, we will use a temporary node to traverse the entire linked list.

Step 7:- We will use this temporary node as a copy of node we are currently traversing. Since we are using temporary node

Output :-

```
>>> q.addB(10)
>>> q.addB(20)
>>> q.addB(30)
>>> q.addB(40)
>>> q.addB(50)
>>> q.addB(60)
>>> q.addB(70)
>>> q.addB(90)
>>> q.display()
40
30
20
10
50
60
70
90
```

Binary tree

```
class node:
    def __init__(self, value):
        self.left = None
        self.val = value
        self.right = None

class BST:
    def __init__(self):
        self.root = None
    def add(self, value):
        p = node(value)
        if self.root == None:
            self.root = p
            print("Root is added successfully", p.val)
        else:
            h = self.root
            while True:
                if h.left == None:
                    h.left = p
                    print(p.val, "Node is added to left side successfully", h.val)
                    break
                else:
                    h = h.left
```


PRACTICAL - 9

Aim: Program based on binary search tree by implementing inorder, preorder, postorder.

Theory: Binary tree is a tree which supports maximum of 2 children for any node within the tree. Thus any particular node can have either 0 or 1 or 2 children. There is another identity of binary tree that is ordered such that one child is identified as left child and another as right child.

In order: (1) Traverse the left subtree. The left subtree in turn might have left and right subtrees.
2) Visit root nodes.
3) Traverse the right subtree and repeat it.

Preorder: 1) Visit the root node.
2) Traverse the left subtree. The left subtree in turn might have left and right subtrees.
3) Traverse the right subtree and repeat it.

Postorder: 1) Traverse the left subtree. The left subtree in turn might have left and right subtrees.
2) Traverse the right subtree.
3) Visit the root node.

Algorithm:

Step 1:- Define class node and define init() with self argument. Initialize the value with method.

Step 2:- Now define a class BST that is binary search tree with with init() with self argument and assign root is none

Step 3:- Define add() method for adding the node Define a variable p that p = node(value)

Step 4:- Use if statement for checking the condition that root is none then use else statement for if node is less than the main node then put in arrange that is leftside.

Step 5:- Use while loop for checking the node if is less than or greater than the main node and break the loop if it is not satisfying

Step 6:- Use if statement within that else statement for checking that Node is greater than main root then it into right side

Step 7:- After the left subtree and right subtree repeat the method to arrange the node according to binary search tree.


```
def Inorder(root)
    if root == None:
        return
    else:
        Inorder(root.left)
        print(root.val)
        Inorder(root.right)

def Preorder(root):
    if root == None:
        return
    else:
        print(root.val)
        preorder(root.left)
        preorder(root.right)

def Postorder(root):
    if root == None:
        return
    else:
        postorder(root.left)
        postorder(root.right)
        print(root.val)
```

output:

```
>>> t = BST()
>>> t.add(7)
(Root is added successfully : 7)
>>> t.add(5)
(Root is added successfully : 5)
>>> t.add(12)
(Root is added successfully : 12)
```

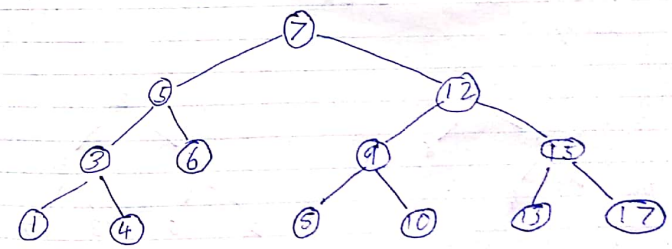
```
>>> t.add(6)
('Root is added successfully', 6)
>>> t.add(9)
('Root is added successfully', 9)
>>> t.add(15)
('Root is added successfully', 15)
>>> t.add(1)
('Root is added successfully', 1)
>>> t.add(4)
('Root is added successfully', 4)
>>> t.add(8)
('Root is added successfully', 8)
>>> t.add(10)
('Root is added successfully', 10)
>>> t.add(13)
('Root is added successfully', 13)
>>> t.add(17)
('Root is added successfully', 17)
```


Step 8:- Define Inorder(), Pre-order(), Post-order() with root argument and use if statement that root is none and return that in all.

Step 9:- In order else statement used for giving that condition first left, root and right node.

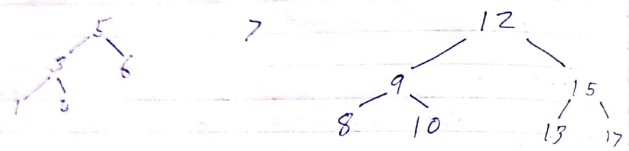
Step 10:- For Preorder we to give condition in else that first root, left and the right node.

Binary search tree

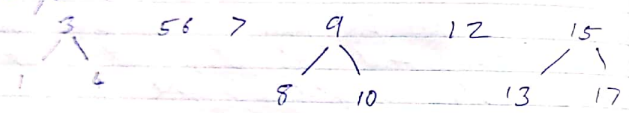


In order (LVR)

Step 1 :



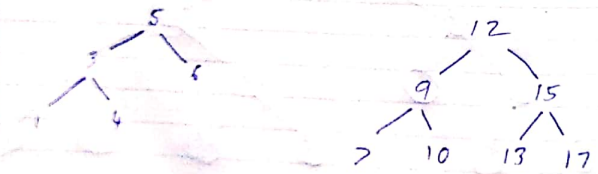
Step 2 :-



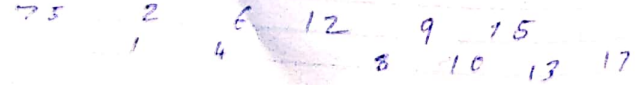
Step 3 :- 1 3 4 5 6 7 8 9 10 12 13 15 17

Pre order : VLR

Step 1 :



Step 2



>>> In order (root)

1
3
4
5
6
7
8
9
10
12
13
15
17

>>> Pre order (root)

7
5
3
1
4
6
12
9
8
10
15
13
17

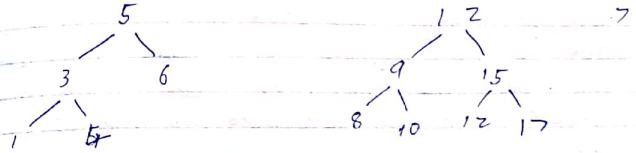
>>> Post order (L-root)

1
4
3
6
5
8
10
9
13
17
15
12
7

Step 3:- 7 5 3 1 4 6 12 9 8 10 15 13 17

Post order : (L R V)

step 1 :-



step 2 :- 3 6 5 9 15 12 7
1 4 8 10 13 17

Step 3 :-
1 4 3 6 5 8 10 9 13 17 15 12 7

PRACTICAL - 10

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 we are actually moving the front of the queue forward, thereby reducing the overall size of the queue and we cannot insert new elements.

1) Computer controlled traffic signal system uses circular queue

2) CPU scheduling and memory management.


```
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.f < 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.l[self.f] = 0
            self.f = self.f + 1
        else :
            print("Queue is empty")
            self.f = 5
```

```
else:
    s = self.f
    self.f = 0
    if R.(self.f (self.r):
            print (self.l[self.f])
            self.f = self.f + 1
        else:
            print ("queue is empty")
            self.f = s
q = Queue e()
```

Output:

```
>>> q.add (100)
    Data added : 100
>>> q.add (200)
    Data added : 200
>>> q.add (300)
    Data added : 300
>>> q.removed()
    Data removed : 100
>>> q
    [0, 200, 300, 0, 0, 0]
```

PRACTICAL NO: 11

Aim :- To sort the list using merge sort

Theory :- Like quicksort, merge sort is a divide and conquer algorithm. It divides input array in the halves and then merges the two 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 merges the two halves till the size becomes 1 and the size becomes 1, the merge process comes into action and starts merging back till the complete array is merged

Applications :-
Merge sort is useful for sorting linked list in O(n log n) time. merge sort accesses data sequentially and the need of random access is low
Inversion count problem
used in external sorting.
Merge sort is more efficient than quicksort some types of list if the data to be sorted can only be efficiently accessed sequentially.

Code :-

```
def merge sort (arr):
    if len (arr) >> 1:
        mid = len (arr) // 2
        left half = arr[:mid]
        right half = arr [mid:]
        mergesort = (left half)
        merge sort (right half)
        i = j = k = 0
        while (i < len (left half) and j < (right half)):
        if left half [i] < right half [j]:
            arr [i < ] = left half [i]
            i = i + 1
        else:
            arr [i < ] = right half [j]
            j = j + 1
        k = k + 1
    while (j < len (left half)):
            arr [i < ] = right half [j]
            j = j + 1
            k = k + 1
arr = [27, 89, 70, 55, 62, 99, 45, 14, 10]
print ("Random list: ", arr]
merge sort (arr)
print ("In merge sorted list: ", arr)
```

Output :
Random list : [27, 89, 70, 55, 62, 99, 45, 14, 10]
merge sorted list: [10, 14, 27, 45, 62, 70, 89, 90]