☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
1.	Implement linear search to find an item in the list.	33	29/11/19	29/11/19
2.	Implement Binary search to sort & find an item in the list	36	6/12/19	20/12/19
3.	Implementation of bubble sort program on given list	38	6/12/19	
4.	Implementation of quick sort to sort the list	41	20/12/19	
5.	Implementation of stacks using Python list	42	3/1/20	03/01/2020
6.	Implementing a Queue using list	43	10/1/20	10/01/2020
7.	Evaluation of Postfix Expression	46	17/1/20	17/01/2020
8.	Implementation of single linked list	48	24/1/20	24/01/2020
9.	Program based on Binary Search Tree	51	07/02/20	07/02/2020

INDEX

No.	Title	Page No.	Date	Staff Member's Signature
10.	To sort a list using mergesort	54	14/2/20	14/02/2020
11.	To demonstrate the use of circular queue.	56	14/2/20	14/02/2020

UNSORTED PROGRAM :-

```
a= list (input ("Enter any list of numbers"))
b= int (input (" Enter a number to be searched "))
for i in range (len (a)):
    if (b == a[i]):
        print " Number searched is found at position", i
        break
if (b != a[i]):
    print " Number not found "
```

Output :-

Enter any list of number 18, 22, 11, 17, 63
Enter a number to be searched : 17
Number searched is found at position 3

PRACTICAL - 1

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

Theory :-

LINEAR SEARCH

Linear search is one of the simplest searching 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 in case of an ordered list, instead of searching the list in sequence. A binary search is used which will start by examining the middle term.

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

1] **UNSORTED :-**

ALGORITHM :-

Step 1 :- Create an empty list & assign it to a variable.

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

Step 3 :- Use for loop for adding the elements into the list.

Step 4 :- Find the new list.

Step 5 :- Accept an element from the user that to be searched in the list.

Step 6 :- Use for loop in a range from '0' to the total no. of elements to be search the element from the list.

Step 7 :- Use if loop that the elements in the list is equal to the element accepted from user.

Step 8 :- If the element is found then print the statement that the element is found along with the element's position.

Step 9 :- Use another if loop to print that the element is not found if the element which is accepted from user is not their in the list.

Step 10 :- Draw the output of given algorithm.

2] SORTED :-

ALGORITHM :-

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

Step 2 :- Accept total no. of element to be inserted into the elements in the list.

SORTED PROGRAM :-

```
a = list (input ("Enter any list of numbers : "))
a.sort ()
print a
b = int (input (" Enter a number to be searched : "))
for i in range (len (a)) :
    if (b == a[i]) :
        print " Number is found at position", i
        break
if (b != a[i]) :
    print " Number not found "
```

Output :-

Enter any list of number 22, 11, 36, 63, 54

a = [11, 22, 36, 54, 63]

Enter a number to be searched : 36

Number is found at position 2

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

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

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

Step 6 :- Then use else statement if element is not found in range then satisfy the given condition.

Step 7 :- Use for loop in range from 0 to the total no. of elements to be searched before doing this accept an search no from user using input statement.

Step 8 :- Use if loop that the elements in the list is equal to the element accept from user.

Step 9 :- If the element is found then print the statement that the element is found along with the element position.

24/11/22

Step 10 :- Use another if loop to print that the element is not found if the element which is accepted from the user is not in the list. Attach the input & output of above algorithm.

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 search is a search algorithm that finds the 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 list in linear search which is time consuming. This can be avoided by using binary search.

* **Algorithm:-**

Step 1:- Create Empty list & 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() method to sort the accepted element and assign it in increasing ordered list. print the list after sorting.

Source code :-

```
# Binary Search
A=[12,14,19,27,32,36,39,40,52]
search=input("Enter
a = list (input ("Enter the list to be sorted : "))
b = int (input ("Enter the number to be searched : "))
a.sort()
print (a)
l = 0
h = len (a) - 1
while (h >= l ):
    mid = (l + h) / 2
    if (b < a[l] or b > a[h]):
        print "Number not found"
        break
    elif (b == a[mid]):
        print "Number is found at position, mid
        break
    elif (b <= a[mid]):
        h = mid - 1
    elif (b >= a[mid]):
        l = mid + 1
```

Output :-

Enter the list to be sorted : 88, 77, 66, 55, 44, 33.
Enter the number to be searched : 33.
[33, 44, 55, 66, 77, 88]
Number is found at position 0.

Enter the list to be sorted : 99, 77, 66, 44
Enter the number to be searched : 55
[44, 66, 77, 99]
Number not found.

Step5:- Use if loop to give the range in which element is found in given range then display a message "Element not found".

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

Step 7:- Accept an arguement & key of the element that element has to searched.

Step 8:- Initialize first to 0 & last to last element of the list. As array is starting from 0 hence it is initialized 1 less than the total count.

Step 9:- Use for loop & assign the given range.

Step 10:- If statement in list and still the element to the searched is not found then find the middle element (m)

Step 11:- Else if the item to be searched is still less than the middle item then
Initialise last (h) = mid (m) - 1
Else
Initialise first (l) = mid (m) + 1

Step 12:- Repeat till you found the element. Stick the input & output of above algorithm.

PRACTICAL -3

* **Aim :-** Implementation of bubble sort program on given list.

* **Theory :-**

BUBBLE SORT

Bubblesort is based on the idea of repeatedly comparing pairs of adjacent elements & then swapping their position if they exist in the wrong order. This is the simplest form of sorting available in ascending or descending order by comparing two adjacent elements at a time.

* **Algorithm :-**

Step 1 :- Start by comparing the first two elements of an array & swapping if necessary.

Step 2 :- Sort the element in ascending order. then swap the elements as required.

Step 3 :- If condition is already ascending then don't swap the elements.

Step 4 :- Same process will continue till the last element

Source Code:-

```
a = list (input ("Enter the list to be sorted :"))
for pass in range (len (a)-1):
    for comparision in range (len (a) - 1 - pass):
        if (a[comparision] > a[comparision + 1]):
            t = a [comparision]
            a[comparision] = a[comparision +1]
            a[comparision +1] = t
print (a)
```

Output :-

[3, 6, 9, 10, 13, 16, 20, 35]

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

Step 6:- Print the output & input of the above algorithm

MB

PRACTICAL 4

* **Aim:-** Implement Quick sort to sort the given list.

* **Theory:-** The quick sort is a recursive algorithm 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 & 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 markers lets' call them leftmark & rightmark. at the beggining and end of remaining items in the list. The goal of the partition process is to move items that are on wrong side w.r.t pivot value while also converging on the split point.

Step 4:- We begin by incrementing leftmark until we locate a value that is greater than the p.v. We then decrement right mark until we find value that is less than the p.v. At this point we have discovered two items that

Source Code :-

```
def quick(alist):
    help(alist, 0, len(alist)-1)
def help(alist, first, last):
    if first < last:
        split = part(alist, first, last)
        help(alist, first, split-1)
        help(alist, split+1, last)
def part(alist, first, last):
    pivot = alist[first]
    l = first+1
    r = last
    done = False
    while not Done:
        while l<=r & alist[l] <= pivot:
            l = l+1
        while alist[r] >= pivot & r>=l:
            r = r-1
        if r<l:
            done = True
        else:
            t = alist[l]
            alist[l] = alist[r]
            alist[r] = t
    t = alist[first]
    alist[first] = alist[r]
    alist[r] = t
    return r
x = int(input("Enter range for the list : "))
alist = []
```

```
for b in range (0,x):
        b = int (input ("Enter element : "))
        alist . append (b)
        n = len (alist)
quick (alist)
print (alist)
```

Output :-

```
Enter the range of the list : 4
Enter elements in the list : 3
Enter elements in the list : 7
Enter elements in the list : 2
Enter elements in the list : 1
```

29/12/17

are out of place with respect to eventual split point.

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

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

Step 7:- In addn, all the items to left of split point are less than p.v. & all the items to the right of split point are greater than p.v. The line can now be divided at split point & quick sort can be invoked recursively.

Step 8:- The quicksort func^n invokes a recursion func^n quicksorthelper.

Step 9:- quicksorthelper begins with same base as the merge sort.

Step 10:- If length of the list is less than equal to 0, it is already sorted.

Step 11:- If it is greater than it can be partitioned and recursively ~~sort~~ sorted.

Step 12:- Display & stick the code & output of above algorithm.

Practical - 5

* **Aim:-** Implementation of stacks using Python list

* **Theory:-** A stack is a linear data structure that can be represented in the real world in the form of a physical stack or a pile. The elements in the stack are added or removed only from one position i.e. the topmost position. Thus, the stack works on the LIFO (Last In First Out) principle as the element that was inserted last will be removed first. A stack can be implemented using array as well as linked list. Stack has three basic operation: push, pop, ~~peek~~ returns the topmost element of the stack. The operations of adding and removing the elements is known as push & pop.

* **Algorithm:-**

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

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

Step 3:- Define methods push & pop under the class stack.

Step 4:- Use the if statement to give the cond" that if length of given list is greater than the range of list then print stack is full.

~~Output~~ Code :-

```
print "Shubham Uniyal "
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 is empty"
        else:
            k = self.l[self.tos]
            print "Data =", k
        self.l[self.tos] = 0
        self.tos = self.tos - 1
~~s = stack()~~
    def peek(self):
        if self.tos < 0 :
            print "Stack is empty"
        else:
            a = self.l[self.tos]
        print "Data =", a
s = stack()
```

MR
03/01/2020

Output :-

```
Shubham Uniyal
Data = 50
>>> s.l
[10,20,30,40,50]
>>> s.pop()
>>> s.pop()
>>> s.l
[10,20,30,0,0]
>>> s.pop()
>>> s.pop()
>>> s.pop()
>>> s.l
[0,0,0,0,0]
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.push(50)
>>> s.l
[10,20,30,40,50]
>>> s.pop()
>>> s.peek()
Data = 40
```

mm 03/01/2020

Step 5:- Or Else print statement as insert the element into the stack & initialize the values.

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

Step 7:- If in pop method, value is less than 1 then return the stack is empty or else delete the element from stack at topmost position.

Step 8:- First condition checks whether the no. of elements are zero while the second case whether top is assinged any value. If top is not assigned any value, then we can be sure that stack is empty.

Step 9:- Assign the element values in push method to print given value is poped or not.

Step 10:- Attach the input & output of above algorithm.

m

PRACTICAL -6

* **Aim :-** Implementing a Queue using list.

* **Theory :-** Queue is a linear data structure which has 2 references front & rear. Implementing a queue using Python list is the simplest as the python list provides inbuilt functions to perform the specified operations of the queue. It is based on the principal that a new element is inserted after rear & element of queue is deleted which is at front. In simple terms a queue can be described as a data structure based on first in first out FIFO principle.

* **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 tail :-

* **Dequeue ()** :- Returns the element which was at the front the front is moved to the successive element. A dequeue operation cannot remove element if the queue is empty.

* **Algorithm :-**

Step 1 :- Define a class Queue & assign global variable the define init () method with self argument in init ()

Code :-

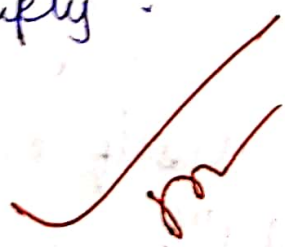
```
class Queue:
    global f, r, a
    def __init__(self):
        self.f = 0
        self.r = 0
        self.a = [0,0,0]
    def enQueue(self, value):
        self.n = len(self.a)
        if (self.r == self.n):
            Print ("Queue is full")
        else:
            self.a[self.r] = value
            self.r += 1
            Print ("Insert :", 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)
            self.f += 1
b = Queue()
```

10/01/2022

Output :

```
>>>b.enQueue(2)
Insert: 2
>>>b.enQueue(4)
Insert: 4
>>>b.enQueue(7)
Insert: 7
>>>b.a
[2,4,7]
>>> b.deQueue()
Queue element deleted: 2
>>>b.deQueue()
Queue element deleted: 4
>>> b.deQueue()
Queue element deleted: 7
>>> b.deQueue()
Queue is empty.
>>>b.a
[0,0,0]
```

```
>>> b.enQueue(8)
Queue is full
```



assign or initialize the initial value with the help of self argument.

Step 2:- Define an empty list & define enQueue() method with 2 argument. Assign the length of empty list

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

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

Step 5:- Now call the Queue() function & give the element that has to be added in the empty list by using enQueue() & print the list after adding and same for deleting and display the list after deleting the element from the list.

10/01/24

PRACTICAL - 7

* **Aim :-** Program an evaluation of given string by using stack in Python Environment i.e. Postfix.

* **Theory :-**

The postfix expression is free of any parathesis further use took care of the priorities of the operators in the program. A given postfix expression can easily be evaluated using stacks. Reading the expression is always from left to right in Postfix.

* **Algorithm :-**

Step 1 :- Define evaluate as function as function then create a empty stack called in Python.

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

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

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

Step 5 :- Scan the token list from left to right. If token is an operand, convert it from a string to an integer & push the value onto the 'p'.

if self.st == None:

Code :-

```
def evaluate(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))
        elif k[i] == '/':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
    return stack.pop()
s = "2 5 5 + *"
r = evaluate(s)
print("The evaluated value is :", r)
```

* Output :-

```
The evaluated value is : 20
```

Step 6:- If the token is an operator *, /, +, -, ^, it will need two operands. Pop the 'p' twice. The first pop is second operand & the second pop is the first operand.

Step 7:- Perform the Arithmetic operation. Push the result back on the 'm'.

Step 8:- When the input expression has been completely processed the result is on the stack. Pop the 'p' & return the value.

Step 9:- Print the result of string after the evaluation of postfix.

Step 10:- Attach the output & input of above algorithm.

17/01/2021

Practical - 8

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

* **Theory:-** A linked list is a linear data structure which stores the elements in a node in a linear fashion, but no necessarily continuous. The individual element of the linked list called a Node. Node comprises of 2 parts. ① Data ② Next. Data stores all the information w.r.t the element, for example roll no, name, address, etc, whereas next refers to the next node. In case of larger list, if we add/remove any element from the list, all the elements of list has to adjust itself. every time we add, it is very tedious task & linked list is used to solving this type of problem.

* **Algorithm:-**

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

Step 2:- The entire linked list means can be accessed use the first node of the linked list. The first node of the linked list in turn is referred by the Head pointer of the linked list.

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

Code :-

```
class node:
    global data, next
    def __init__(self, item):
        self.data = item
        self.next = None

class linkedlist:
    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
    def display(self):
        if self.s == None:
            print
        head = self.s
        while head.next != None:
            print(head.data)
            head = head.next
        print(head.data)
q = linkedlist()
```

Output :-

```
>>> q.addL(40)
>>> q.addL(30)
>>> q.addL(20)
>>> q.addL(10)
>>> q.addB(60)
>>> q.addB(70)
>>> q.addB(80)
>>> q.addB(90)
>>> q.display()
90
80
70
60
40
30
20
10
```

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

Step 5:- We should not use the head pointer to transverse the entired linked list because the head pointer is our only reference to the 1st node in the linked list, modifying the reference of the head pointer can lead to changes which we can't reverse back.

Step 6:- We may lose the reference to the 1st node in our linked list & 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 transverse the entire linked list.

Step 7:- We will use this temporary node as a copy of the node we are currently transversing. Since we are making temporary node a copy of current node the datatype of the temporary node should also be node.

Step 8:- Now that current is referring to the first node, if we want to access 2nd node of list we can refer it as the next node of the 1st node.

Step 9:- But the 1st node is referred by current. So we can transverse to 2nd nodes as h = h.next

Step 10:- Similarly, we can transverse rest of nodes in the linked list using same method by while loop.

Step 11:- Our concern now is to find terminating condition for the while loop.

Step 12:- The last node in the linked list is referred by tail of linked list. Since the last node of linked list does not have any next node, the value in the next field of the last node is None.

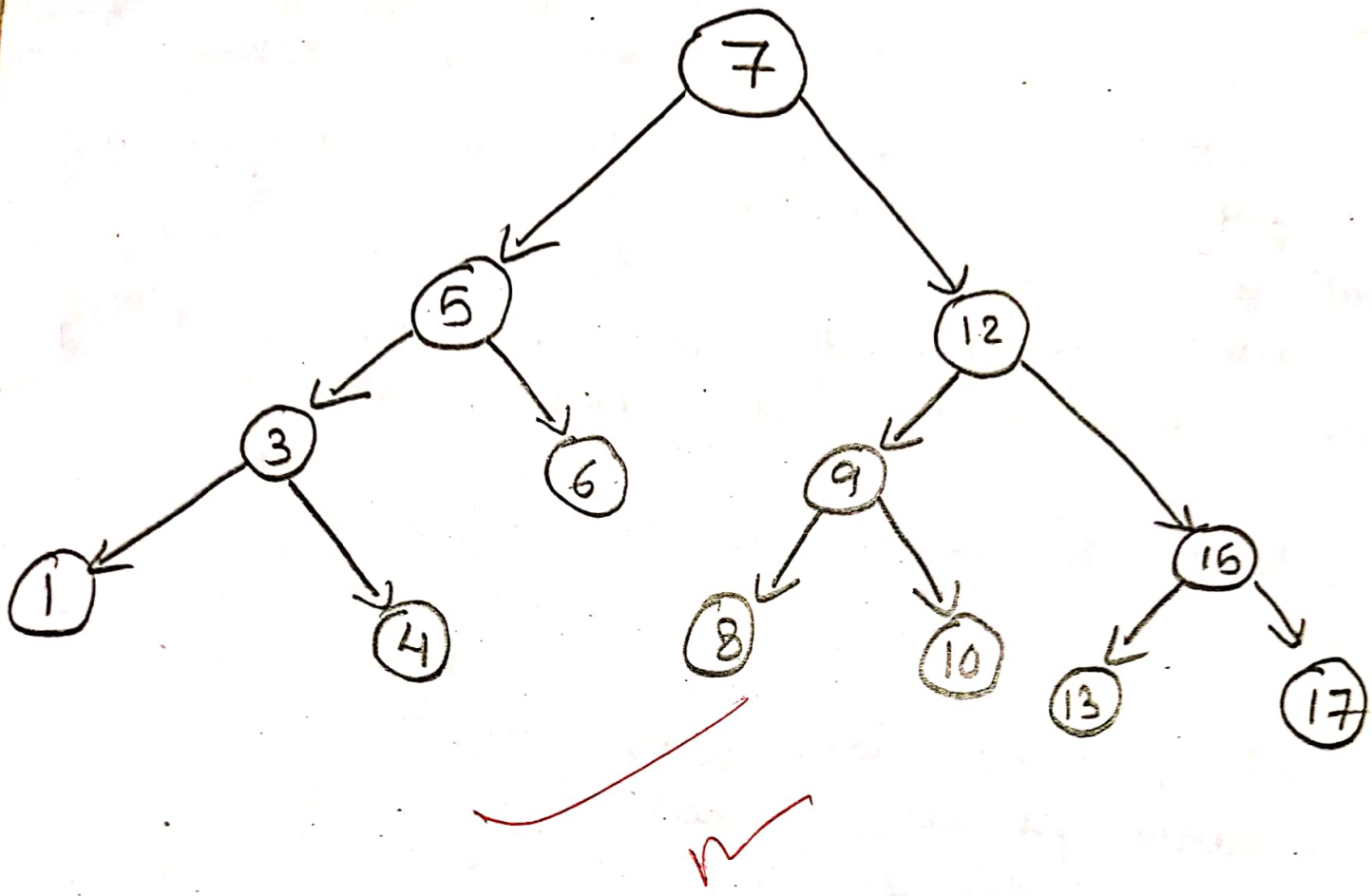
Step 13:- To we can refer the last node of linked list as self.next = None

Step 14:- We have to now see how to start traversing the linked list & how to identify whether we have to reach the last node of linked list or not.

Step 15:- Attach the code & output of above algorithm

24/01/22

Binary Search Tree



Practical - 9

* Aim :- Program based on Binary search tree by implementing Inorder, Preorder & Postorder Transversal.

* 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 it is ordered such that one child is identified as left child & other as right child.

* Inorder : (i) Transverse the left subtree. The left subtree within might have left & right subtrees.
(ii) Visit the root node.
(iii) Transverse the right subtree & repeat it.

* Preorder :- (i) Visit the root node
(ii) Transverse the left subtree. The left subtree within might have left & right subtree.
(iii) Traverse the right subtree & repeat it.

* Postorder :- (i) Transverse the left subtree. The left subtree within might have left & right subtrees
(ii) Transverse the right subtrees.
(iii) Visit the root node

* Algorithm :-

Step 1 :- Define class node & define init () method with 2 argument. Initialise the value in this method.

Step 2 :- Again, Define a class BST that is Binary Search Tree with init () method with self argument & assign the 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 cond^n that root is none then use else statement for if node is less than the main node then put or arrange that in leftside.

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

Step 6 :- Use if statement within that use statement for checking that node is greater than main root then put it into rightside.

Step 7 :- After this, left subtree & right subtree, repeat this method to arrange the node according to Binary Search Tree.

Code:-

```
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 p.val < h.val:
                  if h.left == None:
                      h.left = p
                      print (p.val, "Node is added to leftside successfully at ", h.val)
                      break
                  else:
                      h = h.left
              else:
                  if h.right == None:
                      h.right = p
                      print (p.val, "Node is added to rightside successfully at ", h.val)
                      break
                  else:
                      h = h.right
```

```
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)
t = BST()
```

Step 8:- Define Inorder(), Preorder() & Postorder() with root argument & use if statement that root is None & return that in all.

Step 9:- In Inorder, else statement used for giving that condn first left, root & then right node.

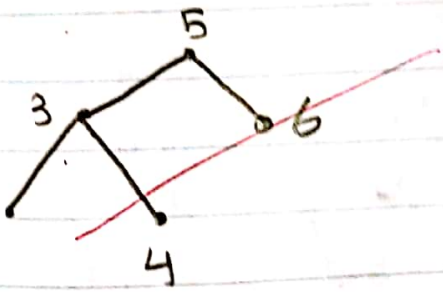
Step 10:- For Preorder, we have to give condn in else that first root, left & then right node.

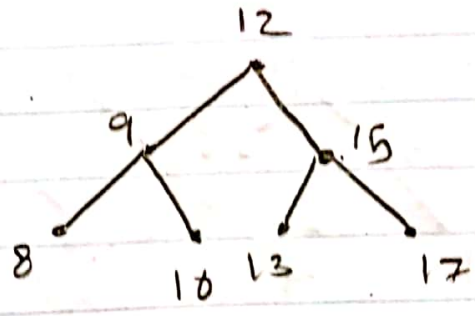
Step 11:- For Postorder. In else part, assign left than right & then go for root node.

Step 12:- Display the output & input of above Algorithm.

* Inorder :- (LVR)

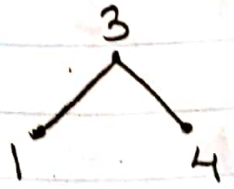
Step 1:



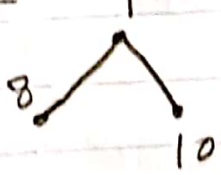


7

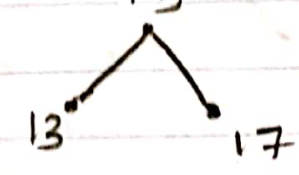
Step 2:



5 6 7



12

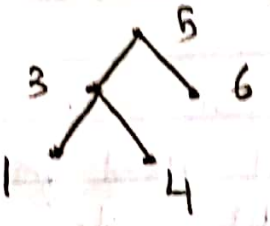


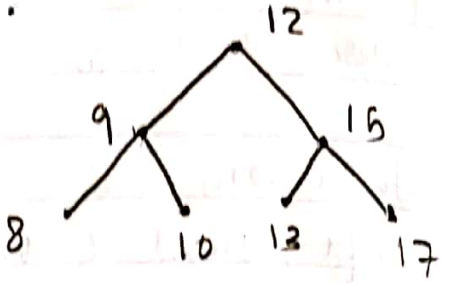
Step 3:

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

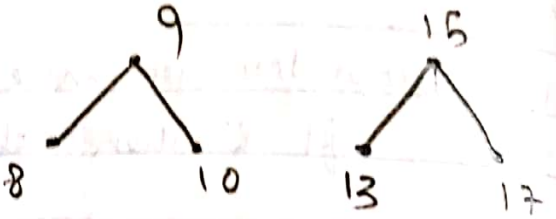
* Preorder : (VLR)

Step 1 :- 7





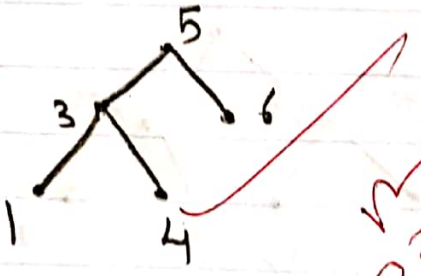
Step 2 :- 7 5 3 6 12

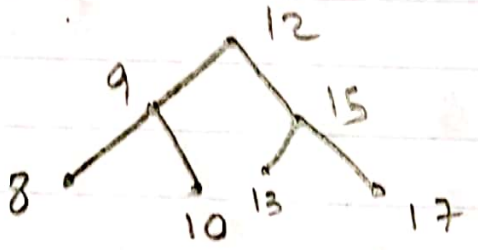


Step 3 :- 7 5 3 1 4 6 12 9 8 10 15 13 17

* Postorder :- (LRV)

Step 1 :-

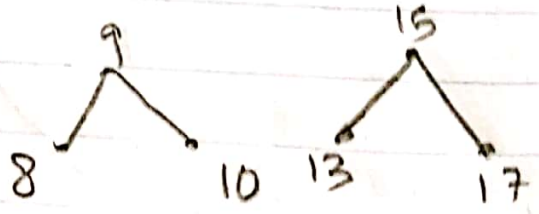




7

Step 2 :-

6 5



12 7

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

PRACTICAL - 10

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 mergeSort() 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 & 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 & starts merging arrays back till the complete array is merged.

* Applications :-
i) Merge Sort is useful for sorting linked lists in O(n logn) tion.
Merge sort accesses data sequentially & the need of random access is low.
ii) Inversion Count Problem.
iii) 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 is popular where sequentially accessed data structures are very common.

Code :-

```
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
```

Output :-

```
arr = [27,89,70,55,62]
print "RANDOM LIST", arr
RANDOM LIST : [27,89,70,55,62]
mergeSort (arr)
print "MERGESORTED LIST : ", arr
MERGESORTED LIST : [27,55,62,70,89]
```

rm
14/02/2020

Code :-

```
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.l[self.f] = 0
            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()
```

Output :-

```
q.l
[0,0,0,0,0,0]
q.add(44)
Data added : 44
q.add(55)
Data added : 55
q.add(66)
Data added : 66
q.l
[44,55,66,0,0,0]
q.remove()
Data removed : 44
q.remove()
Data removed : 55
q.remove()
Data removed : 66
q.remove()
Queue is empty
```

PRACTICAL - 11

Aim:- To demonstrate the use of circular queue.

Theory:- In a linear queue, 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, ~~we are~~ 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 of 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 & 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 :

i) Computer controlled traffic signal system uses circular queue.

ii) CPU scheduling & Memory Management.

MV
14/02/2020

```python
#FY 1786
class queue():
    def __init__ (self):
        self.q=list()

    def isEmpty(self):
        if self.length() is 0:
            return True
        else:
            return False

    def Enqueue(self,value):
        self.q.append(val)

    def length(self):
        return len(self.q)

    def Dequeue(self):
        if self.isEmpty():
            return None
        else:
            val=self.q.pop(0)
            return val

    def display(self):
        if self.isEmpty():
            print("Queue is empty")
        else:
            print(self.q)
```

```
q=queue()

choice=int(input("1.Enqueue \n 2.Dequeue \n 3.Length of Queue \n 4.Check Queue is empty or not \n 5.Display Queue"))
while choice is not 6:
  if choice is 1:
    val=int(input("Enter element to be inserted"))
    q.Enqueue(val)

  if choice is 2:
    val=q.Dequeue()
    if val is None:
      print("Queue is empty")
    else:
      print("The removed element is",val)

  if choice is 3:
    l=q.length()
    print("The length of the Queue is",l)

  if choice is 4:
    if(q.isEmpty() is True):
      print("Queue is empty")
    else:
      print("Queue is not empty")

  if choice is 5:
    q.display()

  choice=int(input("Enter your choice "))
```