Practical - 1

A

Aim :- To Search a number from the list using linear unsorted

Algorithms

Steps

1. The data is entered in random manner
2. User needs to Specify the element to be searched in the entered list.
3. Check the Condition that whether the entered number matches if it matches then display the location & increment by 1 as data is stored from location zero.
4. If all elements are checked one by one & no element not found then prompt message number not found.

Theory :-

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 incase of an ordered list, instead of searching the list in sequence A binary search is used which will start by examing 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 return that element found and its position is also found

Source Code

```
s=list(input("enter the element"))
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 the position ",i
        break
else:
    print "number not found"
```

Output

enter the element 4, 9, 7, 24, 8, 1

[1, 4, 7, 8, 9, 24]

Enter the number to be search 7

number found! in the position, 2

Output

enter the element 4, 9, 7, 24, 8, 1

[1, 4, 7, 8, 9, 24]

Enter the number to be search 3

number not found

Scanned with CamScanner

Unsorted Source Code

```
s= int(input("Enter the required number"))
a=[10,12,9,14,17,9]
for i in range(len(a)):
        if (a[i]==s):
            print("Required number found in position",i)
            break
if(s!=a[i]):
        print("The required number not found")
```

Output
Enter the required number 14
Required number found in position 3

Output
Enter the required number 4
The required number not found

B

Aim :- To Search a number from the list using linear sorted method.

Algorithm

Steps
1. The user is supposed to enter data in sorted manner.
2. User has to given an element for searching through sorted list.
3. If element is found display with an updation as value is stored from location 0.
4. If data or element not found print the same
5. In sorted order list of elements we can check the conditions that whether the entered number lies from starting point till the last element if not then without any processing we can say number not in the list

~~Sort~~ Unsort

1. Accept an integer value from the user
2. Define an array.
3. Using for loop to iterate the given length of an array.
4. Use if conditional statement to check whether the given number is present in an array or not.
5. If the if condition satisfy then display the position of required number.
6. If not then display the required number not found.

Sort

1. Accept a list input from the user.
2. Use sort function for sorting the given list data
3. Display sorted data
4. Accept an int data from the user that to be searched
5. Using for loop to iterate the given length of an array.
6. Use if conditional statement to check whether the given number is present in an array or not.
7. If the if condition satisfy then display the position of required number.
8. If not then display the required number not found.

MB
29/11/19

Scanned with CamScanner

Source Code

```
a = list(input("enter list : "))
a.sort()
c = len(a)
s = int(input("enter search no: "))
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
```

Output

Enter list : 8, 5, 9, 2, 1

[1, 2, 5, 8, 9]

enter search no: 9

(found at ', 2)

Practical - 2

Aim: Implement Binary Search to find an searched number in the list.

Theory:

Binary Search is also known as Half interval search, logarithmic search or binary chop 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 Fashion Search.

Ag Algorithm:

Steps

1. Create Empty list & assign it to a variable
2. Using Input method accept the range of given list
3. Use for loop, add elements in list using append() method.
4. Use sort() method to sort the accepted element & assign it in increasing ordered list. print the list after sorting.

Scanned with CamScanner

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

6. Then use else statement, if statements is not found in range then satisfy the below condition.

7 Accept an arguement & key of the element that element has to be searched

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.

9. Use for loop & assign the given arrange.

10. If statement in list & till the element to be searched is not found then find the middle element (m)

11. Else if the item to be searched is still less than the middle term then
initialize last (h) = mid (m) - 1
Else
Initialize first (l) = mid (m) - 1

12. Repeat till you found the element stick the input & output of above algorithm.

Scanned with CamScanner

Source Code

```
a = list (input(" Enter the element"))
for i in range (0, lin (a)):
    for j in range (0, lin(a)-1):
        if a[j] > a [j+1]:
            a[j] , a[j+1] = a [j+1], a[j]

print a
```

Output

Enter the element : 8, 9, 6, 3, 7

[3, 6, 7, 8, 9]

Practical - 3

Aim :- Implementation of Bubble Sort program on given list.

Theory:- Bubble Sort 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 simplest form of sorting available. In this, we sort the given elements in ascending or descending order by Comparing two adjacent elements at a time.

Algorithm

Steps

1. Bubble Sort algorithm start by Comparing the first two element of an array & Swapping if necessary.

2. If we want to sort the elements of array in ascending order then first element is greater than second then, we need to Swap the element.

3. If the element is Smaller than Second then we do not Swap the element.

4. Again Second & third elements are Compared & Swapped if it is necessary & this process go on until last & second last element is Compared and Swapped.

3. There are n elements in the listed from the process ~~[illegible]~~ mentioned above should be repeated n-1 to get the required result

4. Stick the Output 2 input of above algorithm of bubble sort Repeat

Scanned with CamScanner

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 and alist[l] <= pivot:
            l = l+1
        while alist[r] >= pivot and 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
```

Practical - 4

Aim: Implement Quick Sort to sort the given list

Theory: The quick sort is a Recursive algorithm based on the divide & Conquer technique.

Algorithm:

Steps

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

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.

3. Partitioning begins by locating two position markers - let's call them leftmark & rightmark. at the beginning and end of remaining items in the list. The goal of the partition process is to move items that are on wrong side with respect to pivot value while also converging on the split point.

4. We begin by incrementing leftmark until we locate a value that is greater than the P.V we then decrement rightmark until we find value that is less than the pivot values. At this point we have discovered two items that are out of place with respect to eventual split point.
5. At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point.
6. The pivot value can be exchanged with the content of split point & P.V (pivot value) is now in place.
7. In addition, 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 PV. The list can now be divided at split point & quick sort can be invoked recursively on the two halves.
8. The quicksort funct^n invokes a recursive funct^n, quicksorthelper.
9. quicksorthelper begins with same base as the merge sort.
10. If length of the list is less than equal one, it is already sorted.
11. If it is greater, then it can be partitioned and recursively sorted.
12. The partition funct^n, implements the process described earlier.
13. Display & stick the coding & output of above algorithm.


```
x= int (input ("Enter range for 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 for list: 5
Enter element: 4
Enter element : 2
Enter element = 3
Enter element = 8
Enter element = 1
1 2 3 4 8
```

20/12/19

Scanned with CamScanner

Source Code.

```
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
    def peek(self):
        if self.tos<0:
            print("stack empty")
        else:
            a= self.l[self.tos]
            print("data=",a)
s= Stack()
```

Output
s.stack()

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 canbe implemented using array as well as linked list. Stack has three basic operation: push, pop, peek. The Operations of adding & removing the elements is known as Push & Pop.

Algorithm:-

1. Create a class stack with instance variable items
2. define the init method with self argument & initialize the initial value and then initialize to an empty list.
3. Define methods push and pop under the class stack

Scanned with CamScanner

4. Use if statement to give the condition that if length of given list is greater than the range of list then print stack is full.

5. Or Else print statement as insert the element into the stack & initialize the value

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

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

8. First condition checks whether the no of elements are zero while the second case whether top is assigned any value. if ~~top~~ top is not assigned any value, then can be sure that stack is empty.

9. Assign the element values in push method to on & print the given value is popped not.

10. Attach the input & output of above algorithm.

Output

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

s.peek()

Scanned with CamScanner

Source Code:

```
class Queue:
    global r
    global f
    global a
    def __init__(self):
        self.r = 0
        self.f = 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.rt = 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)
            self.f = 1

b = Queue()
```

Pratical - 6

Aim:- Implementing a Queue Using python List.

Theory:- Queue is a linear data structure which has 2 reference front & rear. Implementing a queue using python list is the simplest as the python list provides in built functions to perform the specified operation of the queue. It is based on the principal that a new element of is inserted after rear & element of queue is deleted which is at front. In simple terms, a queue can be described a data structure based on ~~first~~ first in first Out FIFO Principle

Queue (): Create a new empty queue

Enqueue(): Insert an element at the rear of the queue & 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.

Scanned with CamScanner

Steps Algorithm:

1. Define a class Queue and assign global Variables then define init() method with self argument in init() assign or initialize the initial Value with the help of self argument

2. Define a empty list & define enqueue() method with 2 argument. assign the length of empty list

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 & increment by 1.

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 element from front side & increment it by 1

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 & same for deleting & display the list after deleting the element from the list.

Output

```
>>> b.enqueue(4)
('Queue element inserted: ',4)
>>> b.enqueue(8)
('Queue element inserted: ',8)
>>> b.enqueue(47)
('Queue element inserted: ',7)
>>> b.enqueue(2)
('Queue element inserted: ',2)
>>> b.enqueue(1)
('Queue element inserted: ',1)
>>> b.enqueue(5)
Queue is full
>>> b.dequeue()
('Queue element deleted', 4)
>>> b.dequeue()
('Queue element deleted ', 8)
>>> b.dequeue()
('Queue element deleted ', 7)
>>> b.dequeue()
('Queue element deleted ', 2)
>>> b.dequeue()
('Queue element deleted ', 1)
>>> b.dequeue()
Queue is empty
>>> print (b.a)
[0, 0, 0, 0, 0]
```

Source 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))
        else:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
    return stack.pop()

s = "5 8 3 - +"
r = evaluate(s)
print("Evaluated value is :", r)
```

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 parenthesis. Further we look 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:

1. Define evaluate as function then create a empty stack in Python.
2. Convert the string to a list by using the string method split
3. Calculate the length of string & print it.
4. Use for loop to assign the range of string then give condition using if statement
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'.
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.

Scanned with CamScanner

7. Perform the Arithmetic operation. Push the result back on the 'n'

8. When the input expression has been Completely processed, the result is on the stack. Pop the P & return the Value.

9. Print the result of string after the evaluation of Postfix

10. Attach output and & input of above algorithm

Output

Evaluated Value is: 10

19/01/2021

Source code:

```
class node:
    global data
    global 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):
        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 elements in a node in a linear fashion but not necessarily contiguous. 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 one element from the list, all the element of list has to adjust itself every time we add it, is very tedious task so linked list is used to solving this type of problems.

Algorithm:-

1. Transversing of a linked list means visiting all the nodes in the linked list in order to perform some operation on them.

2. The entire linked list means can be accessed with 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.

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

Scanned with CamScanner

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

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 cannot revert back.

6. We may lose the reference to the 1st node in our linked list, & hence most of our linked list. In order to avoid making some unwanted changes to the 1st node, we will use a temporary node to transverse the entire linked list.

Start = linked list()

7. We will use this temporary node as a copy of the node we are currently transversing. Since, we are making temporary ~~rat~~ node a copy of current node the datatype of the temporary node should also be Node.

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.

12.

Output

```
>>> Start.addL(50)
>>> Start.addL(60)
>>> Start.addL(70)
>>> Start.addL(80)
>>> Start.addB(20)
>>> Start.addB(30)
>>> Start.addB(40)
>>> Start.display()

20
30
40
50
60
70
80
```

24/11/2024

9. But the 1st node is referred by current. So, we can transverse to 2nd order as h = h.next.

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

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

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

13. So, we can refer the last node of linked list self.s = None

14. We have to now see how to start transversing the linked list & how to identify whether we have reached the last node of linked list or not.

15. Attach the coding or input and output of the above algorithm.

Scanned with CamScanner

Practical - 9

Aim:- Program Based on Binary Search tree by implementing Inorder, Preorder & Postorder Traversal

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 in turn 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 in turn might have left & right Subtree
(iii) Transverse the right Subtree repeat it

→ Post order: (i) Transverse the left Subtree. The left Subtree in turn might have left & right Subtrees
(ii) Transverse the right Subtree
(iii) Visit the root node.

- Algorithm:-

1. Define class node & define init() method with 2 argument. Initialize the value in this method.

Source 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 successfully added to rightside at ", h.val)
                        break
                    else:
                        h = h.right

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

Scanned with CamScanner

```
        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()
```

Output

```
>>> t.add(25)
('Root is added Successfully', 25)
>>> t.add(15)
(15, 'Node is added to leftside Successfully at', 25)
>>> t.add(50)
(50, 'Node is added to rightside Successfully at', 25)
>>> t.add(10)
(10, 'Node is added to leftside Successfully at', 15)
>>> t.add(22)
(22, 'Node is added to rightside Successfully at', 15)
>>> t.add(35)
(35, 'Node is added to leftside Successfully at', 50)
```


2. Again, Define a class BST() that is Binary SearchTree with init() method with self argument & assign the root is None.

3. Define add() method for adding the node. Define a variable p that p = Node(value)

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 or arrange that in leftside.

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

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

7. After this, left Subtree & right Subtree, repeat this method to arrange the node according to Binary Search Tree

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

Scanned with CamScanner

9. In order, else statement used for giving that condition first left, root & then right node.

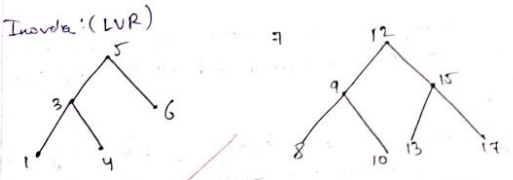
10. For Preorder, we have to give condition in else that first root, left & then right node.

11. For Postorder, In else part, assign left then right and then go for root node.

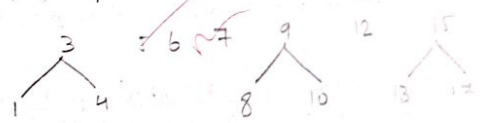
12. Display the output & input of above Algorithm.

• Inorder: (LVR)

Step 1



2.

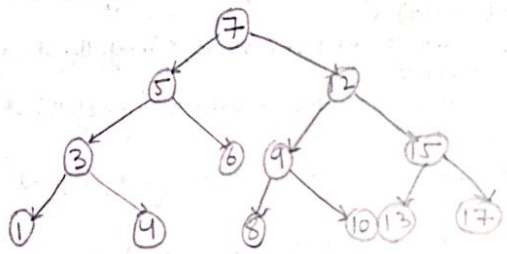


3 1 3 4 5 6 7 8 9 10 12 13 15 17


```
>>> t.add(70)
(70, 'Node is added to rightside successfully at', 50)
>>> t.add(4)
(4, 'Node is added to leftside successfully at', 10)
>>> t.add(12)
(12, 'Node is added to rightside successfully at', 10)
>>> t.add(18)
(18, 'Node is added to leftside successfully at', 22)
>>> t.add(24)
(24, 'Node is added to rightside successfully at', 22)
>>> t.add(33)
(33, 'Node is added to leftside successfully at', 35)
>>> t.add(44)
(44, 'Node is added to rightside successfully at', 35)
>>> t.add(60)
(60, 'Node is added to leftside successfully at', 70)
>>> t.add(90)
(90, 'Node is added to rightside successfully at', 70)
```

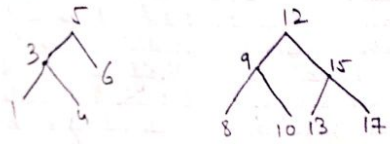
Scanned with CamScanner

Binary Tree

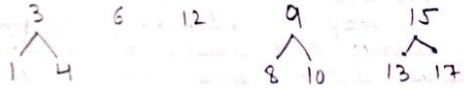


- Preorder (VLR

1 7



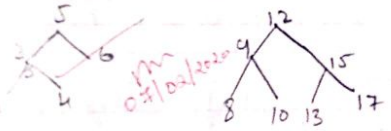
2 5 3 6 12 9 15



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

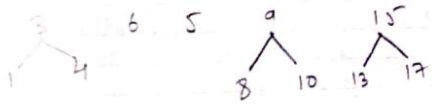
- Post order: (LRV)

1



7

2



6 5 12 7

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

Scanned with CamScanner

PRACTICAL - 10

Aim: To Sort a list using Merge Sort

Theory: Like QuickSort, Merge Sort is a Divide & Conquer algorithm. It divides input array in two halves, calls itself, for the two halves & 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] & arr[m+1... r] are sorted & merges the two sorted subarrays into one.
The array is recursively divided in two halves till the size becomes 1.
Once the size becomes 1, the merge process comes into action & starts merging arrays back till the complete array is merged.

Applications:
1. 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.
2. Inversion Count Problem
3. Used in external sorting.

Merge is more efficient than quick sort for some types of lists if the data to be sorted can only be efficiently accessed sequentially

```
def mergesort (arr):
    if len (arr)> 1:
        mid = len(arr)//2
        left half = arr [:mid]
        right half = arr[mid:]
        mergesort (left half)
        mergesort(right half)
        i=j = k= 0
        while i< len (left half) and j< len(right half):
            if left half[i] < right half[j]:
                arr[k]= left half [i]
                i=i+1
            else:
                arr[k]= right half[j]
                j= j+1
            k=k+1
        while i< len (left half):
            arr[k]= left half[i]
            i=i+1
            k=k+1
        while j< len(right half):
            arr[k]= right half[j]
            j= j+1
            k = k+1
arr = [27,89,70,55,62,99,45,14,10]
print ("RANDOM LIST: ", arr)
mergesort (arr)
print ("\n MergeSorted List: ", arr)
```

Scanned with CamScanner

Output

Random list: [27, 89, 70, 55, 62, 99, 45, 14, 10]

Merge sorted list: [10, 14, 27, 45, 55, 62, 70, 89, 99]

14/02/2020

It is thus is popular where sequentially accessed data structures very common.

Scanned with CamScanner

13 PRACTICAL - 11

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 & tail pointer will always point to the end of the queue.


```
class queue:
    global r
    global j
    def __init__(self):
        self.r=0
        self.j=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.j):
                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.j<=n-1):
            print("Data removed: ",self.l[self.j])
            self.l[self.j]=0
            self.j=self.j+1
        else:
            s=self.j
            self.j=0
            if(self.j<self.r):
                print(self.l[self.j])
```

```
        self.j = self.j+1
    else:
        print('Queue is empty')
        self.j = 5
q = queue()
```

Output

```
>>> q.add(100)
('Data added: ', 100)
>>> q.add(30)
('Data added: ', 30)
>>> q.add(20)
('Data added: ', 20)
>>> q.add(10)
('Data added: ', 10)
>>> q.add(50)
('Data added: ', 50)
>>> q.add(60)
('Queue is full')
>>> q.remove()
('Data removed: ', 100)
>>> q.l
[0, 30, 20, 10, 50, 60]
```


Initially, the head & the tail pointer will be pointing to the same location this would ~~be~~ mean that queue is empty. New data is always ~~dad~~ added to location pointed by the tail pointer. & Once the data is added, tail pointer is incremented to point to the next available location. Aplications:

Below ~~how~~ we have some common real-world examples where circular queue are used:

1. Computer Controlled Traffic signal system uses circular queue
2. CPU Scheduling & Memory Management.

14/03/2020.

Scanned with CamScanner