PRACTICAL-1

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

* Theory:- The proces of identifying or finding a particular record is called searching.
There are two types of search
→ linear search
→ Binary search
The linear search is further classified as
★ Sorted ★ Unsorted

Here we will look on unsorted linear search also known as sequential search, is a proces that checks every elements in the list sequented until the list sequential until the desired element is found when the elements to be searhed are not specifically arranged in ascending or descending order. They are arranged in random manner. That is what it calls unsorted linear search.

```
print("abhinav singh")
a=[5,4,6,24,56,24,5,2]
j=0
print(a)
s=int(input("enter no to be searched. "))
for i in range(len(a)):
    if(s==a[i]):
        print("no founded at ",i+1)
        j=1
        break
if(j==0):
    print("number not founded")

output:-
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p1 ds.py ==========
abhinav singh
[5, 4, 6, 24, 56, 24, 5, 2]
enter no to be searched: 4
no founded at  2
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p1 ds.py ==========
abhinav singh
[5, 4, 6, 24, 56, 24, 5, 2]
enter no to be searched: 01
number not founded
>>>
```

* Unsorted linear search

- The data is entered in random manner
- User need to specify the element to be searched in the entered list.
- Check the condition that whether the entered no matches if it matches then display the location plus increment 1 as data is stored from location zero
- If all elements are checked one by one and elements not found then prompt message no not founded.

PRACTICAL:- 2

* AIM:- To search a number from the list using linear sorted method.

* Theory:- Searching and sorting are different modes or type of data-structure.

SORTING:-
To basically sort the inputed data in ascending or decending manner.

SEARCHING:-
To search elements and to diplay the same.

In searching that too in linear sorted search the data is arranged in ascending to decending or viscaversa. That is all what it meant by searching through 'sorted' that is well arranged data.

```
print("abhinav")
a=[1,3,4,6,7,9,10]
j=0
print(a)
s=int(input("enter no to be searched: "))
if((s<a[0]))or((s>a[6])):
   print("number absent")
else:
   for i in range(len(a)):
      if(s==a[i]):
         print("number founded at: ",i+1)
         j=1
         break
   if(j==0):
         print("number not founded")
```

output:-

```
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p2 ds.py ==========
abhinav
[1, 3, 4, 6, 7, 9, 10]
enter no to be searched: 3
number founded at:  2
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p2 ds.py ==========
abhinav
[1, 3, 4, 6, 7, 9, 10]
enter no to be searched: 55
number absent
```

Sorted linear search :-

- The user is supposed to enter data in sorted manner
- User has to given an element for searching through sorted list.
- If element is found display with an updation as value is stored from location '0'.
- If data or element not found. print the same.
- In sorted order list of elements we can check the condition that whether the entered number lies from starting till the last element.

PRACTICAL:-3

* AIM:- To search a number from the given sorted list using binary search.

* THEORY:- A binary search also known as a half-interval search, is an algorithm used in computer science to locate a specified value (key) within an array. For the search to be binary the array must be sorted in either ascending or descending order.

At each step of the algorithm a comparison is made and the procedure branches into one of two direction

Specifically the key value is compared to the middle element of the array.

If the key value is less than or greater than this middle element, the algorithm known which half of the array to continue searching is because the array is sorted.

This process is repeated on progressively smaller segments of the array until the value is located.

```
print("abhinav")
a=[3,6,9,25,58,78,99]
print(a)
s=int(input("enter the no to searched"))
b=0
c=len(a)-1
d=int((b+c)/2)
if((s<a[b])or(s>a[c])):
   print("number not in range")
elif(s==a[c]):
   print("no founded at :  ",c+1)
elif(s==a[b]):
   print("no founded at",b+1)
else:
   while(b!=c):
      if(s==a[d]):
         print("no founded at ",d+1)
         break
      else:
         if(s<a[d]):
            c=d
            d=int((b+c)/2)
         else:
            b=d
            d=int((b+c)/2)
      if(s!=a[d]):
         print("no absent")
         break
```

output:-

```
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p3 ds.py ==========
abhinav
[3, 6, 9, 25, 58, 78, 99]
enter the no to searched25
no founded at  4
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p3 ds.py ==========
abhinav
[3, 6, 9, 25, 58, 78, 99]
enter the no to searched10
no absent
>>>
```

```
## Stack ##
print("abhinav singh")
class stack:
    global tos
    def __init__(self):
        self.l=[0,0,0,0,0,0,0]
        self.tos=-1
    def push(self,data):

        n=len(self.l)
        if self.tos==n-1:
            print("stack is full")
        else:
            self.tos=self.tos+1
            self.l[self.tos]=data
    def pop(self):
        if self.tos<0:
          print("stack empty")
        else:
            k=self.l[self.tos]
            print("data=",k)
            self.tos=self.tos-1
s=stack()
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.push(50)
s.push(60)
s.push(70)
s.push(80)
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
```

OUTPUT:-

```
= RESTART: C:/Users/Shree/AppData/Local/Programs/Python/Python37-32/stack.py =
abhinav singh
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
```

PRACTICAL - 5

* AIM :- To demonstrate the use of stack

* THEORY :- In computer science, a stack is an abstract data type that serves as a collection and pop, which adds an element to the collection and pop. which, removes the most recently added element that was not yet removed

The order may be LIFO (Last In First out) or FILO (First In last out)

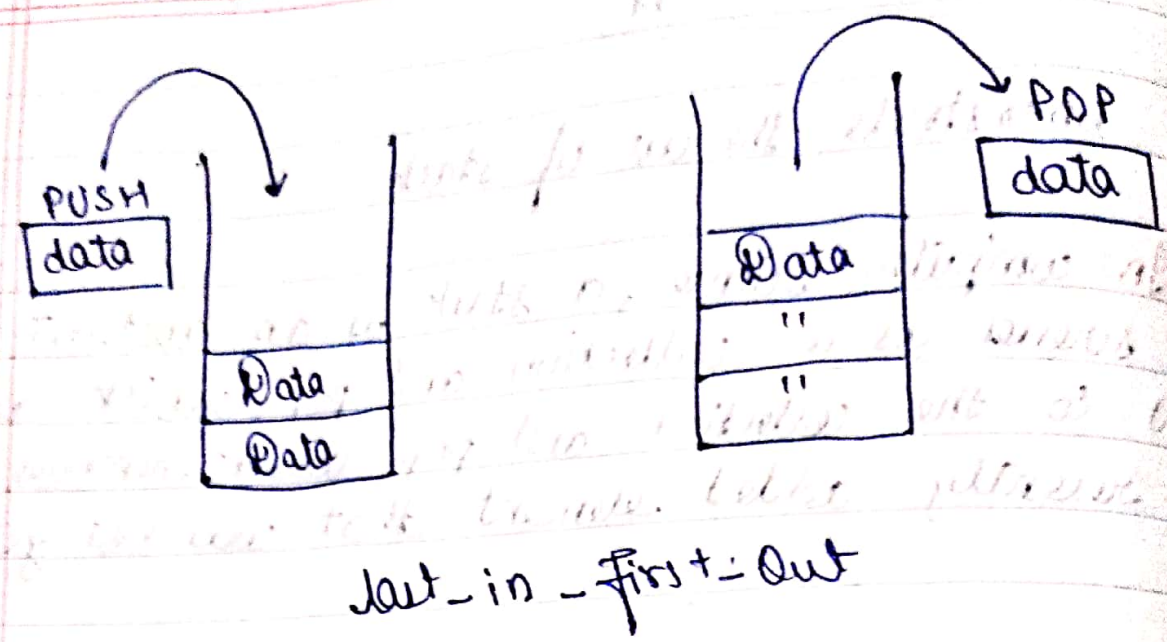
Three Basic operation are performed in the stack

- PUSH :- Adds an item in the if the stack is full then it is said to be over flow condition.

- POP :- Removes an item from the stack the item are popped in the reversed order in which they are pushed if the stack is empty, then it is said to be an underflow condition.

- Peek or Top :- Return top element of stack.

- Isempty :- Return true if stack is empty else false.



last_in - first - Out

```
## Queue add and Delete #
print("abhinav singh")
class Queue:
   global r
   global f
   def __init__(self):
      self.r=0
      self.f=0
      self.l=[0,0,0,0,0,0]
   def add(self,data):
      n=len(self.l)
      if self.r<n-1:
         self.l[self.r]=data
         self.r=self.r+1
      else:
         print("Queue is full")
   def remove(self):
      n=len(self.l)
      if self.f<n-1:
         print(self.l[self.f])
         self.f=self.f+1
      else:
         print("Queue is empty")
Q=Queue()
Q.add(30)
Q.add(40)
Q.add(50)
Q.add(60)
Q.add(70)
Q.add(80)
Q.remove()
Q.remove()
```

output:-

```
>>>
 RESTART: C:/Users/Shree/AppData/Local/Programs/Python/Python37-32/que n delete.py
abhinav singh
Queue is full
30
40
50
60
70
Queue is empty
```

```
Q.remove()
Q.remove()
Q.remove()
Q.remove()
```

PRALTICAL:- 5

* AIM:- To demonstrate queue add and delete.

* THEORY:- Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the outer end called as FRONT.

Front points to the beghining of the queue and rear points to the end of the queue.

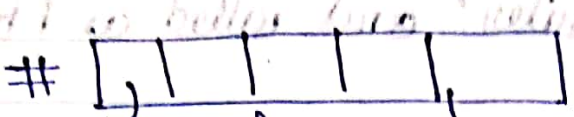
Queue Follows the FIFO (First-in-First-out) structure, according to its FIFO structure, elements inserted first will also be removed first.

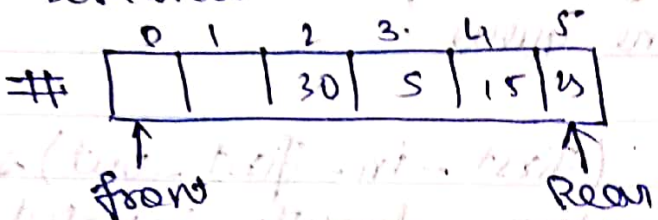
In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue) because queue is open on both of its ends

enqueue() can be termed as add() in queue
Dequeue() can be termed as delete or remove i.e deleting or removing of element.

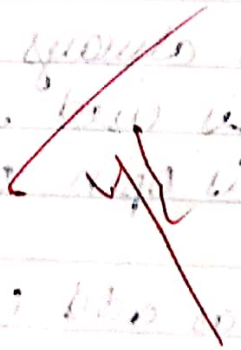
Front is used to get the front data item from a queue.

Rear is used to get the last item from a queue.





Front = 2
Rear = 5



```
#(circular queue)
print("abhinav singh")
class Queue:
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.l=[0,0,0,0,0,0]
    def add(self,data):
        n=len(self.l)
        if self.r<=n-1:
            self.l[self.r]=data
            print("data added:",data)
            self.r=self.r+1
        else:
            s=self.r
            self.r=0
            if self.r<self.f:
                self.l[self.r]=data
                self.r=self.r+1
            else:
                self.r=s
                print("Queue is full")
    def remove(self):
        n=len(self.l)
        if self.f<=n-1:
            print("data removed:",self.l[self.f])
            self.f=self.f+1
        else:
            s=self.f
            self.f=0
            if self.f<self.r:
                print(self.l[self.f])
                self.f=self.f+1
            else:
                print("Queue is empty")
                self.f=s
Q=Queue()
Q.add(44)
Q.add(55)
Q.add(66)
Q.add(77)
Q.add(88)
Q.add(99)
Q.remove()
Q.add(66)

output:-
>>>
 RESTART: C:/Users/Shree/AppData/Local/Programs/Python/Python37-32/circluar que.p
abhinav singh
data added: 44
```

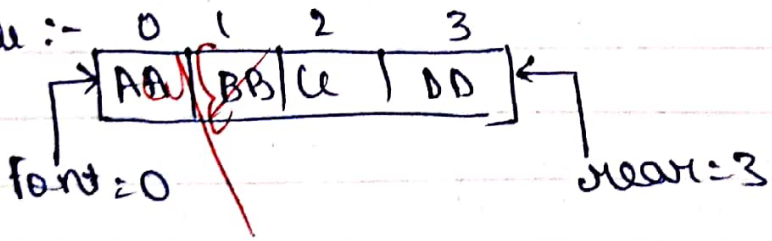
PRACTICAL:- 6

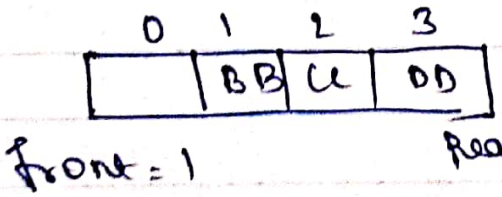
- AIM:- To demonstrate the use of circular queue is data - structure.

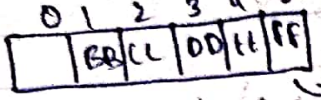
- THEORY:- The queue that we implement using an array suffer from one limitation. In that implementation there is a possibility that the queue is reported as full, even though is actually there might be empty slots at the begning of the queue.

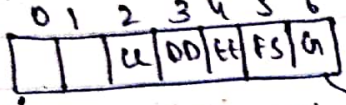
To overcome this limitation we can implement queue as circular queue. In circular queue we go on adding the elements to the queue and reach the end of the array. The next element is stored in the first slot of the array

Example :-









data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
data removed: 44

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:0
tel)] on win32
Type "copyright", "credits" or "license()" for more inf
>>> ========== RESTART ==========
>>>
ABHINAV SINGH 1772
20
30
30
40
40
50
50
60
60
70
70
80
>>>
```

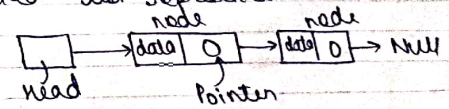
```
PROGRA### After Before limkedlist(simple)###

print("ABHINAV SINGH 1772")

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
```


PRACTICAL:- 07

- AIM:- To demonstrate the use of linked list in data structure

- THEORY:- A linked list is a sequence of data structure linked list is a sequence of links which contains item each link contain a connection to another link.
  - link:- Each link of a linked list can store a data called an elements.
  - NEXT:- Each link of a linked list contains a link to the next link called NEXT.
  - LINKED LIST :- A linked list contain the connection link to the first link called first.

linked list representation:-

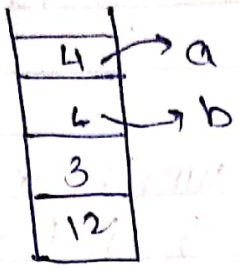


Types of linked list.
↳ simple
↳ Doubly
↳ circular

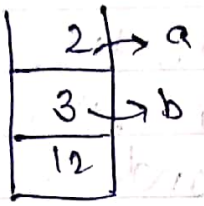
BASIC operations
↳ Instruction
↳ Deletion
↳ Display
↳ search
↳ delete.

```
        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)
start=linkedlist()
start.addL(50)
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
start.addB(30)
start.addB(20)
start.display()
```

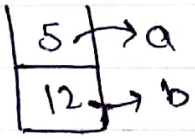
stack :-



b - a = b - 4 = 2 // store again in stack.



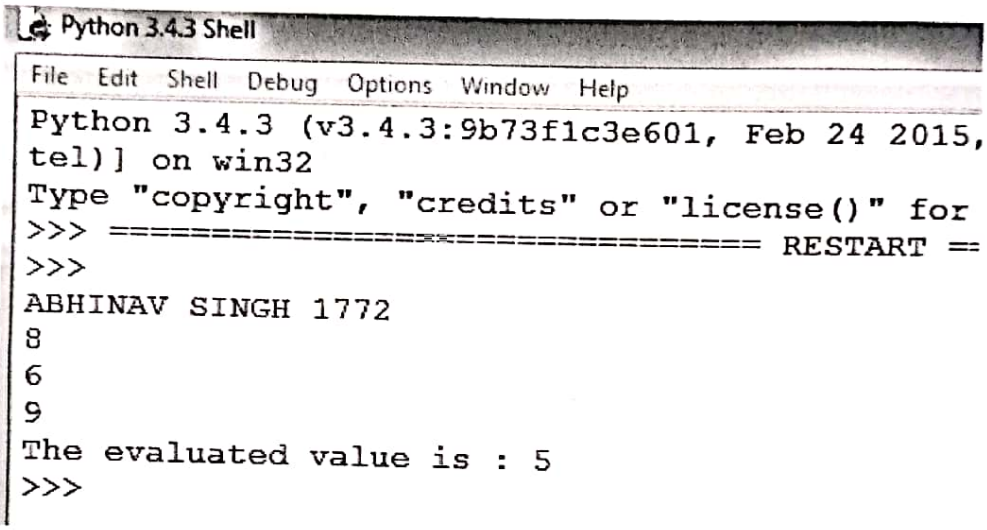
b + a = 3 + 2 = 5 // store result stack.



b * a = 12 * 5 = 60 //.

```
            b=stack.pop()
            stack.apend(int(b)/int(a))
    return stack.pop()
s="8 6 9 - +"
r=evaluate(s)
print("The evaluated value is :",r)
```

OUTPUT:



OUTPUT:

```
print("abhinav ")
a=[56,45,23,12,2,1]
print("before bubble sorting element list are ",a)
for passes in range(len(a)-1):
    for compare in range (len(a)-1-passes):
        if(a[compare]>a[compare+1]):
            temp=a[compare]
            a[compare]=a[compare+1]
            a[compare + 1]=temp
print("after bubble sort elements list",a)
```

output:-

```
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p4.py ==========
abhinav
before bubble sorting element list are  [56, 45, 23, 12, 2, 1]
after bubble sort elements list [1, 2, 12, 23, 45, 56]
>>>
```


PRACTICAL:- 09

* AIM:- To sort given random data by using bubble sort.

* Theory:- sorting is type in which any random data is sorted is arranged in ascending or descending order.

* Bubble sorting sometimes referred to as sinking sort. Is a simple sorting algorithm that repeatedly steps through the list compares adjacent elements and swaps them if they are in wrong order.

The pas through the list is repeated until the list is sorted. The alogrithm which is a comparision sort is named for the w way smaller or larger elements "bubble" to the top of the list

Although the alogrithm is simple, it is too slow as it compares one element checks if condition fail then only swaps otherwise.

* Examples: -

First pass: -

(51428) → (15428) Here algorithm compares the first two elements and swap since 5 > 1.
(15428) → (14528) swap since 5 > 4
(14528) → (14258) swap since 5 > 2
(14258) → (14258) Now since these element are already in order (8 > 5) algorithm does not swap them.

Second pass
(14258) → (14258)
(14258) → (12458) Swap since 4 > 2
(12458) → (12458)

Third pass
(12458) It checks and gives the data in sorted order.

```
## QUICK SORT ##
printf("ABHINAV 1772")
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)
def quickSortHelper(alist,first,last):
    if first<last:
        splitpoint=partition(alist,first,last)
        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)
def partition(alist,first,last):
    pivotvalue=alist[first]
    leftmark=first+1
    rightmark=last
    done=False
    while not done:
        while leftmark<=rightmark and alist[leftmark]<=pivotvalue:
            leftmark=leftmark+1
        while alist[rightmark]>=pivotvalue and rightmark>=leftmark:
            rightmark=rightmark-1
        if rightmark<leftmark:
            done=True
        else:
            temp=alist[leftmark]
            alist[leftmark]=alist[rightmark]
            alist[rightmark]=temp
    temp=alist[first]
    alist[first]=alist[rightmark]
    alist[rightmark]=temp
    return rightmark
alist=[42,54,45,67,89,66,55,80,100]
print("List before QUICK SORT")
print(alist)
quickSort(alist)
print("List after QUICK SORT:")
print(alist)
```

```
OUTPUT:-
ABHINAV 1772
List before QUICK SORT
[42,54,45,67,89,66,55,80,100]
List after QUICK SORT:
[42,45,54,55,66,67,80,89,100]
```

PRACTICAL:- 10

* AIM:- To 'evaluate' is to sort the given data in quick sort.

* THEORY:- Quick sort is an efficient sorting algorithm type of a divide & conquer algorithm type of a divend partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.

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

The key process in quick sort is partition(). Target of partition is given an array and an element x of array as pivot, put x - axis correct position in ~~sorted~~ array and put all smaller element (smaller than x) before x, & put all greater elements (greater than x) after x. All this should be done in linear time.

4/2

PRACTICAL:- 11

* AIM:- To sort given random data by using selection sort.

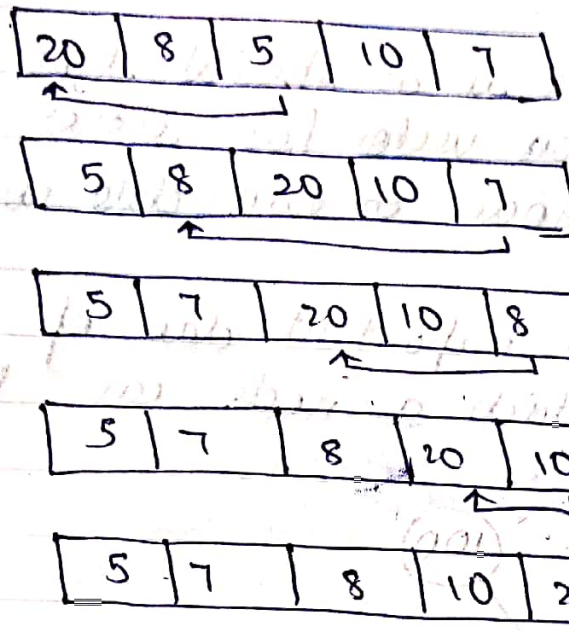
* THEORY:- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the begining. The algorithm maintains two sub arrays in a given array. The sub array which is already sorted.

The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the smallest value as it makes a pass and after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the smallest item is in the correct place.

After the second pass, the next smallest is in place. This process continues and requires n-1 passes to sort n items, since the final item must be place after the $(n-1)^{st}$ pass.

```
## SELECTION SORT ##
print("abhinav singh 1772")
a=[23,22,18,96,56,60]
print("Before sorting \n",a)
for i in range(len(a)-1):
 for j in range(len(a)-1):
  if(a[j]>a[i+1]):
    t=a[j]
    a[j]=a[i+1]
    a[i+1]=t
print("After selection sort \n",a)

========== RESTART: C:/Users/Shree/Desktop/class/selection sort.py ==========
abhinav singh 1772
Before sorting
 [23, 22, 18, 96, 56, 60]
After selection sort
 [18, 22, 23, 56, 60, 96]
>>>
```

Eg:- On each pass the smallest remaining item is selected and then placed in its proper location.

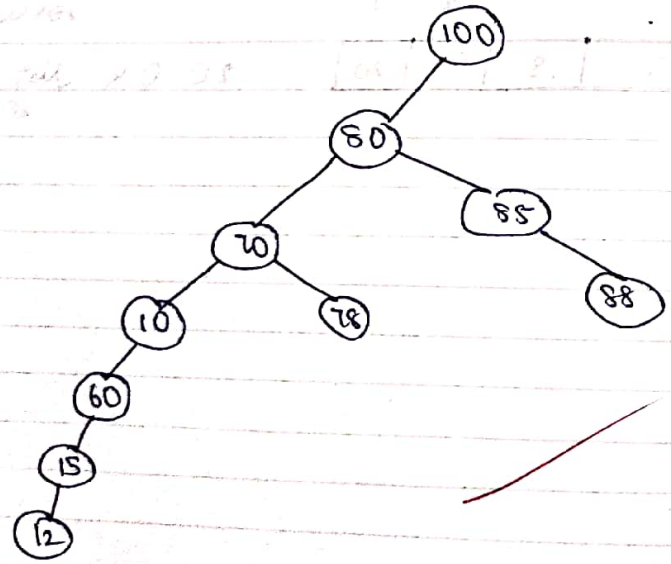


Practical:-12

- Aim:- Binary Tree and Traversal

- Theory:- A binary tree is a special type of tree in which every node or vector has either no child or one child node or two child nodes.

A binary tree is an important class of a tree data structure in which a node can have at most two children.




```
## BINARY SEARCH TREE ##
class Node:
    print("abhinav singh 1772")
    global r
    global l
    global data
    def __init__(self,l):
        self.l=None
        self.data=l
        self.r=None
class Tree:
    global root
    def __init__(self):
        self.root=None
    def add(self,val):
        if self.root==None:
          self.root =Node(val)
        else:
            newnode=Node(val)
            h=self.root
            while True:
                if newnode.data<h.data:
                    if h.l!=None:
```

```
T.add(50)
T.add(30)
T.add(47)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)

========== RESTART: C:/Users/Shree/Desktop/ppp abhinav.py ==========
abhinav singh 1772
11 added left of 400
17 added on right of 11
50 added on right of 17
30 added left of 50
47 added on right of 30
preorder
400
11
17
50
30
47
inorder
11
17
30
47
50
400
postorder
47
30
50
17
11
400
>>>
```

Traversal :- Traversal is a process to visit all the node of a tree and may print their value too

There are 3 ways we use to traverse a tree.

1) In order :- the left - subtree is visited 1st then the root and later the right sub tree, we should always remember that every node may represent a subtree itself.
Output produced is sorted key values in ASCENDING ORDER.

2) PRE ORDER :- The root node is visited 1st then the left subtree and finally the right subtree and finally ~~root node~~ right subtree

3) POST - ORDER :- The root node is visited last, left subtree, then the right subtree and finally root node.

PRACTICAL:-13

* AIM:- MERGE SORT

* THEORY:- Merge sort is a sorting technique based on divide and conquer technique with worst-case time complexity being O(nlogn), it is one of the most respected algorithm.

* Merge sort first divides the array into equal values and then combines them in a sorted manner.

* It divides input array in two halves, call itself for the two halves and then merges the two sorted values. The merge() function is used for merging 2 values. The merge(arr, l, m, r) is key process that assumes that arr[l...m] and arr[m+1...r] are sorted and merges the two sorted sub-arrays into one.

```
print("Abhinav singh 1772")
def sort(arr,l,m,r):
    n1=m-l+1
    n2=r-m
    L=[0]*(n1)
    R=[0]*(n2)
    for i in range(0,n1):
        L[i]=arr[l+i]
    for j in range(0,n2):
        R[j]=arr[m+1+j]
    i=0
    j=0
    k=l
    while i<n1 and j<n2:
        if L[i]<=R[j]:
            arr[k]=L[i]
            i+=1
        else:
            arr[k]=R[i]
            j+=1
        k+=1
        while i<n1:
            arr[k]=L[i]
            i+=1
            k+=1
        while j<n2:
            j+=1
            k+=1
def mergesort(arr,l,r):
    if l<r:
        m=int((l+(r-1))/2)
        mergesort(arr,l,m)
        mergesort(arr,m+1,r)
        sort(arr,l,m,r)
arr=[12,23,34,56,78,45,86,98,42]
print("array before sorting \n",arr)
n=len(arr)
mergesort(arr,0,n-1)
print("array after merge sorting \n",arr)
```

```
>>>
============ RESTART: C:/Users/Shree/Desktop/class/merage sort.py ============
Abhinav singh 1772
array before sorting
 [12, 23, 34, 56, 78, 45, 86, 98, 42]
array after merge sorting
 [12, 23, 34, 56, 42, 45, 78, 86, 98]
>>>
```