1. Python program to Use and demonstrate basic data structures.

Algorithm:

Step 1: Start
Step2: Create list.
Step 3: Insert the values.
Step 4: Create dictionary.
Step 5: Insert the values.
Step 6: Create tuple.
Step 7: Insert the values.
Step 8: Create set.
Step 9: Insert the values.
Step 10: Print the output.
Step 11: Stop.

Program:

```
print("List")
l1 = [1, 2,"ABC", 3, "xyz", 2.3]
print(l1)
print("Dictionary")
d1={"a":134,"b":266,"c":343}
print(d1)
print("Tuples")
t1=(10,20,30,40,50,40)
print (t1)
print("Sets")
s1={10,30,20,40,10,30,40,20,50,50}
print(s1)
```

Output:

```
List
[1, 2, 'ABC', 3, 'xyz', 2.3]
Dictionary
{'a': 134, 'b': 266, 'c': 343}
Tuples
(10, 20, 30, 40, 50, 40)
Sets
{40, 10, 50, 20, 30}
```

2. Implement an ADT with all its operations.

Algorithm:

Step 1: Start.
Step 2: Create class.
Step 3: Add the day ,month,year,months.
Step 4: if (self.y % 400 == 0) or (self.y % 4 == 0 and self.y % 100 != 0):
Print leap year.
else Print it is not a leap year.
Step 5: Stop.

Program:

```
class Date:
   def __init__(self, d, m, y):
      self.d = d
      self.m = m
      self.y = y

   def day(self):
      print("Day =", self.d)

   def month(self):
      print("Month = ", self.m)

   def year(self):
      print("Year = ", self.y)

   def monthName(self):
      months = ["Unknown","January","February","March","April","May","June","July",
            "August","September","October","November","December"]
      print("Month Name:", months[self.m])

   def isLeapYear(self):
      if (self.y % 400 == 0) or (self.y % 4 == 0 and self.y % 100 != 0):
         print("It is a Leap year")
      else:
         print("It is not a Leap year")

dd = int(input("Enter the day: "))
mm = int(input("Enter the month: "))
yy = int(input("Enter the year:"))
```

```
d1 = Date(dd, mm, yy)
d1.day()
d1.month()
d1.year()
d1.monthName()
d1.isLeapYear()
```

Output:

```
1 .Enter the day: 2
Enter the month: 5
Enter the year:2000
Day = 2
Month =  5
Year =  2000
Month Name: May
It is a Leap year

2. Enter the day: 7
Enter the month: 12
Enter the year: 2018
Day = 7
Month = 12
Year = 2018
Month Name: December
It is not a Leap year
```

3. Implement an ADT and compute space and time complexities.

Algorithm:

Step 1: Start
Step 2: Create Class
Step 3: Read the Employee Details.
Step 4: Start the process time (start = time.process_time())
Step 5: Trace memory allocation (tracemalloc.start())
Step 6: Print the Employee Details
Step 7: Print space required (Space required =",tracemalloc.get_traced_memory())
Step 8: End the process time (end = time.process_time())
Step 9: Print time required (Time required=",(end-start))
Step 10: Stop the trace (tracemalloc.stop())
Step 11: Stop

Program:

```
import time
import tracemalloc

class Employee:
   def __init__(self, name, desig, salary):
      self.name=name
      self.desig=desig
      self.salary=salary

   def read(self):
      self.name=input('Enter the Name:')
      self.desig=input('Enter the Designation:')
      self.salary=(int(input('Enter the Salary:')))

   def displayDetails(self):
      print("Name:", self.name, ", Designation:", self.desig, ", Salary:", self.salary)

e1=Employee('','',0)
start = time.process_time()
tracemalloc.start()
print("Details of a employee:")
e1.read()
e1.displayDetails()
print( "Space required =",tracemalloc.get_traced_memory())
end = time.process_time()
```

```
print("\nTime required=",(end-start))
tracemalloc.stop()
```

Output:

```
Details of a employee:
Enter the Name: Ram
Enter the Designation: Manager
Enter the Salary:200000
Name: Ram, Designation: Manager,Salary: 200000
Space required = (56938, 144998)

Time required= 0.0
```

4. Implement the above solution using array and Compute space and complexities and compare two solutions.

Algorithm:

Step 1: Start
Step 2: Create Class
Step 3: Define an array
Step 4: Read the Employee Details.
Step 5: Start the process time (start = time.process_time())
Step 6: Trace memory allocation (tracemalloc.start())
Step 7: Print the Employee Details
Step 8: Print space required (Space required =",tracemalloc.get_traced_memory())
Step 9: End the process time (end = time.process_time())
Step 10: Print time required (Time required=",(end-start))
Step 11: Stop the trace (tracemalloc.stop())
Step 12: Stop

Program:

```
import time
import tracemalloc

class Employee:
   def __init__(self):
      self.details = ['', '', 0]

   def read(self):
      self.details[0] = input('Enter the Name:')
      self.details[1] = input('Enter the Designation:')
      self.details[2] = int(input('Enter the Salary:'))

   def displayDetails(self):
      print("Name:", self.details[0], ", Designation:", self.details[1], ", Salary:", self.details[2])

e1 = Employee()
start = time.process_time()
tracemalloc.start()
print("Details of an employee:")
e1.read()
e1.displayDetails()
print("Space required =", tracemalloc.get_traced_memory())
end = time.process_time()
```

```
print("Time required=", end - start)
tracemalloc.stop()
```

Output:

```
Details of an employee:
Enter the Name: Jhon
Enter the Designation: Developer
Enter the Salary: 50000
Name: Jhon , Designation: Developer , Salary: 50000
Space required = (3159, 7174)
Time required= 0.015625
```

5. Implement Linear Search Compute space and time complexities, plot graph using asymptomatic notations.

Algorithm:

Step 1: Start
Step 2: Input array
Step 3: Input search_element
Step 4: i<-0
Step 5: While i <= length of array do
    i. If search_element = array [i]
        a. Return i
    ii. EndIf
    iii. i <- i + 1
Step 6: endwhile
Step 7: Return -1
Step 8: Stop

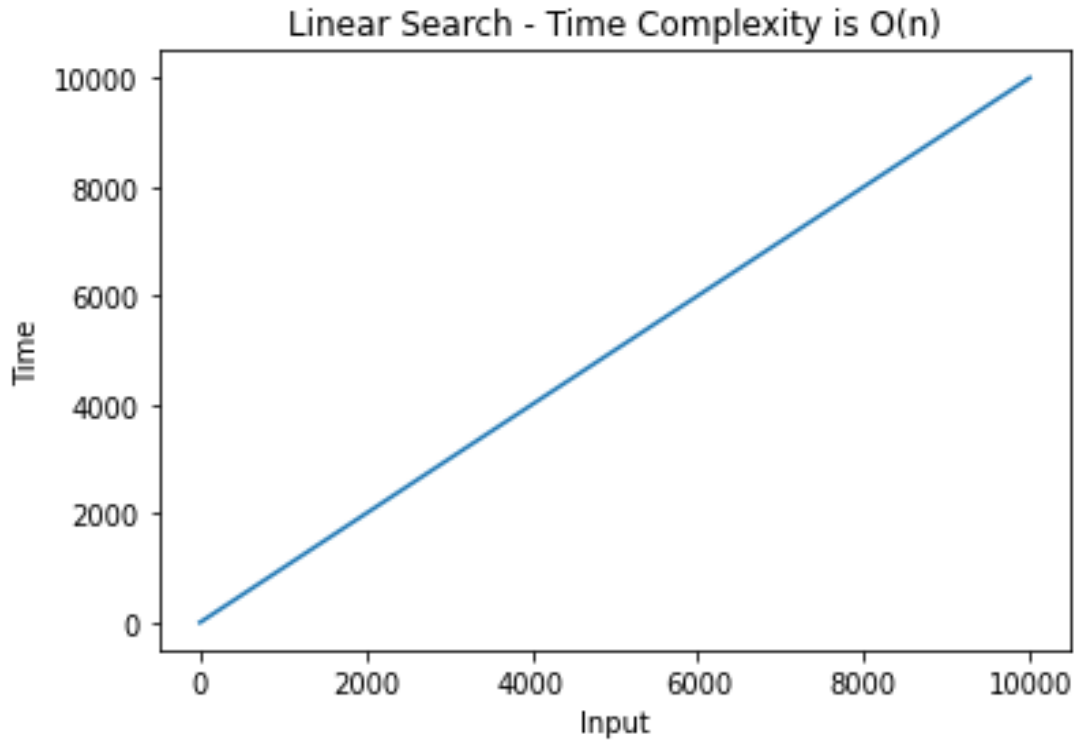
Program:

```
import time
import matplotlib.pyplot as plt

def linearsearch(a, key):
   n = len(a)
   for i in range(n):
      if a[i] == key:
         return i
   return -1

a = [13,24,35,46,57,68,79]

start = time.time()
print(f"The array elements are: {a}")
key = int(input("Enter the key element to search: "))
result = linearsearch(a, key)
if result == -1:
   print("Search unsuccessful.")
else:
   print(f"Search successful, key found at {result} location.")
end = time.time()
print(f"Runtime of the program is {end-start}")
x = list(range(1, 10000))
```

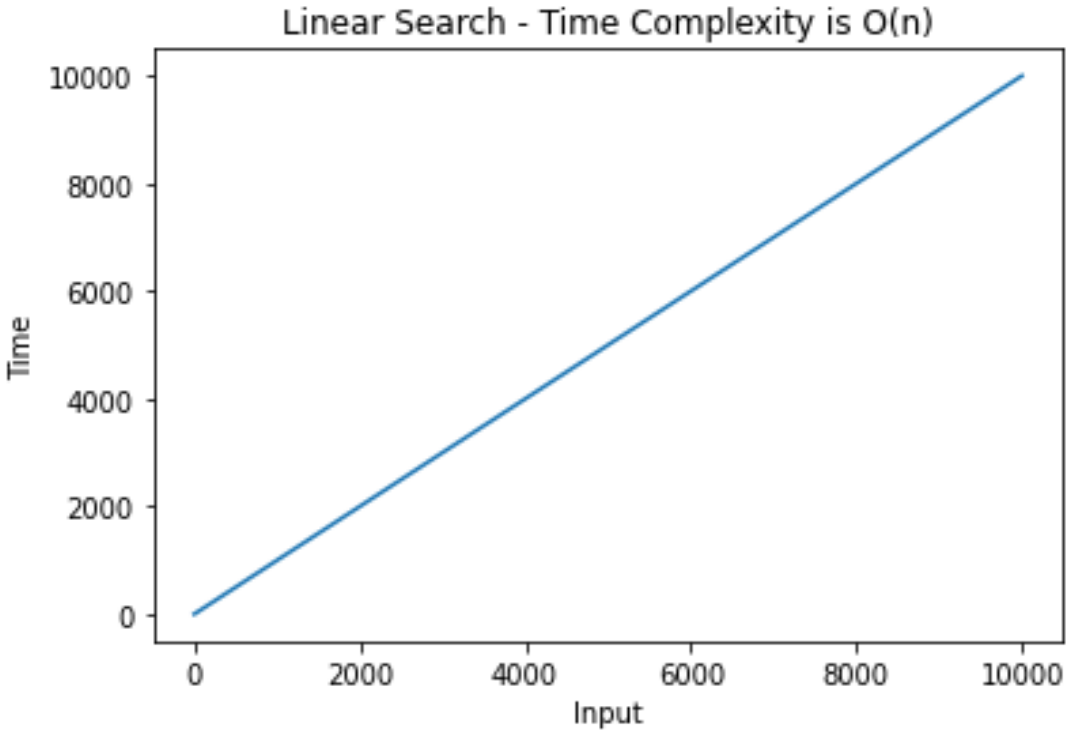
```
plt.plot(x, [y for y in x])
plt.title("Linear Search - Time Complexity is O(n)")
plt.xlabel("Input")
plt.ylabel("Time")
plt.show()
```

Output:

```
The array elements are: [13, 24, 35, 46, 57, 68, 79]
Enter the key element to search: 57
Search successful, key found at 4 location.
Runtime of the program is 12.657373905181885
```



2. The array elements are: [13, 24, 35, 46, 57, 68, 79]
Enter the key element to search: 90
Search unsuccessful.
Runtime of the program is 4.184961557388306



6. Implement Bubble, Selection, Insertion sorting algorithms compute space and time complexities, plot graph using asymptomatic notations.

Algorithm:

Step 1: Start
Step 2: Define a list of numbers
Step 3: Set the length of the list
Step 4: Display the unsorted list
Step 5: Perform Selection Sort on the list:
    i. for i in range(len(array)):
    ii. min_index = i
    iii. for j in range(i + 1, len(array)):
        a. if array[j] < array[min_index]:
            i. min_index = j
    iv. array[i], array[min_index] = array[min_index], array[i]
Step 6: Perform Insertion Sort on the list:
    i. for i in range(1, len(arr)):
    ii. key = arr[i]
    iii. j = i - 1
    iv. while j >= 0 and key < arr[j]:
        a. arr[j + 1] = arr[j]
        b. j -= 1
        c. arr[j + 1] = key
Step 7: Perform Bubble Sort on the list:
    i. n = len(array) - 1
    ii. while n >= 1:
    iii. i = 0
    iv. while i < n:
        a. if array[i] > array[i + 1]:
            i. array[i], array[i + 1] = array[i + 1], array[i]
    vii. i = i + 1
    viii. n = n – 1
Step 8: Display the sorted lists.
Step 9: Stop

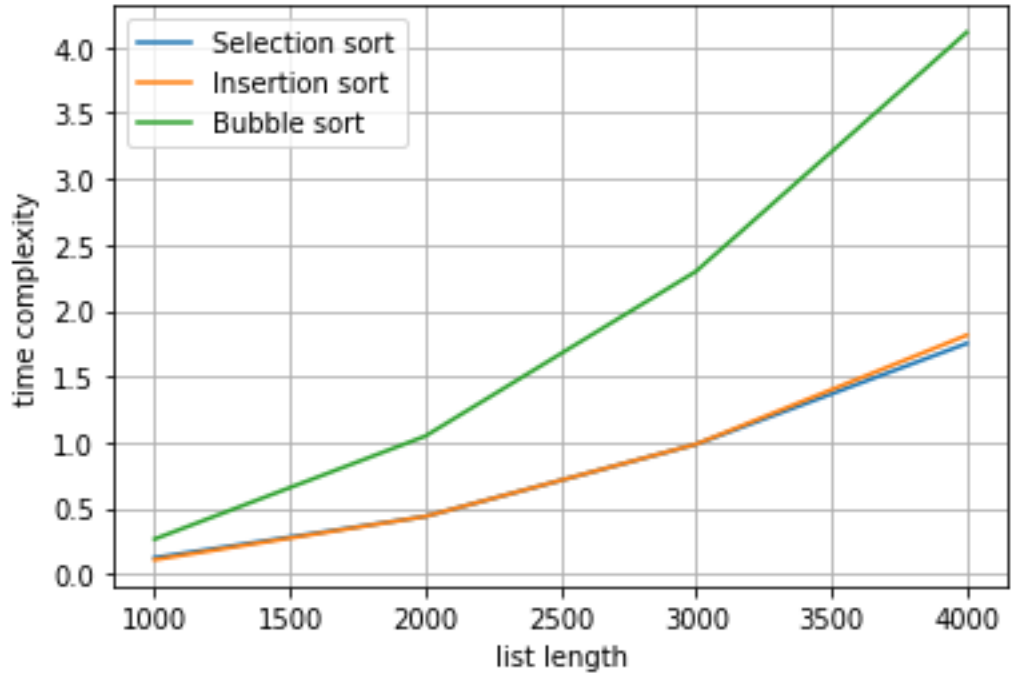
Program:

```
import time
import numpy as np
import matplotlib.pyplot as plt

def Selectionsort(array):
    for i in range(len(array)):
        min_index = i
        for j in range(i + 1, len(array)):
            if array[j] < array[min_index]:
                min_index = j
        array[i], array[min_index] = array[min_index], array[i]

def Insertionsort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def Bubblesort(array):
    n = len(array) - 1
    while n >= 1:
        i = 0
        while i < n:
            if array[i] > array[i + 1]:
                array[i], array[i + 1] = array[i + 1], array[i]
            i = i + 1
        n = n - 1

sorts = [    { "name": "Selection sort",     "sort": lambda arr: Selectionsort(arr)    },    { "name": "Insertion
sort",     "sort": lambda arr: Insertionsort(arr)    },    { "name": "Bubble sort",     "sort": lambda arr:
Bubblesort(arr)    },]

elements = np.array([i*1000 for i in range(1,5)])

plt.xlabel('list length')
plt.ylabel('time complexity')

for sort in sorts:
    times = list()
```

```
    start_all = time.time()
    for i in range(1, 5):
        start = time.time()
        a = np.random.randint(1000, size=i*1000)
        sort['sort'](a)
        end = time.time()
        times.append(end - start)
        print(sort["name"], "sorted", i * 1000, "elements in", end - start, "s")
    end_all = time.time()
    print(sort["name"], "sorted elements in", end_all - start_all, "s")
    plt.plot(elements, times, label=sort["name"])

plt.grid()
plt.legend()
plt.show()
```

Output:

Selection sort sorted 1000 elements in 0.12499856948852539 s
Selection sort sorted 2000 elements in 0.4374828338623047 s
Selection sort sorted 3000 elements in 0.9843454360961914 s
Selection sort sorted 4000 elements in 1.7499475479125977 s
Selection sort sorted elements in 3.296774387359619 s
Insertion sort sorted 1000 elements in 0.10937190055847168 s
Insertion sort sorted 2000 elements in 0.4374871253967285 s
Insertion sort sorted 3000 elements in 0.9855811595916748 s
Insertion sort sorted 4000 elements in 1.8124480247497559 s
Insertion sort sorted elements in 3.344888210296631 s
Bubble sort sorted 1000 elements in 0.26561737060546875 s
Bubble sort sorted 2000 elements in 1.0468435287475586 s
Bubble sort sorted 3000 elements in 2.296806812286377 s
Bubble sort sorted 4000 elements in 4.110517501831055 s
Bubble sort sorted elements in 7.719785213470459 s



7. Implement Binary Search compute space and time complexities, plot graph using asymptomatic notations.

Algorithm:

Step 1: Start
Step 2: If stop start then
i. Return -1
Step 3: Endif
Step 4: mid <-(start + stop)/2
Step 5: If array mid | <search element then
i . Call BinarySearch(parameters: array, mid + 1, stop. search_element)
Step 6: Else If array[mid | >search_element then
i. Call BinarySearch(parameters: array, start, mid-1 search element)
Step 7: Else
i. Return mid
Step 8: EndIf
Step 9: Stop

Program:

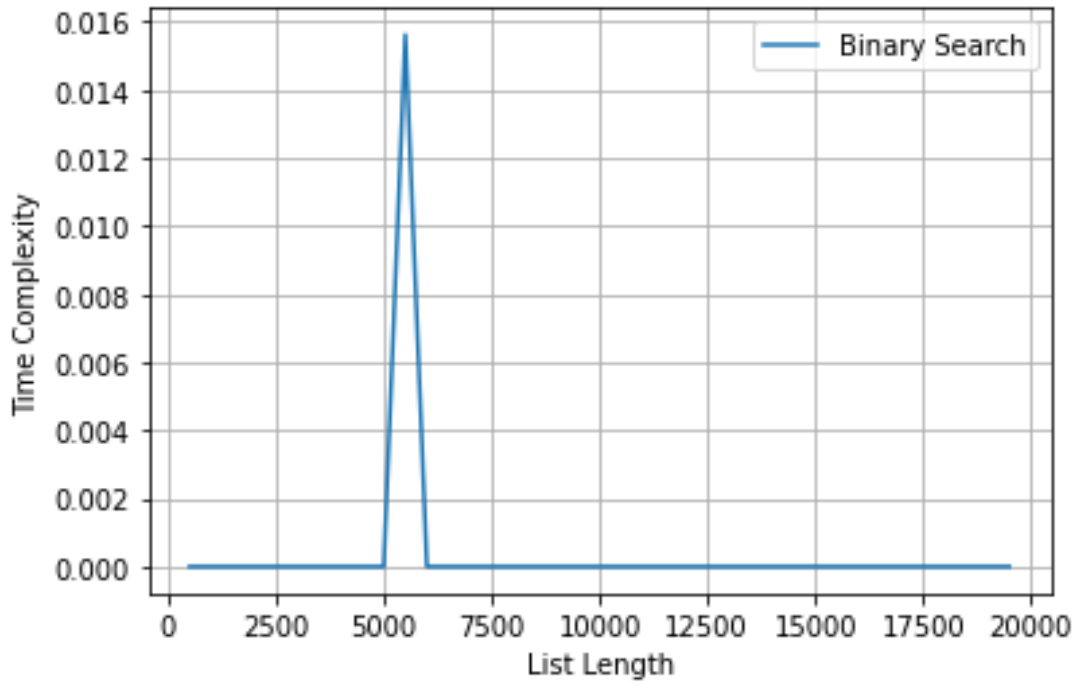
```
import time
import numpy as np
import matplotlib.pyplot as plt

def binary_search_basic(arr, target):
  low, high = 0, len(arr)
  while low < high:
    mid = (low + high) // 2
    if target < arr[mid]:
      high = mid
    elif target > arr[mid]:
      low = mid + 1
    else:
      return mid
  return -1

elements = np.array([i*500 for i in range(1, 40)])
plt.xlabel('List Length')
plt.ylabel('Time Complexity')
times = list()
```

```
for i in range(1, 40):
    start = time.time()
    a = np.random.randint(500, size= i * 500)
    binary_search_basic(a, 1)
    end = time.time()
    times.append(end - start)
    print("Time taken for Binary Search in", i * 500,
          "Elements is", end-start, "s")

plt.plot(elements, times, label="Binary Search")
plt.grid()
plt.legend()
plt.show()
```

Output:

```
Time taken for Binary Search in 500 Elements is 0.0 s
Time taken for Binary Search in 1000 Elements is 0.0 s
Time taken for Binary Search in 1500 Elements is 0.0 s
Time taken for Binary Search in 2000 Elements is 0.0 s
Time taken for Binary Search in 2500 Elements is 0.0 s
Time taken for Binary Search in 3000 Elements is 0.0 s
Time taken for Binary Search in 3500 Elements is 0.0 s
Time taken for Binary Search in 4000 Elements is 0.0 s
Time taken for Binary Search in 4500 Elements is 0.0 s
Time taken for Binary Search in 5000 Elements is 0.0 s
Time taken for Binary Search in 5500 Elements is 0.015622138977050781 s
Time taken for Binary Search in 6000 Elements is 0.0 s
Time taken for Binary Search in 6500 Elements is 0.0 s
Time taken for Binary Search in 7000 Elements is 0.0 s
Time taken for Binary Search in 7500 Elements is 0.0 s
Time taken for Binary Search in 8000 Elements is 0.0 s
Time taken for Binary Search in 8500 Elements is 0.0 s
Time taken for Binary Search in 9000 Elements is 0.0 s
Time taken for Binary Search in 9500 Elements is 0.0 s
Time taken for Binary Search in 10000 Elements is 0.0 s
Time taken for Binary Search in 10500 Elements is 0.0 s
Time taken for Binary Search in 11000 Elements is 0.0 s
Time taken for Binary Search in 11500 Elements is 0.0 s
Time taken for Binary Search in 12000 Elements is 0.0 s
Time taken for Binary Search in 12500 Elements is 0.0 s
Time taken for Binary Search in 13000 Elements is 0.0 s
```

Time taken for Binary Search in 13500 Elements is 0.0 s
Time taken for Binary Search in 14000 Elements is 0.0 s
Time taken for Binary Search in 14500 Elements is 0.0 s
Time taken for Binary Search in 15000 Elements is 0.0 s
Time taken for Binary Search in 15500 Elements is 0.0 s
Time taken for Binary Search in 16000 Elements is 0.0 s
Time taken for Binary Search in 16500 Elements is 0.0 s
Time taken for Binary Search in 17000 Elements is 0.0 s
Time taken for Binary Search in 17500 Elements is 0.0 s
Time taken for Binary Search in 18000 Elements is 0.0 s
Time taken for Binary Search in 18500 Elements is 0.0 s
Time taken for Binary Search in 19000 Elements is 0.0 s
Time taken for Binary Search in 19500 Elements is 0.0 s



8. Implement Merge Sort and compute space and time complexities, plot graph using asymptomatic notations.

Algorithm:

Step 1: Start
Step 2: if len(array) > 1:
i. mid = int(len(array)/2)
ii. L = array[:mid]
iii. R = array[mid:]
iv. call mergesort(mergeSort(L))
v. call mergesort mergeSort(R)
vi. i = 0
vii. j = 0
viii. k = 0
ix. while i < len(L) and j < len(R):
a. if L[i] <= R[j]:
i. array[k] = L[i]
ii. i = i + 1
b. array[k] = R[j]
i. j = j + 1
ii. k = k + 1
c. Endif
x. EndWhile
xi. while i < len(L):
a. array[k] = L[i]
b. k = k + 1
c. i = i + 1
xii.EndWhile
xiii. while j < len(R):
a. array[k] = R[j]
b. k = k + 1
c. j = j + 1
xiv. EndWhile
Step 3: Endif
Step 4: Stop

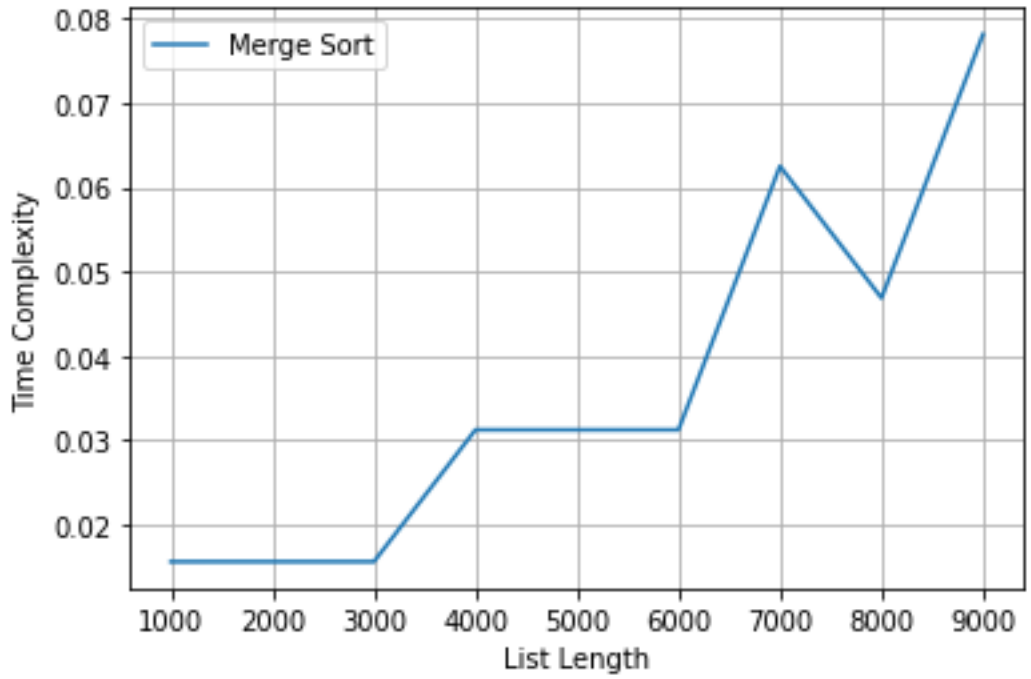
Program:

```
import time
import numpy as np
import matplotlib.pyplot as plt

def mergeSort(array):
    if len(array) > 1:
        mid = int(len(array)/2)
        L = array[:mid]
        R = array[mid:]
        mergeSort(L)
        mergeSort(R)
        i = 0
        j = 0
        k = 0
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                array[k] = L[i]
                i = i + 1
            else:
                array[k] = R[j]
                j = j + 1
            k = k + 1
        while i < len(L):
            array[k] = L[i]
            k = k + 1
            i = i + 1
        while j < len(R):
            array[k] = R[j]
            k = k + 1
            j = j + 1

elements = np.array([i*1000 for i in range(1, 10)])
plt.xlabel('List Length')
plt.ylabel('Time Complexity')
times = list()

for i in range(1, 10):
    start = time.time()
    a = np.random.randint(1000, size=i*1000)
    mergeSort(a)
    end = time.time()
```

```
    times.append(end-start)
    print("Merge Sort Sorted", i*1000, "Elements in", end-start, "s")

plt.plot(elements, times, label="Merge Sort")
plt.grid()
plt.legend()
plt.show()
```

Output:

```
Merge Sort Sorted 1000 Elements in 0.015621662139892578 s
Merge Sort Sorted 2000 Elements in 0.015624523162841797 s
Merge Sort Sorted 3000 Elements in 0.015623807907104492 s
Merge Sort Sorted 4000 Elements in 0.03124833106994629 s
Merge Sort Sorted 5000 Elements in 0.031249284744262695 s
Merge Sort Sorted 6000 Elements in 0.031248092651367188 s
Merge Sort Sorted 7000 Elements in 0.062497615814208984 s
Merge Sort Sorted 8000 Elements in 0.046875953674316406 s
Merge Sort Sorted 9000 Elements in 0.07812213897705078 s
```



9. Implement Quick Sort and compute space and time complexities, plot graph using asymptomatic notations.

Algorithm:

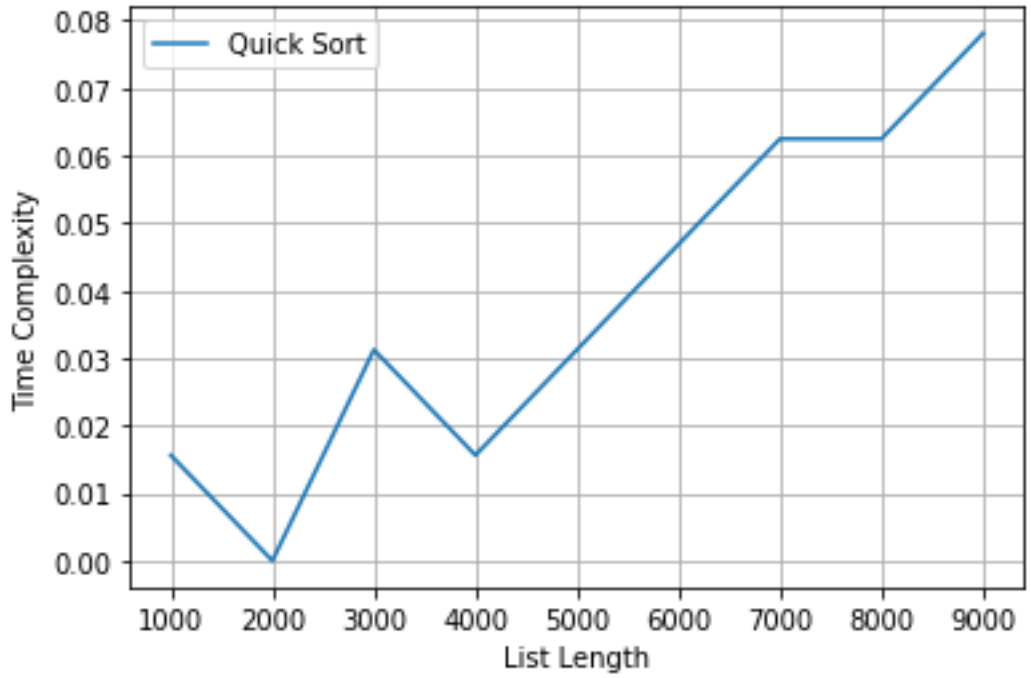
Step1: Start
Step2: While start < stop do
  i. While start < stop do
    a. If array[pivot] > array[stop] then
      i. Break
    b. EndIf
    c. stop <- stop - 1
  ii. End While
  iii. While start < stop do
    a. If array[pivot] < array[start] then
      i. Break
    b. EndIf
    c. start <- start + 1
  iv. End While
  v. If start < stop do
    a. Swap(array[start], array[stop])
  vi. EndIf
Step3. End While
Step4. If array[start] > array[pivot] then
  i. Swap(array[start], array[pivot])
  ii. Return start
Step5. Else
  i. Swap(array[start + 1], array[pivot])
  ii. Return start + 1
Step6. EndIf
Step7. Stop

Program:

```
import time
import numpy as np
import matplotlib.pyplot as plt

def partition(tempList, low, high):
   i = low - 1
   pivot = tempList[high]
   for j in range(low, high):
      if tempList[j] <= pivot:
         i += 1
         tempList[i], tempList[j] = tempList[j], tempList[i]
   tempList[i+1], tempList[high] = tempList[high], tempList[i+1]
   return (i+1)

def quickSort(tempList, low, high):
   if low < high:
      pi = partition(tempList, low, high)
      quickSort(tempList, low, pi-1)
      quickSort(tempList, pi+1, high)

elements = np.array([i*1000 for i in range(1, 10)])

plt.xlabel('List Length')
plt.ylabel('Time Complexity')

times = list()
for i in range(1, 10):
   start = time.time()
   a = np.random.randint(1000, size=i*1000)
   quickSort(a, 0, len(a)-1)
   end = time.time()
   times.append(end - start)
   print("Quick Sort Sorted", i*1000, "Elements in", end - start, "s")

plt.plot(elements, times, label="Quick Sort")
plt.grid()
plt.legend()
plt.show()
```

Output:

Quick Sort Sorted 1000 Elements in 0.015616655349731445 s
Quick Sort Sorted 2000 Elements in 0.0 s
Quick Sort Sorted 3000 Elements in 0.03124833106994629 s
Quick Sort Sorted 4000 Elements in 0.015624284744262695 s
Quick Sort Sorted 5000 Elements in 0.031248807907104492 s
Quick Sort Sorted 6000 Elements in 0.04687356948852539 s
Quick Sort Sorted 7000 Elements in 0.06249809265136719 s
Quick Sort Sorted 8000 Elements in 0.0624997615814209 s
Quick Sort Sorted 9000 Elements in 0.07812356948852539 s



10. Implement Fibonacci sequence with dynamic programming.

Program:

Step 1: Start
Step 2: Create Fibonacci
Step 3: for i in range(2, n - 1):
    i. a.append(a[i - 1] + a[i - 2])
    ii. return a
Step 4: Print Fibonacci sequence
Step 5: Stop

Program:

```
def fibtab(n):
   a = [0, 1]
   for i in range(2, n - 1):
      a.append(a[i - 1] + a[i - 2])
   return a
print(fibtab(6))
```

Output:

[0, 1, 1, 2, 3]

11. Implement Singly Linked List (Traversing the Nodes, Searching for a Node, Prepending Nodes, Removing Nodes)

Algorithm:

Step 1: Create a class Node with data and next pointer as attributes.
Step 2: Create a class LinkedList with head attribute.
Step 3: Initialize head attribute to None.
Step 4: Define a function insert that takes data as an argument.
Step 5: Create a new node with the given data.
Step 6: Check if the linked list is empty.
Step 7: If the linked list is not empty, traverse to the last node and add the new node at the end.
Step 8: If the linked list is empty, set the new node as the head.
Step 9: Define a function search that takes data as an argument.
Step 10: Traverse the linked list until the data is found.
Step 11: Return the node if the data is found, else return None.
Step 12: Define a function atbeg that takes a value as an argument.
Step 13: Create a new node with the given value.
Step 14: Set the next pointer of the new node to the current head.
Step 15: Set the head to the new node.
Step 16: Define a function printll to print the contents of the linked list.
Step 17: Traverse the linked list and print the data of each node.
Step 18: Define a function delend to delete the last node of the linked list.
Step 19: Check if the linked list is empty.
Step 20: If the linked list has only one node, set the head to None.
Step 21: Traverse the linked list until the second last node is found.
Step 22: Delete the last node and set the next pointer of the second last node to None.
Step 23: Call the required functions to perform the operations.
Step 24: Stop

Program:

```
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        newnode = Node(data)
        if self.head:
            current = self.head
            while current.next:
                current = current.next
            current.next = newnode
        else:
            self.head = newnode

    def search(self, data):
        curr_node = self.head
        while curr_node:
            if curr_node.data == data:
                return curr_node
            curr_node = curr_node.next
        return None

    def atbeg(self, value):
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
    def printll(self):
        current = self.head
        while current:
            print(current.data)
            current = current.next

    def delend(self):
        if self.head is None:
            return
        elif self.head.next is None:
            self.head = None
```

```
        else:
            temp_node = self.head
            while temp_node.next.next is not None:
                temp_node = temp_node.next
            print('Deleted item =', temp_node.next.data)
            temp_node.next = None

ll = LinkedList()
ll.insert(99)
ll.insert(98)
ll.insert('welcome')
ll.insert(23)
print("Content of List")
ll.printll()
ll.search(98)
ll.atbeg(5)
print("Content of List:")
ll.printll()
ll.delend()
ll.delend()
print("Content of List")
ll.printll()
```

Output:

```
Content of List
99
98
welcome
23
Content of List:
5
99
98
welcome
23
Deleted item = 23
Deleted item = welcome
Content of List
5
99
98
```

12. Implement Linked List Iterators.

Algorithm:

Step 1: Define a node current which will initially point to the head of the list.
Step 2: Declare and initialize a variable count to 0.
Step 3: Traverse through the list till current point to null.
Step 4: Increment the value of count by 1 for each node encountered in the list.

Program:

```
if __name__=='__main__':
   list = []
   list.append(1)
   list.append(2)
   list.append(3)
   list.append(8)

for i in list:
   print(i, end = ' ')
```

Output:

1 2 3 8

13. Implement DLL.

Step 1: Define a Node class with three attributes: data, prev, and next.

Step 2: Define a DoublyLinkedList class with a start_node attribute initialized to None.

Step 3: Implement an InsertToEmptyList() method that takes a data parameter. If start_node is None, create a new node with the given data and set it as the start_node. Otherwise, print a message stating that the list is not empty.

Step 4: Implement an InsertToEnd() method that takes a data parameter. If start_node is None, create a new node with the given data and set it as the start_node. Otherwise, traverse the list until the last node is reached, create a new node with the given data, and link it as the next node of the last node.

Step 5: Implement a DeleteAtStart() method that removes the first node of the list. If start_node is None print a message stating that the list is empty. If start_node.next is None, set start_node to None. Otherwise, set start_node to the next node and set its prev attribute to None.

Step 6: Implement a DeleteAtEnd() method that removes the last node of the list. If start_node is None, print a message stating that the list is empty. If start_node.next is None, set start_node to None. Otherwise, traverse the list until the last node is reached, set its prev node's next attribute to None.

Step 7: Implement a display() method that prints the data of each node in the list. If start_node is None, print a message stating that the list is empty. Otherwise, traverse the list, print each node's data, and move to the next node.

Step 8: Create an instance of the DoublyLinkedList class named NewDoublyLinkedList.

Step 9: Call the InsertToEmptyList() method with a data value of 10 to insert a new node into the empty list.

Step 10: Call the InsertToEnd() method with data values of 20, 30, 40, 50, and 60 to insert new nodes to the end of the list.

Step 11: Call the display() method to print the list.

Step 12: Call the DeleteAtStart() method to remove the first node of the list.

Step 13: Call the DeleteAtEnd() method to remove the last node of the list.

Step 14: Call the display() method to print the modified list.

Program:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.start_node = None

    def InsertToEmptyList(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
        else:
            print("The list is not empty")

    def InsertToEnd(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
            return
        n = self.start_node
        while n.next is not None:
            n = n.next
        new_node = Node(data)
        n.next = new_node
        new_node.prev = n

    def DeleteAtStart(self):
        if self.start_node is None:
            print("The linked list is empty, no elements to delete")
            return
        if self.start_node.next is None:
            self.start_node = None
            return
        self.start_node = self.start_node.next
        self.start_node.prev = None

    def DeleteAtEnd(self):
        if self.start_node is None:
```

```
            print("The linked list is empty, no elements to delete")
            return
        if self.start_node.next is None:
            self.start_node = None
            return
        n = self.start_node
        while n.next is not None:
            n = n.next
        n.prev.next = None

    def display(self):
        if self.start_node is None:
            print("The list is empty")
            return
        else:
            n = self.start_node
            while n is not None:
                print("Element is:", n.data)
                n = n.next
                print("\n")

NewDoublyLinkedList = DoublyLinkedList()

NewDoublyLinkedList.InsertToEmptyList(10)

NewDoublyLinkedList.InsertToEnd(20)
NewDoublyLinkedList.InsertToEnd(30)
NewDoublyLinkedList.InsertToEnd(40)
NewDoublyLinkedList.InsertToEnd(50)
NewDoublyLinkedList.InsertToEnd(60)

NewDoublyLinkedList.display()

NewDoublyLinkedList.DeleteAtStart()

NewDoublyLinkedList.DeleteAtEnd()

NewDoublyLinkedList.display()
```

Output:

Element is: 10

Element is: 20

Element is: 30

Element is: 40

Element is: 50

Element is: 60

Element is: 20

Element is: 30

Element is: 40

Element is: 50

14. Implement CDLL

Algorithm:

Step 1: Create a Node class with the attributes value, prev, and next.

Step 2: Create a CDoublyLinkedList class with the attributes head and tail.

Step 3: Define the createCDLL method that takes a value as input and creates a new node with the given value. Set the new node as both head and tail.

Step 4: Define the insertAtEnd method that takes a value as input and creates a new node with the given value. If the list is empty, set the new node as both head and tail. Otherwise, set the new node as the next node of the current tail, and set the current tail as the previous node of the new node. Set the next node of the new node as the head and the previous node of the head as the new node.

Step 5: Define the insertAtBeg method that takes a value as input and creates a new node with the given value. Set the next node of the new node as the current head, and set the previous node of the current head as the new node. Set the new node as the new head and set the previous node of the head as the current tail. Set the next node of the current tail as the new head.

Step 6: Define the delBeg method that deletes the first node of the list. If the list is empty, return. If the list has only one node, set both head and tail to None. Otherwise, set the next node of the current head as the new head, set the previous node of the new head as the current tail, and set the next node of the current tail as the new head.

Step 7: Define the searchList method that takes a value as input and searches for the value in the list. If the list is empty, print a message saying that the list does not exist. Otherwise, traverse the list from head to tail until the value is found or the end of the list is reached. If the value is found, print a message with the position of the node where the value is found. If the value is not found, print a message saying that the value does not exist in the list.

Step 8: Define the Display method that prints the values of all nodes in the list. If the list is empty, print a message saying that the list does not exist. Otherwise, traverse the list from head to tail and print the value of each node.

Step 9: Create an instance of the CDoublyLinkedList class.

Step 10: Use the create CDLL method to create the list with an initial value.

Step 11: Use the insertAtBeg and insertAtEnd methods to add more nodes to the list.

Step 12: Use the Display method to print the contents of the list.

Step 13: Use the delBeg method to delete the first two nodes of the list.

Step 14: Use the Display method again to print the contents of the list after deletion.

Step 15: Use the searchList method to search for values that do and do not exist in the list.

Program:

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.prev = None
        self.next = None
class CDoublyLinkedList:
    def __init__(self):
        self.head=None
        self.tail=None

    def createCDLL(self, value):
        new_node = Node(value)
        new_node.next=new_node
        new_node.prev=new_node
        self.head=new_node
        self.tail=new_node
        print("The circular doubly linked list has been created")

    def insertAtEnd(self, value):
        new_node=Node(value)
        if self.head is None:
            self.head=new_node
            self.tail=new_node
            new_node.next=new_node
            new_node.prev=new_node
```

```
            return
        last_node=self.tail
        last_node.next=new_node
        new_node.prev=last_node
        new_node.next=self.head
        self.tail=new_node
        self.head.prev=self.tail

    def insertAtBeg(self,value):
        new_node=Node(value)
        new_node.next=self.head
        self.head.prev=new_node
        self.head=new_node
        self.head.prev=self.tail
        self.tail.next=self.head

    def delBeg(self):
        if(self.head==None):
            return
        elif(self.head is not None):
            next_node=self.head.next
            next_node.prev=self.tail
            self.head=next_node
            self.tail.next=self.head
            return
    def searchList(self,value):
```

```
        position=0
        found=0
        if self.head is None:
            print("The linked list does not exist")
        else:
            temp_node=self.head
            while True:
                position=position+1
                if temp_node.value==value:
                    print("The required value was found at position:" +str(position))
                    found=1
                    break
                if temp_node==self.tail:
                    print("The required value does not exist in the list")
                    break
                temp_node = temp_node.next


    def Display(self):
        if self.head==None:
            print("The list does not exist.")
        else:
            temp_node=self.head
            while True:
                print(temp_node.value)
                if temp_node==self.tail:
                    break
```

```
            temp_node=temp_node.next
CDLL=CDoublyLinkedList()
CDLL.createCDLL(10)
CDLL.insertAtBeg(20)
CDLL.insertAtEnd(30)
CDLL.insertAtEnd(40)
CDLL.insertAtEnd(50)
CDLL.insertAtEnd(60)
print("List contents are")
CDLL.Display()
CDLL.delBeg()
CDLL.delBeg()
print("List Contents after deleting:")
CDLL.Display()
CDLL.searchList(7)
CDLL.searchList(60)
```

Output:

```
The circular doubly linked list has been created
List contents are
20
10
30
40
50
60
List Contents after deleting:
30
40
50
60
The required value does not exist in the list
The required value was found at position: 4
```

15. Implement Stack Data Structure.

Algorithm:

Step 1 – Checks if the stack is full.
Step 2 – If the stack is full, produces an error and exit.
Step 3 – If the stack is not full, increments top to point next empty space.
Step 4 – Adds data element to the stack location, where top is pointing.
Step 5 – Returns success.

Program:

```
def create_stack():
   stack = []
   return stack

def check_empty(stack):
   return len(stack) == 0

def push(stack, item):
   stack.append(item)
   print("Pushed item: " + item)

def pop(stack):
   if (check_empty(stack)):
      return "stack is empty"
   return stack.pop()
stack = create_stack()
push(stack, str(10))
push(stack, str(20))
push(stack, str(30))
push(stack, str(40))
print("\n")
print("Popped item: " + pop(stack))
print("\n")
print("Stack after popping an element: " + str(stack))
```

Output:

Pushed item: 10
Pushed item: 20
Pushed item: 30
Pushed item: 40

Popped item: 40

Stack after popping an element: ['10', '20', '30']

16. Implement bracket matching using stack.

Algorithm:

Step 1: Define two lists open_list and close_list containing opening and closing parenthesis respectively.
Step 2: Define a function check_parenthesis that takes a string mystr as input.
Step 3: Initialize an empty stack.
Step 4: Loop through each character i in mystr.
Step 5: If i is an opening parenthesis (i.e., i is in open_list), push it onto the stack.
Step 6: If i is a closing parenthesis (i.e., i is in close_list):

i. Find the index of i in close_list.
ii. If the stack is not empty and the corresponding opening parenthesis (i.e., open_list[pos]) is at the top of the stack, pop the opening parenthesis from the stack.
iii. If the above condition is not met, return "unbalanced".

Step 7: After the loop, if the stack is empty, return "balanced".
Step 8: Otherwise, return "unbalanced".
Step 9: Test the function with some sample strings to check if it correctly identifies balanced or unbalanced parenthesis.

Program:

```
open_list = ["[", "{", "("]
close_list = ["]", "}", ")"]

def check_parenthesis(mystr):
    stack = []
    for i in mystr:
        if i in open_list:
            stack.append(i)
        elif i in close_list:
            pos = close_list.index(i)
            if len(stack) > 0 and open_list[pos] == stack[len(stack) - 1]:
                stack.pop()
            else:
                return "unbalanced"
    if len(stack) == 0:
        return "balanced"
    else:
        return "unbalanced"

string = "{[]{()}}"
print(string, "-", check_parenthesis(string))

string = "[{}{})()]"
print(string, "-", check_parenthesis(string))

string = "((()))"
print(string, "-", check_parenthesis(string))
```

Output:

```
{[]{()}} - balanced
[{}{})() - unbalanced
((())) – balanced
```

17. Program to demonstrate recursive operations (Factorial/Fibonacci).

Algorithm:

Step1: START
Step 2: Input the non-negative integer 'n'
Step 3: If (n==o || n==1)
i. return n;
else
i. return fib(n-1)+fib(n-2);
Step 4: Print, nth Fibonacci number
Step 5: END

Program:

```
def fib(n):
   if n <= 1:
      return n
   else:
      return(fib(n-1) + fib(n-2))

n = int(input('Enter the number:\n'))
if n <= 0:
   print("Plese enter a positive integer")
else:
      print("Fibonacci sequence:")
      for i in range(n):
         print(fib(i))
```

Output:

```
Enter the number:
5
Fibonacci sequence:
0
1
1
2
3
```

18. Implement solution of Towers of Hanoi.

Algorithm:

Step1: Create a tower_of_hanoi recursive function and pass two arguments: the number of disks n and the name of the rods such as source, aux, and destination.
Step2: We can define the base case when the number of disks is 1. In this case, simply move the one disk from the source to destination and return.
Step3: Now, move remaining n-1 disks from source to auxiliary using the target as the auxiliary.
Step4: Then, the remaining 1 disk move on the source to destination
Step5: Move the n-1 disks on the auxiliary to the target using the source as the auxiliary.

Program:

```
def TowerOfHanoi(disks, source,  auxiliary, destination):
   if (disks == 1):
      print("Move disk 1 from", source, "to", destination)
      return
   TowerOfHanoi(disks-1, source, destination, auxiliary)
   print("Move disk", disks, "from", source, "to", destination)
   TowerOfHanoi(disks-1, auxiliary, source, destination)

disks = int(input('Enter the number of disks:'))
TowerOfHanoi(disks, 'A', 'B', 'C')
```

Output:

```
Enter the number of disks: 4
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
Move disk 4 from A to C
Move disk 1 from B to C
Move disk 2 from B to A
Move disk 1 from C to A
Move disk 3 from B to C
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
```

19. Implement Queue.

Algorithm:

Step1: start
Step2: Initialize a queue
Step3: adding the element in the queue.
Step4: Removing element from queue.
Step5: Return Boolean for Empty.
Step6: Remove and return an item from the queue.
Step7: Stop.

Program:

```
from queue import Queue
q = Queue(maxsize = 3)

print(q.qsize())
q.put('a')
q.put('b')
q.put('c')

print("\nFull: ", q.full())

print("\nElements dequeued from the queue")
print(q.get())
print(q.get())
print(q.get())

print("\nEmpty: ", q.empty())

q.put(1)
print("\nEmpty: ", q.empty())
print("Full: ", q.full())
```

Output:

```
0

Full:  True

Elements dequeued from the queue
a
b
c

Empty:  True

Empty:  False
Full:  False
```

20. Implement priority queue.

Algorithm:

Step1: Create a function heapify() to heapify the elements in the Binary Tree if any changes are made.
Step2: Find the largest among root, left child, and right child, then recursively call the same function until the atches the largest element.
Step3: Create a function insert() to insert an element into the tree, which takes an array and the number which is to be inserted as input.
Step 4: If the size of the array is zero, then this number will be the root, else append the number and call the heapify function recursively to heapify the elements.
Step5: Create a function deleteNode() that deletes the selected element from the tree
Step 6: Delete the element and again heapify the elements recursively.
Step 7: Insert elements into an empty array using the insert() function then try deleting an element from the tree.

Program:

```
class PriorityQueue(object):
  def __init__(self):
    self.queue = []

  def __str__(self):
    return ' '.join([str(i) for i in self.queue])

  def isEmpty(self):
    return len(self.queue) == 0

  def insert(self, data):
    self.queue.append(data)

  def delete(self):
    try:
      max_val = 0
      for i in range(len(self.queue)):
        if self.queue[i] > self.queue[max_val]:
          max_val = i
      item = self.queue[max_val]
      del self.queue[max_val]
      return item
    except IndexError:
      print()
      exit()
```

```
if __name__ == '__main__':
    myQueue = PriorityQueue()
    myQueue.insert(12)
    myQueue.insert(1)
    myQueue.insert(14)
    myQueue.insert(7)
    print(myQueue)
    while not myQueue.isEmpty():
        print(myQueue.delete())
```

Output:

```
12 1 14 7
14
12
7
1
```

21. Implement Binary Search tree and its operations using list.

Algorithm:

Step 1: Create a class BSTNode with attributes data, leftChild, and rightChild.
Step 2: Define the insertNode() function which takes root_node and node_value as parameters.
Step 3: Check if the root_node is None, if yes, set the node_value as the root_node data.
Step 4: If the node_value is less than or equal to the root_node data, check if the root_node leftChild is None.
Step 5: If the root_node leftChild is None, create a new BSTNode with node_value as the data and set it as the leftChild of the root_node.
Step 6: If the root_node leftChild is not None, recursively call the insertNode() function with the root_node leftChild and node_value as the parameters.
Step 7: If the node_value is greater than the root_node data, check if the root_node rightChild is None.
Step 8: If the root_node rightChild is None, create a new BSTNode with node_value as the data and set it as the rightChild of the root_node.
Step 9: If the root_node rightChild is not None, recursively call the insertNode() function with the root_node rightChild and node_value as the parameters.
Step 10: Return "The node has been successfully inserted".
Step 11: Define the minValueNode() function which takes bstNode as the parameter.
Step 12: Initialize current as the bstNode.
Step 13: While current leftChild is not None, set current as current leftChild.
Return current.
Step 14: Define the deleteNode() function which takes root_node and node_value as parameters.
Step 15: Check if the root_node is None, if yes, print "Element is not found" and return root_node.
Step 16: If the node_value is less than the root_node data, recursively call the deleteNode() function with the root_node leftChild and node_value as the parameters.
Step 17: If the node_value is greater than the root_node data, recursively call the deleteNode() function with the root_node rightChild and node_value as the parameters.
Step 18: If the node_value is equal to the root_node data, check if the root_node leftChild is None.
Step 19: If the root_node leftChild is None, set temp as the root_node rightChild, set root_node as None, and return temp.
Step 20: If the root_node rightChild is None, set temp as the root_node leftChild, set root_node as None, and return temp.
Step 21: Set temp as the minimum value node in the root_node rightChild subtree by calling the minValueNode() function.
Step 22: Set the root_node data as the temp data.
Step 23: Recursively call the deleteNode() function with the root_node rightChild and temp data as the parameters.
Step 24: Return root_node.
Step 25: Define the searchNode() function which takes root_node and node_value as parameters.
Step 26: Check if the root_node is None, if yes, print "Element is not found" and return.
Step 27: If the node_value is equal to the root_node data, print "The element has been found" and return.

Step 28: If the node_value is less than the root_node data, recursively call the searchNode() function with the root_node leftChild and node_value as the parameters.
Step 29: If the node_value is greater than the root_node data, recursively call the searchNode() function with the root_node rightChild and node_value as the parameters.
Step 30: Define the inOrderTraversal() function which takes root_node as the parameter.
Step 31: Check if the root_node is None, if yes, return.
Step 32: Stop.

Program:

```
class BSTNode:
    def __init__(self, data):
        self.leftChild = None
        self.rightChild = None
        self.data = data

def insertNode(root_node, node_value):
    if root_node.data is None:
        root_node.data = node_value
    elif node_value <= root_node.data:
        if root_node.leftChild is None:
            root_node.leftChild = BSTNode(node_value)
        else:
            insertNode(root_node.leftChild, node_value)
    else:
        if root_node.rightChild is None:
            root_node.rightChild = BSTNode(node_value)
        else:
            insertNode(root_node.rightChild, node_value)
    return "The node has been successfully inserted"

def minValueNode(bstNode):
    current = bstNode
    while current.leftChild is not None:
        current = current.leftChild
    return current

def deleteNode(root_node, node_value):
    if root_node is None:
        print(node_value, "Can't be deleted as it is not found")
        return root_node
    if node_value < root_node.data:
```

```
        root_node.leftChild = deleteNode(root_node.leftChild, node_value)
    elif node_value > root_node.data:
        root_node.rightChild = deleteNode(root_node.rightChild, node_value)
    else:
        if root_node.leftChild is None:
            temp = root_node.rightChild
            print("Element deleted successfully")
            root_node = None
            return temp

        if root_node.rightChild is None:
            temp = root_node.leftChild
            print("Element deleted successfully")
            root_node = None
            return temp

        temp = minValueNode(root_node.rightChild)
        root_node.data = temp.data
        root_node.rightChild = deleteNode(root_node.rightChild, temp.data)
    return root_node

def searchNode(root_node, node_value):
    if root_node is None:
        print("Element is not found")
        return
    elif root_node.data == node_value:
        print("The element has been found")
    elif node_value < root_node.data:
        if root_node.leftChild is not None and root_node.leftChild.data == node_value:
            print("The element has been found")
        else:
            searchNode(root_node.leftChild, node_value)
    else:
        if root_node.rightChild is not None and root_node.rightChild.data == node_value:
            print(node_value, "the element has been found")
        else:
            searchNode(root_node.rightChild, node_value)

def inOrderTraversal(root_node):
    if not root_node:
        return
    inOrderTraversal(root_node.leftChild)
    print(root_node.data)
```

```
    inOrderTraversal(root_node.rightChild)
r = BSTNode(50)
insertNode(r, 30)
insertNode(r, 20)
insertNode(r, 40)
insertNode(r, 70)
insertNode(r, 60)
insertNode(r, 80)

print("Inorder Traversal of tree:")
inOrderTraversal(r)
searchNode(r, 20)
deleteNode(r, 50)
print("Inorder traversal of tree:")
inOrderTraversal(r)
deleteNode(r,50)
```

Output:

```
Inorder Traversal of tree:
20
30
40
50
60
70
80
The element has been found
Element deleted successfully
Inorder traversal of tree:
20
30
40
60
70
80
50 Can't be deleted as it is not found
```

22.Implementations of BFS.

Algorithm:

Step 1: Choose any one node randomly, to start traversing.
Step 2: Visit its adjacent unvisited node.
Step 3: Mark it as visited in the boolean array and display it.
Step 4: Insert the visited node into the queue.
Step 5: If there is no adjacent node, remove the first node from the queue.
Step 6: Repeat the above steps until the queue is empty.

Program:

```
graph = {
   'A' : ['B','D','E','F'],
   'D':['A'],
   'B':['A','F','C'],
   'F':['B','A'],
   'C':['B'],
   'E':['A']
}
print("The given graph is:")
print(graph)
visited = []
queue = []

def bfs(visited, graph, node):

   visited.append(node)
   queue.append(node)

   while queue:
      m = queue.pop(0)
      print(m, end = " ")

      for neighbour in graph[m]:
         if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)

print("BFS traversal graph with source A is:")

bfs(visited, graph, 'A')
```

Output:

The given graph is:
{'A': ['B', 'D', 'E', 'F'], 'D': ['A'], 'B': ['A', 'F', 'C'], 'F': ['B', 'A'], 'C': ['B'], 'E': ['A']}

BFS traversal graph with source A is:
A B D E F C

23. Implementation of DFS.

Algorithm:

Step1: Start by putting any one of the graph's vertices on top of a stack.
Step2: Take the top item of the stack and add it to the visited list.
Step3: Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
Step 4: Keep repeating steps 2 and 3 until the stack is empty.

Program:

```
graph = {
  'A': ['B', 'D', 'E', 'F'],
  'D': ['A'],
  'B': ['A', 'F', 'C'],
  'F': ['B', 'A'],
  'C': ['B'],
  'E': ['A']
}
print("The given graph is:")
print(graph)

visited = set()

def dfs(visited, graph, node):
  if node not in visited:
    print(node)
    visited.add(node)
    for neighbour in graph[node]:
      dfs(visited, graph, neighbour)

print("DFS traversal graph with source A is:")
dfs(visited, graph, 'A')
```

Output:

```
The given graph is:
{'A': ['B', 'D', 'E', 'F'], 'D': ['A'], 'B': ['A', 'F', 'C'], 'F': ['B', 'A'], 'C': ['B'], 'E': ['A']}
DFS traversal graph with source A is:
A
B
F
C
D
E
```

24. Implement Hash functions.

Algorithm:

Step1: Create hash table.
Step2: Assign for loop condition to check the length of hash table.
Step3: Assign for loop to perform the operation.
Step4: Creating Hash table as a nested list.
Step5: Insert Function to add .values to the hash table. Print the table.
Step6: stop

Program:

```
def display_hash(hashTable):
   for i in range(len(hashTable)):
      print("\n")
      print(i, end = " ")

      for j in hashTable[i]:

         print("-->", end = " ")

         print(j, end = " ")

print()
HashTable = [[] for _ in range(10)]

HashTable = [[] for _ in range(10)]


def Hashing(keyvalue):
   return keyvalue % len(HashTable)

def insert(Hashtable, keyvalue, value):
   hash_key = Hashing(keyvalue)
   Hashtable[hash_key].append(value)

insert(HashTable, 10, 'Bangalore')
insert(HashTable, 25, 'Mumbai')
insert(HashTable, 20, 'Mathura')
insert(HashTable, 9, 'Delhi')
insert(HashTable, 21, 'Punjab')
insert(HashTable, 21, 'Noida')
display_hash(HashTable)
```

Output:

0 --> Bangalore --> Mathura

1 --> Punjab --> Noida

2

3

4

5 --> Mumbai

6

7

8

9 --> Delhi