Sri Shanmugha™

College of Engineering And Technology

(Approved by AICTE, New Delhi & Affiliated To Anna University)

Connects Life & Learning

AD8261-DATA STRUCTURES DESIGN LABORATORY

NAME : _____

REGISTER NO. : _ YEAR/SEM

: _____

SECTION : _____

BRANCH : _____

(Approved By AICTE, Accredited by NAAC, & Affiliated to Anna University)

Tiruchengode – Sankari Mani Rd, Pullipalayam, Morur (PO), Sankari (Tk), Salem 637304.

RECORD NOTE BOOK

REGNO. _____

Certified that this is a bonafide observation of Practical work done by Mr/Ms/Mrs……of the…… Semester…… ..... Branch during the Academic year……in the… ..... laboratory.

Staff–in–Charge **Head of the Department**

Internal Examiner **External Examine**

LIST OF EXPERIMENTS

S.No	NAME OF THE EXPERIMENT
1.	Implement simple ADTs as Python classes
2.	Implement recursive algorithms in Python
3.	Implement List ADT using Python arrays
4.	Linked list implementations of List
5.	Implementation of Stack and Queue ADTs
6.	Applications of List, Stack and Queue ADTs
7.	Implementation of sorting and searching algorithms
8.	Implementation of Hash tables
9.	Tree representation and traversal algorithms
10.	Implementation of Binary Search Trees
11.	Implementation of Heaps
12.	Graph representation and Traversal algorithms
13.	Implementation of single source shortest path algorithm
14.	Implementation of minimum spanning tree

CONTENTS

COURSE OBJECTIVES AND OUTCOMES

OBJECTIVES:

- ✓To implement ADTs in Python
- ✓To design and implement linear data structures – lists, stacks, and queues
- ✓To implement sorting, searching and hashing algorithm
- ✓To solve problems using tree and graph structures

OUTCOMES:

At the end of the course, the students will be able

- ✓Design and implement various mobile applications using emulators.
- ✓Implement ADTs as Python classes
- ✓Design, implement, and analyse linear data structures, such as lists, queues, and stacks, according to the needs of different applications
- ✓Design, implement, and analyse efficient tree structures to meet requirements such as searching, indexing, and sorting
- ✓Model problems as graph problems and implement efficient graph algorithms to solve theme.

EX:NO:1 **SIMPLE ADTS AS PYTHON CLASSES**

DATE

AIM:

To write a Python program that appends, deletes and displays elements of a list using classes.

ALGORITHM:

1. Create a class and using a constructor initialize values of that class.
2. Create methods for adding, removing and displaying elements of the list and return the respective values.
3. Create an object for the class.
4. Using the object, call the respective function depending on the choice taken from the user.
5. Print the final list.
6. Exit

PROGRAM:

```
class check():
    def __init__(self):
        self.n=[]
    def add(self,a):
        return self.n.append(a)
    def remove(self,b):
        self.n.remove(b)
    def dis(self):
        return (self.n)
obj=check()
choice=1
while choice!=0:
    print("0. Exit")
    print("1. Add")
    print("2. Delete")
    print("3. Display")
    choice=int(input("Enter choice: "))
    if choice==1:
        n=int(input("Enter number to append: "))
        obj.add(n)
        print("List: ",obj.dis())
     elif choice==2:
        n=int(input("Enter number to remove: "))
        obj.remove(n)
        print("List: ",obj.dis())
     elif choice==3:
        print("List: ",obj.dis())
    elif choice==0:
        print("Exiting!")
    else:
        print("Invalid choice!!")
 print()
```

OUTPUT:

```
Case 1:
0. Exit
1. Add
2. Delete
3. Display
Enter choice: 1
Enter number to append: 23
List:  [23]
0. Exit
1. Add
2. Delete
3. Display
Enter choice: 1
Enter number to append: 45
List:  [23, 45]
0. Exit
1. Add
2. Delete
3. Display
Enter choice: 1
Enter number to append: 56
List: [23, 45, 56]
0. Exit
1. Add
2. Delete
3. Display
Enter choice: 2
Enter number to remove: 45
List: [23, 56]
0. Exit
1. Add
2. Delete
3. Display
Enter choice: 0
Exiting!
Case 2:
0. Exit
1. Add
2. Delete
3. Display
Enter choice: 1
Enter number to append: 10
List: [10]
0. Exit
1. Add
2. Delete
```

```
3. Display
Enter choice: 1
Enter number to append: 7
List: [10, 7]
0. Exit
1. Add
2. Delete
3. Display
Enter choice: 0
Exiting!
```

RESULT:

Thus, the Python program that appends, deletes and displays elements of a list using classes has been implemented successfully.

EX:NO:2 **RECURSIVE ALGORITHMS IN PYTHON**

DATE:

AIM:

To write a python program takes a number and determines the factorial of the number using recursion.

ALGORITHM:

1. Take a number from the user and store it in a variable.
2. Pass the number as an argument to a recursive factorial function.
3. Define the base condition as the number to be lesser than or equal to 1 and return 1 if it is.
4. Otherwise call the function recursively with the number minus 1 multiplied by the number itself.
5. Then return the result and print the factorial of the number.
6. Exit.

PROGRAM:

```
def factorial(n):
    if (n <= 1):
        return 1
    else:
        return(n*factorial(n-1))
n = int (input ("Enter number:"))
print("Factorial:")
print(factorial(n))
```

OUTPUT:

Case 1:
Enter number:5
Factorial:
120

Case 2:
Enter number:9
Factorial:
362880

RESULT:

Thus, the Python program that takes a number and determines the factorial of the number using recursion has been implemented successfully.

EX:NO:3 **LIST ADT USING PYTHON ARRAYS**

DATE:

AIM

To write a Python program for creation and insertion to implement list using an array.

ALGORITHM

1: Start.
2: Declare the necessary functions for implementation. Step 3: Get the input from the user and store it an array.
3: In Insertion, half of the elements to be shifted upwards and in deletion half of the elements to be shifted downwards.
4: Display the output using an array.
5: Stop.

PROGRAM:

```
import array
arr = array.array('i', [1, 2, 3])
print ("The new created array is : ",end=" ")
for i in range (0, 3):
    print (arr[i], end=" ")
print("\r")
arr.append(4);
print("The appended array is : ", end="")
for i in range (0, 4):
    print (arr[i], end=" ")
arr.insert(2, 5)
print("\r")
print ("The array after insertion is : ", end="")
for i in range (0, 5):
    print (arr[i], end=" ")
```

OUTPUT:

The new created array is: 1 2 3

The appended array is: 1 2 3 4

The array after insertion is: 1 2 5 3 4

RESULT:

Thus, the Python program for creation and insertion to implement list using an array has been executed successfully.

EX:NO:4 **LINKED LIST IMPLEMENTATIONS OF LIST**

DATE:

AIM

To write a Python program to implement list using linked list.

ALGORITHM

1. A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node. A singly linked list travels one way.
2. The beginning of the linked list is stored in a "start" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on.
3. The last node in the list has its next field set to NULL to mark the end of the list.
4. The basic operations in a single linked list are: Creation, Insertion, Deletion, Traversing.

PROGRAM:

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


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

    def get_node(self, index):
        current = self.head
        for i in range(index):
            if current is None:
                return None
            current = current.next
        return current

    def get_prev_node(self, ref_node):
        current = self.head
        while (current and current.next != ref_node):
            current = current.next
        return current

    def insert_after(self, ref_node, new_node):
        new_node.next = ref_node.next
        ref_node.next = new_node

    def insert_before(self, ref_node, new_node):
        prev_node = self.get_prev_node(ref_node)
        self.insert_after(prev_node, new_node)
```

```
    def insert_at_beg(self, new_node):
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head = new_node

    def insert_at_end(self, new_node):
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node

    def remove(self, node):
        prev_node = self.get_prev_node(node)
        if prev_node is None:
            self.head = self.head.next
        else:
            prev_node.next = node.next

    def display(self):
        current = self.head
        while current:
            print(current.data, end = ' ')
            current = current.next

a_llist = LinkedList()
print('Menu')
print('insert <data> after <index>')
print('insert <data> before <index>')
print('insert <data> at beg')
print('insert <data> at end')
print('remove <index>')
print('quit')

while True:
    print('The list: ', end = '')
    a_llist.display()
    print()
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()

    if operation == 'insert':
        data = int(do[1])
        position = do[3].strip().lower()
        new_node = Node(data)
```

```
        suboperation = do[2].strip().lower()
        if suboperation == 'at':
            if position == 'beg':
                a_llist.insert_at_beg(new_node)
            elif position == 'end':
                a_llist.insert_at_end(new_node)
        else:
            index = int(position)
            ref_node = a_llist.get_node(index)
            if ref_node is None:
                print('No such index.')
                continue
            if suboperation == 'after':
                a_llist.insert_after(ref_node, new_node)
            elif suboperation == 'before':
                a_llist.insert_before(ref_node, new_node)

    elif operation == 'remove':
        index = int(do[1])
        node = a_llist.get_node(index)
        if node is None:
            print('No such index.')
            continue
        a_llist.remove(node)

    elif operation == 'quit':
        break
```

OUTPUT:

Case 1:

Menu

insert <data> after <index>

insert <data> before <index>

insert <data> at beg

insert <data> at end

remove <index>

quit

The list:

What would you like to do? insert 7 at beg

The list: 7

What would you like to do? insert 3 at end

The list: 7 3

What would you like to do? insert 1 after 0

The list: 7 1 3

What would you like to do? insert 9 before 2

The list: 7 1 9 3

What would you like to do? remove 2

The list: 7 1 3

What would you like to do? insert 12 at end

The list: 7 1 3 12
What would you like to do? remove 0
The list: 1 3 12
What would you like to do? quit
Case 2:
Menu
insert <data> after <index>
insert <data> before <index>
insert <data> at beg
insert <data> at end
remove <index>
quit
The list:
What would you like to do? insert 5 after 0
No such index.
The list:
What would you like to do? insert 3 at end
The list: 3
What would you like to do? insert 1 after 0
The list: 3 1
What would you like to do? insert 2 before 1
The list: 3 2 1
What would you like to do? insert 0 at end
The list: 3 2 1 0
What would you like to do? remove 3
The list: 3 2 1
What would you like to do? remove 2
The list: 3 2
What would you like to do? quit

RESULT:
Thus, the Python program to implement list using linked list has been executed successfully.

EX:NO:5 **STACK AND QUEUE ADTS**

DATE:

A. IMPLEMENTATION OF STACK:

AIM:

To write a python program creates a stack and allows the user to perform push and pop operations on it.

ALGORITHM:

1. Create a class Node with instance variables data and next.
2. Create a class Stack with instance variable head.
3. The variable head points to the first element in the linked list.
4. Define methods push and pop inside the class Stack.
5. The method push adds a node at the front of the linked list.
6. The method pop returns the data of the node at the front of the linked list and removes the node. It returns None if there are no nodes.
7. Create an instance of Stack and present a menu to the user to perform operations on the stack.

PROGRAM:

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

    def push(self, data):
        if self.head is None:
            self.head = Node(data)
        else:
            new_node = Node(data)
            new_node.next = self.head
            self.head = new_node
     def pop(self):
        if self.head is None:
            return None
        else:
            popped = self.head.data
            self.head = self.head.next
            return popped
 a_stack = Stack()
while True:
    print('push <value>')
    print('pop')
    print('quit')
    do = input('What would you like to do? ').split()
     operation = do[0].strip().lower()
    if operation == 'push':
```

```
        a_stack.push(int(do[1]))
    elif operation == 'pop':
        popped = a_stack.pop()
        if popped is None:
            print('Stack is empty.')
        else:
            print('Popped value: ', int(popped))
    elif operation == 'quit':
        break
```

OUTPUT:

Case 1:
push <value>
pop
quit
What would you like to do? push 15
push <value>
pop
quit
What would you like to do? push 3
push <value>
pop
quit
What would you like to do? pop
Popped value: 3
push <value>
pop
quit
What would you like to do? pop
Popped value: 15
push <value>
pop
quit
What would you like to do? pop
Stack is empty.
push <value>
pop
quit
What would you like to do? quit

RESULT:

Thus the python program to creates a stack and allows the user to perform push and pop operations on it has been executes successfully.

.

B. IMPLEMENTATION OF QUEUE:

AIM:

To write a python program creates a queue and allows the user to perform enqueue and dequeue operations on it.

ALGORITHM:

1. Create a class Node with instance variables data and next.
2. Create a class Queue with instance variables head and last.
3. The variable head points to the first element in the linked list while last points to the last element.
4. Define methods enqueue and dequeue inside the class Queue.
5. The method enqueue adds a node at the end of the linked list.
6. The method dequeue returns the data of the node at the front of the linked list and removes the node. It returns None if there are no nodes.
7. Create an instance of Queue and present a menu to the user to perform operations on the queue.

PROGRAM:

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

class Queue:
    def __init__(self):
        self.head = None
        self.last = None

    def enqueue(self, data):
        if self.last is None:
            self.head = Node(data)
            self.last = self.head
        else:
            self.last.next = Node(data)
            self.last = self.last.next

    def dequeue(self):
        if self.head is None:
            return None
        else:
            to_return = self.head.data
            self.head = self.head.next
            return to_return

a_queue = Queue()
while True:
    print('enqueue <value>')
    print('dequeue')
    print('quit')
```

```
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'enqueue':
        a_queue.enqueue(int(do[1]))
    elif operation == 'dequeue':
        dequeued = a_queue.dequeue()
        if dequeued is None:
            print('Queue is empty.')
        else:
            print('Dequeued element: ', int(dequeued))
    elif operation == 'quit':
        break
```

OUTPUT:

```
Case 1:
enqueue <value>
dequeue
quit
What would you like to do? enqueue 3
enqueue <value>
dequeue
quit
What would you like to do? enqueue 4
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Dequeued element:  3
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Dequeued element:  4
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Queue is empty.
enqueue <value>
dequeue
quit
What would you like to do? quit

Case 2:
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Queue is empty.
```

enqueue <value>
dequeue
quit
What would you like to do? enqueue 5
enqueue <value>
dequeue
quit
What would you like to do? dequeue
Dequeued element: 5
enqueue <value>
dequeue
quit
What would you like to do? quit

RESULT:
Thus, the python program to creates a queue and allows the user to perform enqueue and dequeue operations on it has been executes successfully.

EX:NO:6 **APPLICATIONS OF LIST, STACK AND QUEUE ADTS**

DATE:

A. **APPLICATIONS OF LIST ADT**

AIM:

To write a Python program for implementation of polynomial ADT.

ALGORITHM:

1: Start the program
2: Get the coefficients and powers for the two polynomials to be added.
3: Add the coefficients of the respective powers.
4: Display the added polynomial. Step5: Terminate the program.

PROGRAM:

```
def add(A, B, m, n):

    size = max(m, n);
    sum = [0 for i in range(size)]
    for i in range(0, m, 1):
        sum[i] = A[i]

    for i in range(n):
        sum[i] += B[i]

    return sum

def printPoly(poly, n):
    for i in range(n):
        print(poly[i], end = "")
        if (i != 0):
            print("x^", i, end = "")
        if (i != n - 1):
            print(" + ", end = "")
if __name__ == '__main__':

    # The following array represents
    # polynomial 5 + 10x^2 + 6x^3
    A = [5, 0, 10, 6]

    # The following array represents
    # polynomial 1 + 2x + 4x^2
    B = [1, 2, 4]
    m = len(A)
    n = len(B)

    print("First polynomial is")
    printPoly(A, m)
```

```
print("\n", end = "")
print("Second polynomial is")
printPoly(B, n)
print("\n", end = "")
sum = add(A, B, m, n)
size = max(m, n)

print("sum polynomial is")
printPoly(sum, size)
```

OUTPUT:

First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Sum polynomial is
6 + 2x^1 + 14x^2 + 6x^3

RESULT:

Thus, the Python program for implementation of polynomial ADT has been executed successfully.

B. **APPLICATIONS OF STACK ADT**

AIM:

To write a Python program to Check if Expression is Correctly Parenthesized
.

ALGORITHM:

1. Create a class Stack with instance variable items initialized to an empty list.
2. Define methods push, pop and is_empty inside the class Stack.
3. The method push appends data to items.
4. The method pop pops the first element in items.
5. The method is_empty returns True only if items is empty.
6. Prompt the user for an expression.
7. Iterate through the characters of the expression and push to the stack if an open parenthesis is encountered and pop if a close parenthesis is encountered.
8. Determine whether the expression has balanced parentheses.

PROGRAM:

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, data):
        self.items.append(data)

    def pop(self):
        return self.items.pop()

s = Stack()
exp = input('Please enter the expression: ')

for c in exp:
    if c == '(':
        s.push(1)
    elif c == ')':
        if s.is_empty():
            is_balanced = False
            break
        s.pop()
else:
    if s.is_empty():
        is_balanced = True
    else:
        is_balanced = False

if is_balanced:
    print('Expression is correctly parenthesized.')
```

```
else:
    print('Expression is not correctly parenthesized.')
```

OUTPUT:

Case 1:
Please enter the expression: (3 + 4 * (1 + (2))/(7 * (8 + 9)))
Expression is correctly parenthesized.

Case 2:
Please enter the expression: (a + b))(3)
Expression is not correctly parenthesized.

Case 3:
Please enter the expression: (4 + (3 * 2)
Expression is not correctly parenthesized

RESULT:

Thus, the Python program for implementation of whether Expression is Correctly Parenthesized has been executed successfully.

C. **APPLICATIONS OF QUEUE ADT**

AIM:

To write a Python program that creates a dequeue and allows the user to perform append and pop operations on it from both sides.

.

ALGORITHM:

1. Create a class Dequeue with instance variable items initialized to an empty list.
2. Define methods append, append_left, pop, pop_left and is_empty inside the class Dequeue.
3. The method append appends data to items from the right.
4. The method append_left appends data to items from the left.
5. The method pop pops from the right from items.
6. The method pop_left pops from the left from items.
7. The method is_empty returns True only if items is empty.

PROGRAM:

```
class Dequeue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def append(self, data):
        self.items.append(data)

    def append_left(self, data):
        self.items.insert(0, data)

    def pop(self):
        return self.items.pop()

    def pop_left(self):
        return self.items.pop(0)


q = Dequeue()
print('Menu')
print('append <value>')
print('appendleft <value>')
print('pop')
print('popleft')
print('quit')
while True:
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'append':
```

```
        q.append(int(do[1]))
    elif operation == 'appendleft':
        q.append_left(int(do[1]))
    elif operation == 'pop':
        if q.is_empty():
            print('Dequeue is empty.')
        else:
            print('Popped value from right: ', q.pop())
    elif operation == 'popleft':
        if q.is_empty():
            print('Dequeue is empty.')
        else:
            print('Popped value from left: ', q.pop_left())
    elif operation == 'quit':
        break
```

OUTPUT:

Case 1:
Menu
append <value>
append left <value>
pop
pop left
quit
What would you like to do? append 3
What would you like to do? append 4
What would you like to do? append left 2
What would you like to do? append left 1
What would you like to do? pop
Popped value from right: 4
What would you like to do? pop left
Popped value from left: 1
What would you like to do? pop
Popped value from right: 3
What would you like to do? pop left
Popped value from left: 2
What would you like to do? pop
Dequeue is empty.
What would you like to do? quit

Case 2:
Menu
append <value>
append left <value>
pop
pop left
quit
What would you like to do? append 1
What would you like to do? append 2

What would you like to do? pop
Popped value from right: 2
What would you like to do? pop
Popped value from right: 1
What would you like to do? append left 1
What would you like to do? append left 2
What would you like to do? pop
Popped value from right: 1
What would you like to do? pop
Popped value from right: 2
What would you like to do? append 1
What would you like to do? append 2
What would you like to do? popleft
Popped value from left: 1
What would you like to do? popleft
Popped value from left: 2
What would you like to do? quit

RESULT:

Thus, the Python program for implementation of dequeue and allows the user to perform append and pop operations on it from both sides. has been executed successfully.

EX:NO:7 **SORTING AND SEARCHING ALGORITHMS**

DATE:

A. SORTING ALGORITHM (INSERTION SORT):

AIM:

To Perform Insertion Sorting in Python Programming.

ALGORITHM:

Step 1: Start
Step 2: Define list of elements(alist)
Step 3: Take first element find its appropriate position and insert them
Step 4: Repeat till every element is sorted
Step 5: Print the list of elements
Step 6: Stop

PROGRAM:

```
def insertionSort(lst):
for index in range(1, len(lst)):
currentvalue = lst[index]
position = index
while position > 0 and lst[position - 1] > currentvalue:
lst[position] = lst[position - 1]
position = position - 1
lst[position] = currentvalue
lst = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertionSort(lst)
print(lst)
```

OUTPUT:

17,20,26,31,44,54,55,77,93

RESULT:

Thus, the python program to perform Insertion Sort is created and executed successfully.

B. SEARCHING ALGORITHM (LINEAR SEARCH)

AIM:

To Perform a Linear Search in Python Programming

ALGORITHM:

Step 1: Start
Step 2: Define a list of elements list_of_elements[]
Step 3: Get the element to be checked from the user(x)
Step 4: Compare the elements with each element in the list
Step 5: If found print found and print index number
Step 6: Else print element not found
Step 6: Stop

PROGRAM:

```
list_of_elements = [4, 2, 8, 9, 3, 7]
x = int(input("Enter number to search: "))
found = False
for i in range(len(list_of_elements)):
if(list_of_elements[i] == x):
found = True
print("%d found at %dth position"%(x,i))
break
if(found == False):
print("%d is not in list"%x)
```

OUTPUT:

```
Enter element to search:4
4 is found at 0th position
```

RESULT:

Thus, the python program to perform Linear Search is created and executed successfully.

EX:NO:8 **HASH TABLE**

DATE:

AIM:

To write a python program to implement the concept of hashing using separate chaining.

ALGORITHM:

1: Start
2: Create Table size
3: Create hash function
4: To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.
5: Display hash entry.
6: Stop

PROGRAM:

```
def display_hash(hashTable):
    for i in range(len(hashTable)):
       print(i, end = " ")
       for j in hashTable[i]:
          print("-->", end = " ")
          print(j, end = " ")
       print()
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, 'Allahabad')
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 --> Allahabad --> Mathura
1 --> Punjab --> Noida
2
3
4
5 --> Mumbai
6
7
8
9 --> Delhi
```

RESULT:

Thus, the python program to implement the concept of hashing using separate chaining. has been implemented successfully.

EX:NO:9 TREE REPRESENTATION AND TRAVERSAL ALGORITHM

DATE:

AIM:

To write a python program to implement the tree representation and traversal algorithm

ALGORITHM:

1. The left sub tree of a node contains smaller nodes than a root node.
2. The right sub tree of a node contains greater nodes than a root node.
3. Both the left and right sub trees must also be binary search trees.
4. There are three types of tree traversals: Preorder, Postorder, and Inorder.

Pre-order traversal
Algorithm:
1. Visit the root (we will print it when we visit to show the order of visiting)
2. Traverse the left subtree in pre-order
3. Traverse the right subtree in pre-order

In-order traversal
Visit the root node in between the left and right node (in)
Algorithm:
1. Traverse the left subtree in in-order
2. Visit the root (we will print it when we visit to show the order of visiting)
3. Traverse the right subtree in in-order

Post-order traversal
Visit the root node after (post) visiting the left and right subtree.
Algorithm:
1. Traverse the left subtree in in-order
2. Traverse the right subtree in in-order
3. Visit the root (we will print it when we visit to show the order of visiting)

PROGRAM:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
def printInorder(root):

    if root:
         printInorder(root.left)
        print(root.val),
        printInorder(root.right)
```

```
 def printPostorder(root):

    if root:
         printPostorder(root.left)
        printPostorder(root.right)
         print(root.val),
def printPreorder(root):

    if root:
         print(root.val),
         printPreorder(root.left)
         printPreorder(root.right)
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print "Preorder traversal of binary tree is"
printPreorder(root)

print "\nInorder traversal of binary tree is"
printInorder(root)

print "\nPostorder traversal of binary tree is"
printPostorder(root)
```

OUTPUT:

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

RESULT:

Thus, the python program to implement the concept of tree representation and traversal algorithm. has been implemented successfully.

EX:NO:10 **BINARY SEARCH TREE**

DATE:

AIM:

To write a python program creates a binary search tree and presents a menu to the user to perform insertion, deletion and inorder traversal operations.

ALGORITHM:

1. Create a class BSTNode with instance variables key, left, right and parent.
2. Define methods insert, inorder, replace_node_of_parent, find_min, remove and search in BSTNode.
3. The method insert takes a node as argument and inserts that node in the BST with the BSTNode object as root.
4. The method inorder displays the inorder traversal of the BST with the BSTNode object as root.
5. The method replace_node_of_parent takes a node as argument and replaces the current object in the BST with the node.
6. The method find_min finds the the left-most node in the BST with the BSTNode object as root.
7. The method remove removes the current BSTNode object from the BST.
8. The method search takes a key as argument and returns the node with that key in the BST with the BSTNode object as root.
9. Create a class BSTree with instance variable root.
10. Define methods inorder, add, remove and search in BSTree.
11. The method inorder calls the inorder method of the root node.

PROGRAM:

```
class BSTNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None

    def insert(self, node):
        if self.key > node.key:
            if self.left is None:
                self.left = node
                node.parent = self
            else:
                self.left.insert(node)
        elif self.key < node.key:
            if self.right is None:
                self.right = node
                node.parent = self
            else:
                self.right.insert(node)
```

```
def inorder(self):
    if self.left is not None:
        self.left.inorder()
    print(self.key, end=' ')
    if self.right is not None:
        self.right.inorder()

def replace_node_of_parent(self, new_node):
    if self.parent is not None:
        if new_node is not None:
            new_node.parent = self.parent
        if self.parent.left == self:
            self.parent.left = new_node
        elif self.parent.right == self:
            self.parent.right = new_node
    else:
        self.key = new_node.key
        self.left = new_node.left
        self.right = new_node.right
        if new_node.left is not None:
            new_node.left.parent = self
        if new_node.right is not None:
            new_node.right.parent = self

def find_min(self):
    current = self
    while current.left is not None:
        current = current.left
    return current

def remove(self):
    if (self.left is not None and self.right is not None):
        successor = self.right.find_min()
        self.key = successor.key
        successor.remove()
    elif self.left is not None:
        self.replace_node_of_parent(self.left)
    elif self.right is not None:
        self.replace_node_of_parent(self.right)
    else:
        self.replace_node_of_parent(None)

def search(self, key):
    if self.key > key:
        if self.left is not None:
            return self.left.search(key)
        else:
            return None
    elif self.key < key:
```

```
            if self.right is not None:
                return self.right.search(key)
            else:
                return None
        return self


class BSTree:
    def __init__(self):
        self.root = None

    def inorder(self):
        if self.root is not None:
            self.root.inorder()

    def add(self, key):
        new_node = BSTNode(key)
        if self.root is None:
            self.root = new_node
        else:
            self.root.insert(new_node)

    def remove(self, key):
        to_remove = self.search(key)
        if (self.root == to_remove
            and self.root.left is None and self.root.right is None):
            self.root = None
        else:
            to_remove.remove()

    def search(self, key):
        if self.root is not None:
            return self.root.search(key)


bstree = BSTree()

print('Menu (this assumes no duplicate keys)')
print('add <key>')
print('remove <key>')
print('inorder')
print('quit')

while True:
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'add':
        key = int(do[1])
        bstree.add(key)
```

```
    elif operation == 'remove':
        key = int(do[1])
        bstree.remove(key)
    elif operation == 'inorder':
        print('Inorder traversal: ', end='')
        bstree.inorder()
        print()
    elif operation == 'quit':
        break
```

OUTPUT:

Case 1:

```
Menu (this assumes no duplicate keys)
add <key>
remove <key>
inorder
quit
What would you like to do? add 5
What would you like to do? add 1
What would you like to do? add 10
What would you like to do? add 7
What would you like to do? add 3
What would you like to do? inorder
Inorder traversal: 1 3 5 7 10
What would you like to do? remove 3
What would you like to do? remove 7
What would you like to do? inorder
Inorder traversal: 1 5 10
What would you like to do? remove 5
What would you like to do? inorder
Inorder traversal: 1 10
What would you like to do? quit
```

Case 2:

```
Menu (this assumes no duplicate keys)
add <key>
remove <key>
inorder
quit
What would you like to do? add 2
What would you like to do? add 8
What would you like to do? inorder
Inorder traversal: 2 8
What would you like to do? add 5
What would you like to do? inorder
Inorder traversal: 2 5 8
What would you like to do? remove 2
What would you like to do? remove 8
```

What would you like to do? inorder
Inorder traversal: 5
What would you like to do? remove 5
What would you like to do? inorder
Inorder traversal:
What would you like to do? quit

RESULT:

Thus, the python program to implement the concept of binary search tree. has been implemented successfully.

EX:NO:11 **HEAP IMPLEMENTATION**

DATE:

AIM:

To write a python program creates a binary max-heap and presents a menu to the user to perform various operations on it.

ALGORITHM:

1. Create a class BinaryHeap with an instance variable items set to an empty list. This empty list is used to store the binary heap.
2. Define methods size, parent, left, right, get, get_max, extract_max, max_heapify, swap and insert.
3. The method size returns the number of elements in the heap.
4. The method parent takes an index as argument and returns the index of the parent.
5. The method left takes an index as argument and returns the index of its left child.
6. The method right takes an index as argument and returns the index of its right child.
7. The method get takes an index as argument and returns the key at the index.
8. The method get_max returns the maximum element in the heap by returning the first element in the list items.
9. The method extract_max returns the the maximum element in the heap and removes it.
10. The method max_heapify takes an index as argument and modifies the heap structure at and below the node at this index to make it satisfy the heap property.
11. The method swap takes two indexes as arguments and swaps the corresponding elements in the heap.
12. The method insert takes a key as argument and adds that key to the heap.

PROGRAM:

```
class BinaryHeap:
    def __init__(self):
        self.items = []

    def size(self):
        return len(self.items)

    def parent(self, i):
        return (i - 1)//2

    def left(self, i):
        return 2*i + 1

    def right(self, i):
        return 2*i + 2

    def get(self, i):
        return self.items[i]
```

```
    def get_max(self):
        if self.size() == 0:
            return None
        return self.items[0]

    def extract_max(self):
        if self.size() == 0:
            return None
        largest = self.get_max()
        self.items[0] = self.items[-1]
        del self.items[-1]
        self.max_heapify(0)
        return largest

    def max_heapify(self, i):
        l = self.left(i)
        r = self.right(i)
        if (l <= self.size() - 1 and self.get(l) > self.get(i)):
            largest = l
        else:
            largest = i
        if (r <= self.size() - 1 and self.get(r) > self.get(largest)):
            largest = r
        if (largest != i):
            self.swap(largest, i)
            self.max_heapify(largest)

    def swap(self, i, j):
        self.items[i], self.items[j] = self.items[j], self.items[i]

    def insert(self, key):
        index = self.size()
        self.items.append(key)

        while (index != 0):
            p = self.parent(index)
            if self.get(p) < self.get(index):
                self.swap(p, index)
            index = p

 bheap = BinaryHeap()

print('Menu')
print('insert <data>')
print('max get')
print('max extract')
print('quit')

while True:
    do = input('What would you like to do? ').split()
```

```
    operation = do[0].strip().lower()
    if operation == 'insert':
        data = int(do[1])
        bheap.insert(data)
    elif operation == 'max':
        suboperation = do[1].strip().lower()
        if suboperation == 'get':
            print('Maximum value: {}'.format(bheap.get_max()))
        elif suboperation == 'extract':
            print('Maximum value removed: {}'.format(bheap.extract_max()))

    elif operation == 'quit':
        break
```

OUTPUT:

Case 1:
Menu
insert <data>
max get
max extract
quit
What would you like to do? insert 5
What would you like to do? insert 3
What would you like to do? insert -3
What would you like to do? insert 10
What would you like to do? insert 8
What would you like to do? max get
Maximum value: 10
What would you like to do? max extract
Maximum value removed: 10
What would you like to do? max extract
Maximum value removed: 8
What would you like to do? max extract
Maximum value removed: 5
What would you like to do? max extract
Maximum value removed: 3
What would you like to do? max get
Maximum value: -3
What would you like to do? quit

Case 2:
Menu
insert <data>
max get
max extract
quit
What would you like to do? insert 3

What would you like to do? insert 11
What would you like to do? insert 5
What would you like to do? max extract
Maximum value removed: 11
What would you like to do? max get
Maximum value: 5
What would you like to do? max extract
Maximum value removed: 5
What would you like to do? insert 15
What would you like to do? max get
Maximum value: 15
What would you like to do? quit

RESULT:
To write a python program that creates a binary max-heap and presents a menu to the user to perform various operations on it. has been implemented successfully.

EX:NO:12 GRAPH REPRESENTATION AND TRAVERSAL ALGORITHM

DATE:

AIM:

To write a Python program to implement depth first search. and breadth first search.

ALGORITHM:

1. Take the graph as input and find the adjacency list
2. Start at a random vertex and visit all nodes using depth first search (DFS) and then breadth first search (BFS).
3. Use stack for DFS and queue for BFS.

PROGRAM:

Depth First Search Program

```
from collections import defaultdict

class Graph:

  def __init__(self):
      self.graph = defaultdict(list)

  def addEdge(self, u, v):
      self.graph[u].append(v)

   def DFSUtil(self, v, visited):

      visited.add(v)
      print(v, end=' ')

      for neighbour in self.graph[v]:
         if neighbour not in visited:
            self.DFSUtil(neighbour, visited)

  def DFS(self, v):

      visited = set()
      self.DFSUtil(v, visited)

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
```

```
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

OUTPUT:

```
Depth First Traversal (starting from vertex 2)
2 0 1 9 3
```

Breadth First Search Program;

```
from collections import defaultdict
class Graph:

    def __init__(self):

        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)

    def BFS(self, s):
         visited = [False] * (max(self.graph) + 1)
         queue = []
         queue.append(s)
        visited[s] = True

        while queue:
             s = queue.pop(0)
            print (s, end = " ")
           for i in self.graph[s]:
               if visited[i] == False:
                   queue.append(i)
                   visited[i] = True
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                  " (starting from vertex 2)")
g.BFS(2)
```

OUTPUT:

Breadth First Traversal (starting from vertex 2)
2 0 3 1

RESULT;

Thus, the Python program to implement depth first search. and breadth first search has been executed successfully.

EX:NO:13 SINGLE SOURCE SHORTEST PATH ALGORITHM

DATE:

AIM:

To write aPython program to implement Dijkstra's Shortest Path algorithm on a graph.

ALGORITHM:

1. Create classes for Graph and Vertex.
2. Create a function dijkstra that takes a Graph object and a source vertex as arguments.
3. The function begins by creating a set unvisited and adding all the vertices in the graph to it.
4. A dictionary distance is created with keys as the vertices in the graph and their value all set to infinity.
5. distance[source] is set to 0.
6. The algorithm proceeds by finding the vertex that has the minimum distance in the set unvisited.
7. It then removes this vertex from the set unvisited.
8. Then all the neighbours of this vertex that have not been visited yet have their distances updated.
9. The above steps repeat until the set unvisited becomes empty.
10. The dictionary distance is returned.
11. This algorithm works for both undirected and directed graphs

PROGRAM:

```
class Graph:
    def __init__(self):
        # dictionary containing keys that map to the corresponding vertex object
        self.vertices = {}

    def add_vertex(self, key):
        """Add a vertex with the given key to the graph."""
        vertex = Vertex(key)
        self.vertices[key] = vertex

    def get_vertex(self, key):
        """Return vertex object with the corresponding key."""
        return self.vertices[key]

    def __contains__(self, key):
        return key in self.vertices

    def add_edge(self, src_key, dest_key, weight=1):
        """Add edge from src_key to dest_key with given weight."""
        self.vertices[src_key].add_neighbour(self.vertices[dest_key], weight)

    def does_edge_exist(self, src_key, dest_key):
        """Return True if there is an edge from src_key to dest_key."""
        return self.vertices[src_key].does_it_point_to(self.vertices[dest_key])
```

```
    def __iter__(self):
        return iter(self.vertices.values())

class Vertex:
    def __init__(self, key):
        self.key = key
        self.points_to = {}

    def get_key(self):
        """Return key corresponding to this vertex object."""
        return self.key

    def add_neighbour(self, dest, weight):
        """Make this vertex point to dest with given edge weight."""
        self.points_to[dest] = weight

    def get_neighbours(self):
        """Return all vertices pointed to by this vertex."""
        return self.points_to.keys()

    def get_weight(self, dest):
        """Get weight of edge from this vertex to dest."""
        return self.points_to[dest]

    def does_it_point_to(self, dest):
        """Return True if this vertex points to dest."""
        return dest in self.points_to

def dijkstra(g, source):
    """Return distance where distance[v] is min distance from source to v.

    This will return a dictionary distance.

    g is a Graph object.
    source is a Vertex object in g.
    """
    unvisited = set(g)
    distance = dict.fromkeys(g, float('inf'))
    distance[source] = 0

    while unvisited != set():
        # find vertex with minimum distance
        closest = min(unvisited, key=lambda v: distance[v])

        # mark as visited
        unvisited.remove(closest)

        # update distances
        for neighbour in closest.get_neighbours():
```

```
            if neighbour in unvisited:
                new_distance = distance[closest] + closest.get_weight(neighbour)
                if distance[neighbour] > new_distance:
                    distance[neighbour] = new_distance

    return distance

g = Graph()
print('Undirected Graph')
print('Menu')
print('add vertex <key>')
print('add edge <src> <dest> <weight>')
print('shortest <source vertex key>')
print('display')
print('quit')

while True:
    do = input('What would you like to do? ').split()

    operation = do[0]
    if operation == 'add':
        suboperation = do[1]
        if suboperation == 'vertex':
            key = int(do[2])
            if key not in g:
                g.add_vertex(key)
            else:
                print('Vertex already exists.')
        elif suboperation == 'edge':
            src = int(do[2])
            dest = int(do[3])
            weight = int(do[4])
            if src not in g:
                print('Vertex {} does not exist.'.format(src))
            elif dest not in g:
                print('Vertex {} does not exist.'.format(dest))
            else:
                if not g.does_edge_exist(src, dest):
                    g.add_edge(src, dest, weight)
                    g.add_edge(dest, src, weight)
                else:
                    print('Edge already exists.')

    elif operation == 'shortest':
        key = int(do[1])
        source = g.get_vertex(key)
        distance = dijkstra(g, source)
        print('Distances from {}: '.format(key))
        for v in distance:
            print('Distance to {}: {}'.format(v.get_key(), distance[v]))
```

```
        print()

    elif operation == 'display':
        print('Vertices: ', end='')
        for v in g:
            print(v.get_key(), end=' ')
        print()

        print('Edges: ')
        for v in g:
            for dest in v.get_neighbours():
                w = v.get_weight(dest)
                print('(src={}, dest={}, weight={}) '.format(v.get_key(),
                                                    dest.get_key(), w))
        print()

    elif operation == 'quit':
        break
```

OUTPUT:

```
Case 1:
Undirected Graph
Menu
add vertex <key>
add edge <src> <dest> <weight>
shortest <source vertex key>
display
quit
What would you like to do? add vertex 1
What would you like to do? add vertex 2
What would you like to do? add vertex 3
What would you like to do? add vertex 4
What would you like to do? add vertex 5
What would you like to do? add vertex 6
What would you like to do? add vertex 7
What would you like to do? add edge 1 2 10
What would you like to do? add edge 1 3 80
What would you like to do? add edge 3 4 70
What would you like to do? add edge 2 5 20
What would you like to do? add edge 2 3 6
What would you like to do? add edge 5 6 50
What would you like to do? add edge 5 7 10
What would you like to do? add edge 6 7 5
What would you like to do? shortest 1
Distances from 1:
Distance to 6: 45
Distance to 3: 16
Distance to 4: 86
```

Distance to 5: 30
Distance to 2: 10
Distance to 7: 40
Distance to 1: 0

What would you like to do? quit

Case 2:
Undirected Graph
Menu
add vertex <key>
add edge <src> <dest> <weight>
shortest <source vertex key>
display
quit
What would you like to do? add vertex 1
What would you like to do? add vertex 2
What would you like to do? add vertex 3
What would you like to do? add vertex 4
What would you like to do? add edge 1 2 10
What would you like to do? add edge 2 3 20
What would you like to do? add edge 3 4 30
What would you like to do? add edge 1 4 100
What would you like to do? shortest 1
Distances from 1:
Distance to 2: 10
Distance to 4: 60
Distance to 3: 30
Distance to 1: 0

RESULT:
Thus, the Python program to implement Dijkstra's Shortest Path algorithm on a graph has been executed successfully.

EX:NO:14 MINIMUM SPANNING TREE IMPLEMENTATION

DATE:

AIM:

To wrote a Python program to find a minimum spanning tree of an undirected weighted graph using Kruskal's algorithm.

ALGORITHM:

1. Create classes for Graph and Vertex.
2. Create a function mst_krusal that takes a Graph object g as argument.
3. The function will return a Graph object which is a minimum spanning tree of the graph g.
4. An empty graph called mst is created which will hold a MST of the graph g.
5. The algorithm works by first sorting all the edges of g in ascending order by weight.
6. Then the smallest edge is taken from the sorted list.
7. If that edge does not form a cycle when added to mst, it is added.
8. Then the next smallest edge is taken and step 7 is performed again.
9. The above is performed until mst has the same number of vertices as g.
10. To determine whether adding an edge will form a cycle, each vertex in g is assigned a component.
11. When any vertex is added to the MST, its component is updated.
12. If both vertices of an edge belong to the same component, then adding the edge will form a cycle.

PROGRAM:

```
class Graph:
    def __init__(self):
        # dictionary containing keys that map to the corresponding vertex object
        self.vertices = {}

    def add_vertex(self, key):
        """Add a vertex with the given key to the graph."""
        vertex = Vertex(key)
        self.vertices[key] = vertex

    def get_vertex(self, key):
        """Return vertex object with the corresponding key."""
        return self.vertices[key]

    def __contains__(self, key):
        return key in self.vertices

    def add_edge(self, src_key, dest_key, weight=1):
        """Add edge from src_key to dest_key with given weight."""
        self.vertices[src_key].add_neighbour(self.vertices[dest_key], weight)
```

```
    def does_vertex_exist(self, key):
        return key in self.vertices

    def does_edge_exist(self, src_key, dest_key):
        """Return True if there is an edge from src_key to dest_key."""
        return self.vertices[src_key].does_it_point_to(self.vertices[dest_key])

    def display(self):
        print('Vertices: ', end='')
        for v in self:
            print(v.get_key(), end=' ')
        print()

        print('Edges: ')
        for v in self:
            for dest in v.get_neighbours():
                w = v.get_weight(dest)
                print('(src={}, dest={}, weight={}) '.format(v.get_key(),
                                                    dest.get_key(), w))

    def __len__(self):
        return len(self.vertices)

    def __iter__(self):
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, key):
        self.key = key
        self.points_to = {}

    def get_key(self):
        """Return key corresponding to this vertex object."""
        return self.key

    def add_neighbour(self, dest, weight):
        """Make this vertex point to dest with given edge weight."""
        self.points_to[dest] = weight

    def get_neighbours(self):
        """Return all vertices pointed to by this vertex."""
        return self.points_to.keys()

    def get_weight(self, dest):
        """Get weight of edge from this vertex to dest."""
        return self.points_to[dest]

    def does_it_point_to(self, dest):
```

```
    """Return True if this vertex points to dest."""
    return dest in self.points_to
def mst_krusal(g):
 """Return a minimum cost spanning tree of the connected graph g."""
 mst = Graph() # create new Graph object to hold the MST

 if len(g) == 1:
    u = next(iter(g)) # get the single vertex
    mst.add_vertex(u.get_key()) # add a copy of it to mst
    return mst

 # get all the edges in a list
 edges = []
 for v in g:
    for n in v.get_neighbours():
       # avoid adding two edges for each edge of the undirected graph
       if v.get_key() < n.get_key():
          edges.append((v, n))

 # sort edges
 edges.sort(key=lambda edge: edge[0].get_weight(edge[1]))

 # initially, each vertex is in its own component
 component = {}
 for i, v in enumerate(g):
    component[v] = i

 # next edge to try
 edge_index = 0

 # loop until mst has the same number of vertices as g
 while len(mst) < len(g):
    u, v = edges[edge_index]
    edge_index += 1

    # if adding edge (u, v) will not form a cycle
    if component[u] != component[v]:

       # add to mst
       if not mst.does_vertex_exist(u.get_key()):
          mst.add_vertex(u.get_key())
       if not mst.does_vertex_exist(v.get_key()):
          mst.add_vertex(v.get_key())
       mst.add_edge(u.get_key(), v.get_key(), u.get_weight(v))
       mst.add_edge(v.get_key(), u.get_key(), u.get_weight(v))

       # merge components of u and v
       for w in g:
          if component[w] == component[v]:
             component[w] = component[u]
```

```
    return mst


g = Graph()
print('Undirected Graph')
print('Menu')
print('add vertex <key>')
print('add edge <src> <dest> <weight>')
print('mst')
print('display')
print('quit')

while True:
    do = input('What would you like to do? ').split()

    operation = do[0]
    if operation == 'add':
        suboperation = do[1]
        if suboperation == 'vertex':
            key = int(do[2])
            if key not in g:
                g.add_vertex(key)
            else:
                print('Vertex already exists.')
        elif suboperation == 'edge':
            src = int(do[2])
            dest = int(do[3])
            weight = int(do[4])
            if src not in g:
                print('Vertex {} does not exist.'.format(src))
            elif dest not in g:
                print('Vertex {} does not exist.'.format(dest))
            else:
                if not g.does_edge_exist(src, dest):
                    g.add_edge(src, dest, weight)
                    g.add_edge(dest, src, weight)
                else:
                    print('Edge already exists.')

    elif operation == 'mst':
        mst = mst_krusal(g)
        print('Minimum Spanning Tree:')
        mst.display()
        print()

    elif operation == 'display':
        g.display()
        print()
     elif operation == 'quit':
```

```
        break
```

OUTPUT:

```
Case 1:
Undirected Graph
Menu
add vertex <key>
add edge <src> <dest> <weight>
mst
display
quit
What would you like to do? add vertex 1
What would you like to do? add vertex 2
What would you like to do? add vertex 3
What would you like to do? add vertex 4
What would you like to do? add vertex 5
What would you like to do? add vertex 6
What would you like to do? add edge 1 2 10
What would you like to do? add edge 1 5 30
What would you like to do? add edge 1 4 40
What would you like to do? add edge 2 5 20
What would you like to do? add edge 4 5 40
What would you like to do? add edge 5 3 40
What would you like to do? add edge 5 6 70
What would you like to do? add edge 3 6 50
What would you like to do? mst
Minimum Spanning Tree:
Vertices: 1 2 3 4 5 6
Edges:
(src=1, dest=4, weight=40)
(src=1, dest=2, weight=10)
(src=2, dest=5, weight=20)
(src=2, dest=1, weight=10)
(src=3, dest=5, weight=40)
(src=3, dest=6, weight=50)
(src=4, dest=1, weight=40)
(src=5, dest=2, weight=20)
(src=5, dest=3, weight=40)
(src=6, dest=3, weight=50)

What would you like to do? quit

Case 2:
Undirected Graph
Menu
add vertex <key>
add edge <src> <dest> <weight>
mst
display
quit
What would you like to do? add vertex 1
```

What would you like to do? add vertex 2
What would you like to do? add vertex 3
What would you like to do? add edge 1 2 10
What would you like to do? add edge 1 3 20
What would you like to do? add edge 2 3 30
What would you like to do? mst
Minimum Spanning Tree:
Vertices: 1 2 3
Edges:
(src=1, dest=3, weight=20)
(src=1, dest=2, weight=10)
(src=2, dest=1, weight=10)
(src=3, dest=1, weight=20)

What would you like to do? quit

RESULT:
Thus, the Python program to implement Kruskal's algorithm on a minimum spanning tree has been executed successfully.