UIT2402 -- ADVANCED DATA STRUCTURES AND ALGORITHM ANALYSIS LAB

EX NO : 4

1. Implement and analyze the fundamental operations of a Splay Tree.

Operations to Implement:

- Insertion
- Search (Splaying an element to the root)
- Deletion

Do the following operations:

1. Insert a following sequence of numbers into a splay tree,

   1. 50, 30, 70, 20, 40, 60, 80,100

2. Search element 70 and observe how it moves to the root.

3. Delete an element 60 from the constructed splay tree.

   While performing above operations perform the necessary rotations.

<u>ALGORITHM:</u>

- Start with an empty splay tree.
- Insert elements one by one, placing each in the correct position.

- After each insertion, splay the newly inserted node to the root using rotations.
- To search for an element, splay it to the root if found; otherwise, bring the closest node to the root.
- Splaying is performed using zig, zig-zig, or zig-zag rotations to move the node up.
- For deletion, first splay the target node to the root.
- If the node has two children, splay the largest node in the left subtree to become the new root.
- Attach the right subtree of the deleted node to the new root.
- Maintain the splay tree's self-balancing property through splaying after every operation.
- The tree adapts dynamically to access patterns, improving performance over time.

CODING:

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

```
        self.root=None
    def right_rotate(self,x):
        y=x.left
        x.left=y.right
        y.right=x
        return y
    def left_rotate(self,x):
        y=x.right
        x.right=y.left
        y.left=x
        return y
    def splay(self,root,key):
        if root is None or root.key==key:
            return root
        if key<root.key:
            if root.left is None:
                return root
            if key<root.left.key:
                root.left.left=self.splay(root.left.left,key)
                root=self.right_rotate(root)
            elif key>root.left.key:
                root.left.right=self.splay(root.left.right,key)
```

```
            if root.left.right:
                root.left=self.left_rotate(root.left)
        return root if root.left is None else self.right_rotate(root)
    else:
        if root.right is None:
            return root
        if key>root.right.key:
            root.right.right=self.splay(root.right.right,key)
            root=self.left_rotate(root)
        elif key<root.right.key:
            root.right.left=self.splay(root.right.left,key)
            if root.right.left:
                root.right=self.right_rotate(root.right)
        return root if root.right is None else self.left_rotate(root)
def insert(self,key):
    if self.root is None:
        self.root=Node(key)
        return
    self.root=self.splay(self.root,key)
    if key==self.root.key:
```

```
            return
        new_node=Node(key)
        if key<self.root.key:
            new_node.right=self.root
            new_node.left=self.root.left
            self.root.left=None
        else:
            new_node.left=self.root
            new_node.right=self.root.right
            self.root.right=None
        self.root=new_node
    def search(self,key):
        self.root=self.splay(self.root, key)
        return self.root.key == key if self.root else False
    def delete(self,key):
        if self.root is None:
            return
        self.root=self.splay(self.root,key)
        if self.root.key!=key:
            return
        if self.root.left is None:
            self.root=self.root.right
```

```
        else:
            temp=self.root.right
            self.root=self.splay(self.root.left,key)
            self.root.right=temp
    def print_tree(self,node,indent="",last=True):
        if node:
            print(indent, "`- " if last else "|- ",node.key,sep="")
            indent+="   " if last else "|  "
            self.print_tree(node.left,indent,False)
            self.print_tree(node.right,indent,True)
splay_tree=SplayTree()
for num in [50,30,70,20,40,60,80,100]:
    splay_tree.insert(num)
print("\nTree after insertions:")
splay_tree.print_tree(splay_tree.root)
print("\nSearching for 70:")
splay_tree.search(70)
splay_tree.print_tree(splay_tree.root)
print("\nDeleting 60:")
splay_tree.delete(60)
splay_tree.print_tree(splay_tree.root)
```

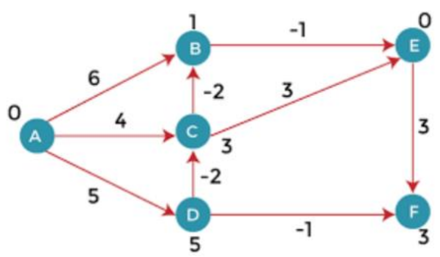
<u>OUTPUT:</u>

```
Tree after insertions:
`- 100
   |- 80
   |   |- 70
   |   |   |- 60
   |   |   |   |- 50
   |   |   |   |   |- 40
   |   |   |   |   |   |- 30
   |   |   |   |   |   |   |- 20

Searching for 70:
`- 70
   |- 60
   |   |- 50
   |   |   |- 40
   |   |   |   |- 30
   |   |   |   |   |- 20
   `- 80
       `- 100

Deleting 60:
`- 50
   |- 40
   |   |- 30
   |   |   |- 20
   `- 70
       `- 80
           `- 100
```

2. Implement Bellman Ford algorithm to find the shortest path from source vertex ‘A’ to all other vertices.



ALGORITHM:

- Initialize the graph with vertices and edges.
- Set the source vertex distance to 0 and all other vertices to infinity.
- Iterate (V-1) times to relax all edges.
- For each edge (u, v, w), update dist[v] if dist[u] + w < dist[v].
- Repeat this process for all edges to ensure shortest paths are updated.
- After (V-1) iterations, check for negative weight cycles by performing one more relaxation.
- If any distance still reduces, a negative weight cycle exists.
- If no updates occur in an iteration, the algorithm can terminate early.
- Store or display the shortest distances from the source vertex to all other vertices.
- The algorithm runs in O(V × E) time complexity and is useful for graphs with negative weights.

CODING:

```
class Graph:
    def __init__(self):
        self.adj_list={}
    def add_edge(self,node,neighbor,cost):
        if node not in self.adj_list:
```

```
            self.adj_list[node]=[]
        self.adj_list[node].append((neighbor,cost))
        if neighbor not in self.adj_list:
            self.adj_list[neighbor]=[]
    def get_neighbors(self,node):
        return self.adj_list.get(node,[])
    def bellman_ford(self,src):
        dist={node:float('inf') for node in self.adj_list}
        dist[src]=0
        for _ in range(len(self.adj_list)-1):
            for node in self.adj_list:
                for neighbor, cost in self.adj_list[node]:
                    if dist[node]!=float('inf') and
dist[node]+cost<dist[neighbor]:
                        dist[neighbor]=dist[node]+cost
        for node in self.adj_list:
            for neighbor,cost in self.adj_list[node]:
                if dist[node]!=float('inf') and
dist[node]+cost<dist[neighbor]:
                    print("Graph contains a negative weight cycle")
                    return
        print("Vertex\t Distance from Source")
```

```python
        for vertex in sorted(dist.keys()):
            print(f"{vertex}\t\t{dist[vertex]}")
g=Graph()
g.add_edge('A','B',6)
g.add_edge('A','C',4)
g.add_edge('A','D',5)
g.add_edge('B','E',-1)
g.add_edge('C','B',-2)
g.add_edge('C','E',3)
g.add_edge('D','C',-2)
g.add_edge('D','F',-1)
g.add_edge('E','F',3)
g.bellman_ford('A')
```

<u>OUTPUT:</u>

```
Vertex   Distance from Source
A                0
B                1
C                3
D                5
E                0
F                3
```

3. Write a python program to find the Binomial coefficient C(n,k) for the given n and k value using dynamic programming approach, where n<=k. and both are integers.

<u>ALGORITHM:</u>

- The binomial coefficient C(n, k) can be computed efficiently using dynamic programming to avoid redundant calculations.
- First, create a 2D table (dp) of size $(n+1) \times (k+1)$, where dp[i][j] stores the value of C(i, j).
- Initialize the base cases: C(i, 0) = 1 for all i, because selecting 0 elements from any set has exactly one way, and C(i, i) = 1 since choosing all elements from a set has only one combination.
- Iterate over i from 1 to n, and for each i, iterate over j from 1 to min(i, k). Use the recurrence relation:
  $C(i,j)=C(i-1,j-1)+C(i-1,j)C(i, j) = C(i-1, j-1) + C(i-1, j)C(i,j)=C(i-1,j-1)+C(i-1,j)$
  to fill up the table iteratively.
- This ensures that smaller subproblems are solved first and reused, reducing time complexity to $O(n \times k)$.
- Once the table is filled, the required result C(n, k) is stored at dp[n][k].
- This approach significantly reduces redundant calculations compared to the recursive approach.
- The space complexity is $O(n \times k)$, but it can be optimized to O(k) using a 1D array.
- Finally, return dp[n][k] as the output.

CODING:

```
def binomial_coefficient(n,k):
    dp=[[0 for _ in range(k+1)] for _ in range(n+1)]
    for i in range(n+1):
        for j in range(min(i,k)+1):
            if j==0 or j==i:
                dp[i][j]=1
            else:
                dp[i][j]=dp[i-1][j-1]+dp[i-1][j]
    return dp[n][k]
n=int(input("Enter n:"))
k=int(input("Enter k:"))
if n>=k and n>=0 and k>=0:
    print(f"Binomial Coefficient C({n},{k})={binomial_coefficient(n,k)}")
else:
    print("Invalid input! Ensure that n >= k and both are non-negative integers.")
```

OUTPUT:

```
Enter n:10
Enter k:4
Binomial Coefficient C(10,4)=210
```