```python
from math import inf

class Vertex:
    def __init__(self, name = None, parent = None, distance = None):
        self.name = name
        self.parent = parent
        self.distance = distance

    def __str__(self):
        return str(self.name)

    def __repr__(self):
        return str(self.name)

class Edge:
    def __init__(self, src = None, dst = None, weight = None):
        self.src = src
        self.dst = dst
        self.weight = weight

class Graph:
    def __init__(self, vertices = None, edges = None):
        self.vertices = vertices
        self.edges = edges

    def getNode(self, name):
        for v in self.vertices:
            if v.name == name:
                return v

    def GetInDegrees(self):
        list = []

        for v in self.vertices:
            counter = 0

            for e in self.edges:
                if e.dst == v:
                    counter += 1

            list.append(counter)

        return list

    def GetOutDegrees(self):
        list = []

        for v in self.vertices:
            counter = 0
            for e in self.edges:
                if e.src == v:
                    counter += 1

            list.append(counter)

        return list

    def initialiseSingleSource(self, s):
        for v in self.vertices:
            v.distance = inf
            v.parent = None

        s.distance = 0

    def relax(self, u, v, w):
        if v.distance > u.distance + w:
            v.distance = u.distance + w
            v.parent = u

    def extractMin(self, Q):
        min = Q[0]

        for v in Q:
            if v.distance < min.distance:
                min = v

        Q.remove(min)

        return min

    def getAdj(self, x):
        list = []

        for e in self.edges:
            if e.src == x:
                list.append(e.dst)

        return list

    def getWeight(self, u, v):
        for e in self.edges:
            if e.src == u and e.dst == v:
                return e.weight

    def Dijkstra(self, a):
        self.initialiseSingleSource(a)
        S = []
        Q = self.vertices[:]
```

```
        while len(Q) > 0:
            u = self.extractMin(Q)
            S.append(u)
            for v in self.getAdj(u):
                self.relax(u, v, self.getWeight(u, v))

    def printShortestPath(self, a, b, list):
        if a == b:
            list.append(a)
        elif b.parent is not None:
            self.printShortestPath(a, b.parent, list)
            list.append(b)
        else:
            print("There is no such a path")
            list.clear()
            return

    def Bellman_Ford(self, a):
        self.initialiseSingleSource(a)
        for v in self.vertices:
            for e in self.edges:
                self.relax(e.src, e.dst, e.weight)

        for e in self.edges:
            if e.dst.distance > e.src.distance + e.weight:
                return False

        return True

    def ShortestPath(self, a, b, flag):
        list = []

        if(flag == True):
            self.Dijkstra(a)
        else:
            if(self.Bellman_Ford(a)):
                print("Bellman Ford successfully finished")
            else:
                print("Bellman Ford unsuccessfully finished")

        self.printShortestPath(a, b, list)
        return list, b.distance

    def UpdateEdge(self, a, b, w):
        for e in self.edges:
            if e.src == a and e.dst == b:
                e.weight = w
                return

        e = Edge(src = a, dst = b, weight = w)
        self.edges.append(e)

def MakeGraph():
    a = Vertex(name = "a")
    b = Vertex(name = "b")
    c = Vertex(name = "c")
    d = Vertex(name = "d")
    e = Vertex(name = "e")
    f = Vertex(name = "f")
    g = Vertex(name = "g")
    h = Vertex(name = "h")

    vertices = [a, b, c, d, e, f, g, h]

    e1  = Edge(src = a, dst = b, weight = 5 )
    e2  = Edge(src = a, dst = d, weight = 8 )
    e3  = Edge(src = b, dst = d, weight = 9 )
    e4  = Edge(src = d, dst = c, weight = 1 )
    e5  = Edge(src = d, dst = e, weight = 10)
    e6  = Edge(src = d, dst = f, weight = 13)
    e7  = Edge(src = e, dst = g, weight = 7 )
    e8  = Edge(src = f, dst = g, weight = 20)
    e9  = Edge(src = h, dst = c, weight = 16)
    e10 = Edge(src = h, dst = e, weight = 2 )
    e11 = Edge(src = h, dst = g, weight = 24)

    edges = [e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11]

    graph = Graph(vertices = vertices, edges = edges)

    return graph

    #return graph

if __name__ == "__main__":
    #1
    graph = MakeGraph()

    a = graph.getNode("a")
    d = graph.getNode("d")
    g = graph.getNode("g")
    h = graph.getNode("h")

    print("Existing path Demo:")
    list, distance = graph.ShortestPath(a, g, True)
    print("List of nodes on shortest path = {0}".format(list))
    print("Distance = {0}".format(distance))

    print("")
    print("Non existing path Demo:")
    list, distance = graph.ShortestPath(a, h, True)
    print("List of nodes on shortest path = {0}".format(list))
    print("Distance = {0}".format(distance))

    print("")

    print("Update Edge Demo:")
    graph.UpdateEdge(d, g, -4)

    list, distance = graph.ShortestPath(a, g, False)
    print("List of nodes on shortest path = {0}".format(list))
    print("Distance = {0}".format(distance))

    print("")

    print("Update Edge Demo(2):")
    graph.UpdateEdge(a, d, 16)

    list, distance = graph.ShortestPath(a, g, False)
    print("List of nodes on shortest path = {0}".format(list))
    print("Distance = {0}".format(distance))




    ####################

    bellman raw


    from math import inf

class Vertex:   # cvor
    def __init__(self, name):
        self.name = name
        self.p = None   # parent
        self.d = inf    # distance

    def __str__(self):
        return self.name

    def __repr__(self):     # preklapanje stringa za liste
        return self.name

class Edge:     # ivica
    def __init__(self, src, dst, w):   # putanja od sorce do distenation i w tezina
        self.src = src
        self.dst = dst
        self.w = w

class Graph:
    def __init__(self, V, E):
        self.V = V
        self.E = E

    def __str__(self):
        ret_str = ""
        for e in self.E:
            ret_str += str(e.src) + " -> " + str(e.dst) + ", w = " + str(e.w) + "\n"
        return ret_str

    def init_single_source(self, s):    # pocetni podesavanje grafa
        for v in self.V:
            v.d = inf
            v.p = None
        s.d = 0

    def get_weight(self, u, v):     # vraca samo za direktrno povezane cvorove
        for e in self.E:
            if e.src == u and e.dst == v:
                return e.w
        return inf

    def relax(self, u, v, w):       # provera da li postoji kraca putanja od cvora
do cvora
        if v.d > u.d + w:
            v.d = u.d + w
            v.p = u

    def bellman_ford(self, s):
        self.init_single_source(s)
        for v in self.V:
            for e in self.E:
                self.relax(e.src, e.dst, e.w)
        for e in self.E:            # false ako postoji  negativni ciklus
            if e.dst.d > e.src.d + e.w:
                return False
        return True

    def print_path(self, u, v, ret_list):
        if v is u:
            ret_list.append(u)
        elif v.p is None:
            print("No valid path")
            return None
        else:
            self.print_path(u, v.p, ret_list)
            ret_list.append(v)
        return ret_list

    def shortest_path(self, u, v):
        ret = self.bellman_ford(u)
        if not ret:
            print("No valid path")
        path = self.print_path(u, v, [])
        return path, v.d


    def get_in_degrees(self):   # ulazne linije u cvor
        ret_list = []
        ret_dict = {}
        for v in self.V:
            counter = 0
            for e in self.E:
                if e.dst == v:
                    counter += 1
            ret_list.append(counter)
            ret_dict[v.name] = counter

        return ret_list, ret_dict

    def get_out_degrees(self):   # izlazne linije iz cvora
        ret_list = []
        ret_dict = {}
        for v in self.V:
            counter = 0
            for e in self.E:
                if e.src == v:
                    counter += 1
            ret_list.append(counter)
            ret_dict[v.name] = counter

        return ret_list, ret_dict

    def update_egde(self, u, v, W): # pravimo novu ivicu ili menjamo postojecu
        for e in self.E:
            if e.src == u and e.dst == v:
                e.w = W
                return
        e = Edge(u, v, W)
        E.append(e)


if __name__ == "__main__":

    s = Vertex("s")
    t = Vertex("t")
    x = Vertex("x")
    y = Vertex("y")
    z = Vertex("z")

    V = [s, t, x, y, z]

    e1 = Edge(s, t, 6)
    e2 = Edge(s, y, 7)
    e3 = Edge(t, y, 8)
    e4 = Edge(t, z, -4)
    e5 = Edge(t, x, 5)
    e6 = Edge(x, t, -2)
    e7 = Edge(y, x, -3)
    e8 = Edge(y, z, 9)
    e9 = Edge(z, x, 7)
    e10 = Edge(z, s, 2)

    E = [e1, e2, e3, e4, e5, e6, e7, e8, e9, e10]

    G = Graph(V, E)

    G.bellman_ford(s)
    # print(G)
    # print(G.get_in_degrees())
    # print(G.get_out_degrees())
    print(G.shortest_path(s, z))
    G.update_egde(s, x, 3)
    print(G)



    ##############  djik


    from enum import Enum
from math import inf
from queue import Queue


class Vertex:

    def __init__(self, p=None, d=None, name=None):
        self.p = p
        self.d = d
        self.name = name

    def __str__(self):
        return str(self.name)

    def __repr__(self):
        return str(self.name)
```

```
class Edge:
    def __init__(self, src=None, dst=None, w=None):
        self.src = src
        self.dst = dst
        self.w = w


class Graph:
    def __init__(self, V=None, E=None):
        self.V = V
        self.E = E

    def print_path(self, s, v):
        if v is s:
            print(s)
        elif v.p is None:
            print("No path from", s, "to", v, "exists.")
        else:
            self.print_path(s, v.p)
            print(v, " do ove tacke presli smo:", v.d)

    def get_adj(self, v):       # pronalazenje suseda
        ret = []

        for e in self.E:
            if v == e.src:
                ret.append(e.dst)

        return ret

    def extract_min(self, Q):   # dobijamo minimum distancu iz cvorova
        local_min = Q[0]
        for v in Q:
            if v.d < local_min.d:
                local_min = v
        Q.remove(local_min)
        return local_min


    def initialize_single_source(self, s):  # pocetno podesavanje algoritma dijkstra
        for v in self.V:
            v.d = inf
            v.p = None
        s.d = 0

    def get_weight(self, s, d):     # tezina/duzina veze izmedju cvorova s i d
        for e in self.E:
            if e.src == s and e.dst == d:
                return e.w
        return -1

    def relax(self, u, v, w):   # smanjuje/relaksira tezine ivica izmedju cvorova
        if v.d > u.d + w:
            v.d = u.d + w
            v.p = u

    def dijkstra(self, s):
        self.initialize_single_source(s)
        S = []
        Q = self.V[:] # test [] + self.V
        while len(Q) > 0:
            u = self.extract_min(Q)
            S.append(u)
            for v in self.get_adj(u):
                self.relax(u, v, self.get_weight(u, v))

if __name__ == "__main__":
    v1 = Vertex(name="Novi Sad")
    v2 = Vertex(name="Naplatna rampa NS")
    v3 = Vertex(name="Petrovaradin")
    v4 = Vertex(name="Sremski Karlovci")
    v5 = Vertex(name="Ruma")
    v6 = Vertex(name="Naplatna rampa Stara Pazova")
    v7 = Vertex(name="Stara Pazova")
    v8 = Vertex(name="Pecinci")
    v9 = Vertex(name="Naplatna rampa Simanovci")
    v10 = Vertex(name="Iskljucenje Batajnica")
    v11 = Vertex(name="Batajnica")
    v12 = Vertex(name="Ukljucenje na AP E70")
    v13 = Vertex(name="Beograd")

    V = [v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13]

    e1 = Edge(v1, v2, 9)
    e2 = Edge(v1, v3, 7)
    e3 = Edge(v1, v4, 14)
    e4 = Edge(v1, v5, 33)
    e5 = Edge(v2, v6, 45)
    e6 = Edge(v3, v4, 6)
    e7 = Edge(v5, v8, 16)
    e8 = Edge(v2, v6, 45)
    e9 = Edge(v4, v7, 32)
    e10 = Edge(v8, v9, 15)
    e11 = Edge(v7, v9, 14)
    e12 = Edge(v6, v10, 28)
    e13 = Edge(v7, v11, 13)
    e14 = Edge(v9, v12, 11)
    e15 = Edge(v10, v11, 4)
    e16 = Edge(v10, v12, 16)
    e17 = Edge(v11, v13, 22)
    e18 = Edge(v12, v13, 20)


    E = [e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13, e14, e15, e16, e17,
e18]

    G = Graph(V, E)
    G.dijkstra(v1)
    G.print_path(v1, v13)
    print('Ukupna duzina je: ', v13.d)




    ####################  BST SA PROSLOG

    ##  bst.py

    class Node:
    """
    Tree node: left child, right child and data
    """
    def __init__(self, parent=None, left=None, right=None, data=None):
        """
        Node constructor
        @param A node data object
        """
        self.parent = parent
        self.left = left
        self.right = right
        self.data = data


class Tree:
    """
    Tree class: contains tree nodes
    """

    def __init__(self, root=None):
        """
        Tree constructor
        @param root element
        """
        self.root = root

    def print_tree(self):
        def display(root):
            """Returns list of strings, width, height, and horizontal coordinate of
the root."""
```

```
            # No child.
            if root.right is None and root.left is None:
                line = '%s' % root.data
                width = len(line)
                height = 1
                middle = width // 2
                return [line], width, height, middle

            # Only left child.
            if root.right is None:
                lines, n, p, x = display(root.left)
                s = '%s' % root.data
                u = len(s)
                first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
                second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
                shifted_lines = [line + u * ' ' for line in lines]
                return [first_line, second_line] + shifted_lines, n + u, p + 2, n +
u // 2

            # Only right child.
            if root.left is None:
                lines, n, p, x = display(root.right)
                s = '%s' % root.data
                u = len(s)
                first_line = s + x * '_' + (n - x) * ' '
                second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
                shifted_lines = [u * ' ' + line for line in lines]
                return [first_line, second_line] + shifted_lines, n + u, p + 2, u //
2

            # Two children.
            left, n, p, x = display(root.left)
            right, m, q, y = display(root.right)
            s = '%s' % root.data
            u = len(s)
            first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s + y * '_' + (m - y) *
' '
            second_line = x * ' ' + '/' + (n - x - 1 + u + y) * ' ' + '\\' + (m - y
- 1) * ' '
            if p < q:
                left += [n * ' '] * (q - p)
            elif q < p:
                right += [m * ' '] * (p - q)
            zipped_lines = zip(left, right)
            lines = [first_line, second_line] + [a + u * ' ' + b for a, b in
zipped_lines]
            return lines, n + m + u, max(p, q) + 2, n + u // 2

        lines, *_ = display(self.root)
        for line in lines:
            print(line)

    # z - čvor koji se dodaje u stablo
    def insert(self, z):
        y = None
        x = self.root
        while x is not None:
            y = x
            if z.data < x.data:
                x = x.left
            else:
                x = x.right
        z.parent = y
        if y is None:
            self.root = z
        elif z.data < y.data:
            y.left = z
        else:
            y.right = z

        return z

    def walk(self, x): #x je vrednost node-a
        node = self.search(self.root, x)
        x_str = ""
        y_str = ""
        x = node.left
        if x == None:
            x_str = "None"
        else:
            x_str = x.data

        y = node.right
        if y == None:
            y_str = "None"
        else:
            y_str = y.data
        z = node.parent
        print("value: "+ str(node.data)+", parent: "+str(z.data)+",
left:"+str(x_str)+", right: "+str(y_str))

    # key - vrednost čvora koji se traži
    def search(self, x, key):
        if x is None:
            return None
        if x.data == key:
            return x
        left = self.search(x.left, key)
        right = self.search(x.right, key)
        if left is not None:
            return left
        if right is not None:
            return right
        return None

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

    # data - vrednost čvora čiji se naslednik traži
    def find_successor(self, data):
        t = self.search(self.root, data)
        if t.right is not None:
            return self.find_min(t.right)
        else:
            y = t.parent
            while y is not None and t == y.right:
                t = y
                y = y.parent
            return y


#T = Tree()
#T.print_tree()





##### main.py



import bst as b

tree = b.Tree()

insert_list =
[200,190,210,180,195,205,220,154,192,198,207,210,230,194,197,199,215,240,193,239]

for br in insert_list:
    tree.insert(b.Node(data=br))

tree.print_tree()  #a

testB = 201 #b   , postaviti zeljenu vrednost

bb = tree.search(tree.root, testB)
if bb==None:
    tree.insert(b.Node(data=testB)) #c
    print("Vrednost " + str(testB) + " nije postojala, sada je dodata")
else:
    print("Vrednost "+ str(testB)+ " postoji")

tree.print_tree() #d

x = tree.find_min(tree.root) #e

print("Minimalna vrednost : " + str(x.data))


y = tree.find_successor(x.data) #f

print("Vrednost naslednika najmanjeg cvora je : "+ str(y.data))


z = tree.search(tree.root, 215) #e
print("Vrednost crvenog cvora jeste : "+str(z.data))

k = tree.find_successor(z.data) #g
print("Vrednost naslednika crvenog cvora je : "+ str(k.data))



######## HEAP

import math

#TODO Heap
def parent(index):
    return (index - 1) // 2

def left(index):
    return 2*index + 1

def right(index):
    return 2*index + 2

def max_heapify(array, index, heap_size):
    l = left(index)
    r = right(index)

    largest = index

    if l < heap_size and array[l] > array[largest]:
        largest = l

    if r < heap_size and array[r] > array[largest]:
        largest = r
    if largest != index:
        array[index], array[largest] = array[largest], array[index]
        max_heapify(array, largest, heap_size)

def build_max_heap(array, heap_size):
    for i in range(heap_size // 2 - 1, -1, -1):
        max_heapify(array, i, heap_size)

def heap_sort(array, heap_size):
    build_max_heap(array, len(array))
    for i in range(0, len(array)):
        array[0], array[heap_size - 1] = array[heap_size - 1], array[0]
        heap_size -= 1
        max_heapify(array, 0, heap_size)

#TODO Priority queue
def heap_maximum(array):
    return array[0]

def heap_extract_max(array, heap_size):
    if(heap_size < 1):
        print("Error")
        return None

    max = array[0]
    array[0] = array[heap_size - 1]
    heap_size -= 1

    max_heapify(array, 0, heap_size)

    return max, heap_size

if __name__ == "__main__":
    lista = [12, 13, 64, 8, 39, 24, 38, 56, 2]

    print("pre  ", lista)

    build_max_heap(lista, len(lista))

    print("posle ", lista)

    lista.append(100)

    print("100 | ", lista)

    build_max_heap(lista, len(lista))

    print("posle ", lista)
```

```
## JUNSKKK

# Mihajlo Karadzic RA154/2020
from math import inf

class Node:
    def __init__(self, parent=None, left=None, right=None, data=None):
        self.parent = parent
        self.left = left
        self.right = right
        self.data = data

class Tree:
    def __init__(self, root=None):
        self.root = root

    def __str__(self):
        x = self.root

        l = []

        self.in_order_tree_walk(x,l)

        s1 = ""

        for x in l:

            s1= s1 + "Node : "+ str(x.data)
            s1 = s1 + " ------> Left Node: "
            if x.left is None:
                s1 = s1 + "None"
            else:
                s1 = s1 + str(x.left.data)

            s1 = s1 + " , Right Node: "
            if x.right is None:
                s1 = s1 + "None"
            else:
                s1 = s1 + str(x.right.data)

            s1 = s1 + "\n"

        return s1


    def in_order_tree_walk(self, x,l):
        if x is not None:
            self.in_order_tree_walk(x.left,l)
            l.append(x)
            self.in_order_tree_walk(x.right,l)

        return


    def insert(self, z):
        y = None
        x = self.root
        while x is not None:
            y = x
            if z.data < x.data:
                x = x.left
            else:
                x = x.right
        z.parent = y
        if y is None:
            self.root = z
        elif z.data < y.data:
            y.left = z
        else:
            y.right = z

        return z


class Vertex:
    def __init__(self, name = None, parent = None, distance = None):
       self.name = name
       self.parent = parent
       self.distance = distance

    def __str__(self):
        return str(self.name)

    def __repr__(self):  # preklapanje stringa za liste
        return self.name

class Edge:
    def __init__(self, src= None, dst=None, weight = None):
        self.src= src
        self.dst = dst
        self.weight = weight
class Graph:
    def __init__(self,vertices = None,edges = None):
        self.vertices = vertices
        self.edges = edges
```

```
def getNode(self,name):
    for v in self.vertices:
        if v.name == name:
            return v

def initiate_single_source(self,s):
    for v in self.vertices:
        v.distance = inf
        v.parent= None
    s.distance= 0
def extract_min(self,Q):
    min = Q[0]
    for v in Q:
        if v.distance < min.distance:
            min = v
    Q.remove(min)
    return min

def getAdj(self,x):
    lista = list()

    for e in self.edges:
        if e.src==x:
            lista.append(e.dst)
    return lista

def relax(self,u,v,w):
    if v.distance > u.distance +w:
        v.distance = u.distance + w
        v.parent = u

def getWeight(self,s,d):
    for e in self.edges:
        if e.src == s and e.dst ==d:
            return e.weight
    return -1
def Dijkstra(self,a):
    self.initiate_single_source(a)
    S = []

    Q = self.vertices[:]

    while len(Q) > 0:
        u = self.extract_min(Q)
        S.append(u)
        for v in self.getAdj(u):
            self.relax(u,v,self.getWeight(u,v))

    print(S)
    def printShortestPath(self,a,b,lista):
        if a==b:
            lista.append(a)
        elif b.parent is not None:
            self.printShortestPath(a,b.parent,lista)
            lista.append(b)
        else:
            print("There is no path")
            lista.clear()
            return
    def shortestPath(self,a,b):
        lista = list()
        self.Dijkstra(a)

        self.printShortestPath(a ,b ,lista)

        tup = (b.distance, lista)
        return tup

def MakeGraph():
    a = Vertex(name = "60")
    b = Vertex(name = "62")
    c = Vertex(name = "58")
    d = Vertex(name = "42")
    e = Vertex(name = "22")
    f = Vertex(name = "47")
    g = Vertex(name = "45")
    vertices = [a,b,c,d,e,f,g]


    e1 = Edge(src = a, dst = b,weight = 14)  #a2b
    e2 = Edge(src = a, dst = c, weight = 2) #a2c
    e3 = Edge(src=e, dst=b, weight=1)  #e2b
    e4 = Edge(src=d, dst=b, weight=10) #d2b
    e5 = Edge(src=e, dst=d, weight=2) #e2d
    e6 = Edge(src=f, dst=e, weight=5)  #f2e
    e7 = Edge(src=d, dst=f, weight=3)  #d2f
    e8 = Edge(src=g, dst=f, weight=10) #g2f
    e9 = Edge(src=d, dst=a, weight=8)  #d2a
    e10 = Edge(src=c, dst=d, weight=4)  #c2d
    e11 = Edge(src=c, dst=g, weight=9) #c2g
    e12 = Edge(src=d, dst=g, weight=2) #d2g

    edges = [e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12]


    graph = Graph(vertices, edges)

    return graph
```

```
def MakeTree(shortest_path):
    tree = Tree()

    for x in shortest_path:
        n = Node(data=int(x.name))
        tree.insert(n)

    return tree


if __name__ == "__main__":
    print("Podzadatak a)")

    graph = MakeGraph()

    a = graph.getNode("60")

    b = graph.getNode("62")

    c = graph.getNode("58")

    d = graph.getNode("42")

    e = graph.getNode("22")

    f = graph.getNode("47")

    g = graph.getNode("45")

    print("Podzadatak b)")
    graph.Dijkstra(c)


    # c

    print("Podzadatak c)")
    print("Unesi pocetni cvor:")

    start_c = input()
    start_c = graph.getNode(start_c)

    print("Unesi krajnji cvor:")

    end_c = input()
    end_c = graph.getNode(end_c)


    distance, list = graph.shortestPath(start_c, end_c)
    print("List of nodes on shortest path = "+format(list))
    print("Distance = "+format(distance))
    print("Podzadatak d)")

    tree = MakeTree(list)

    print("Podzadatak e)")

    print(tree)
```