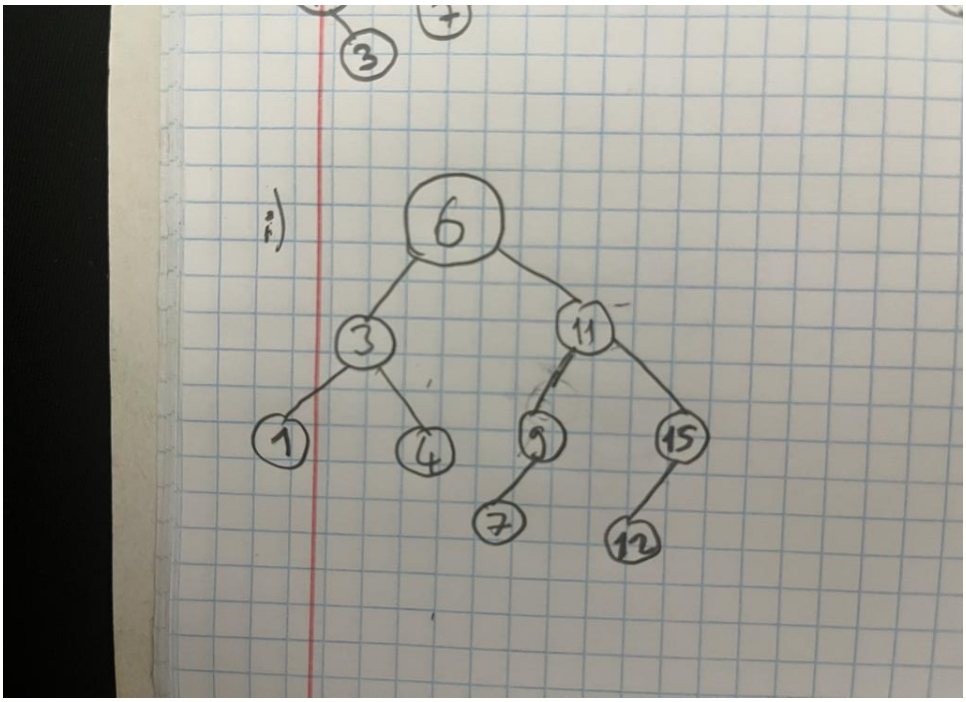
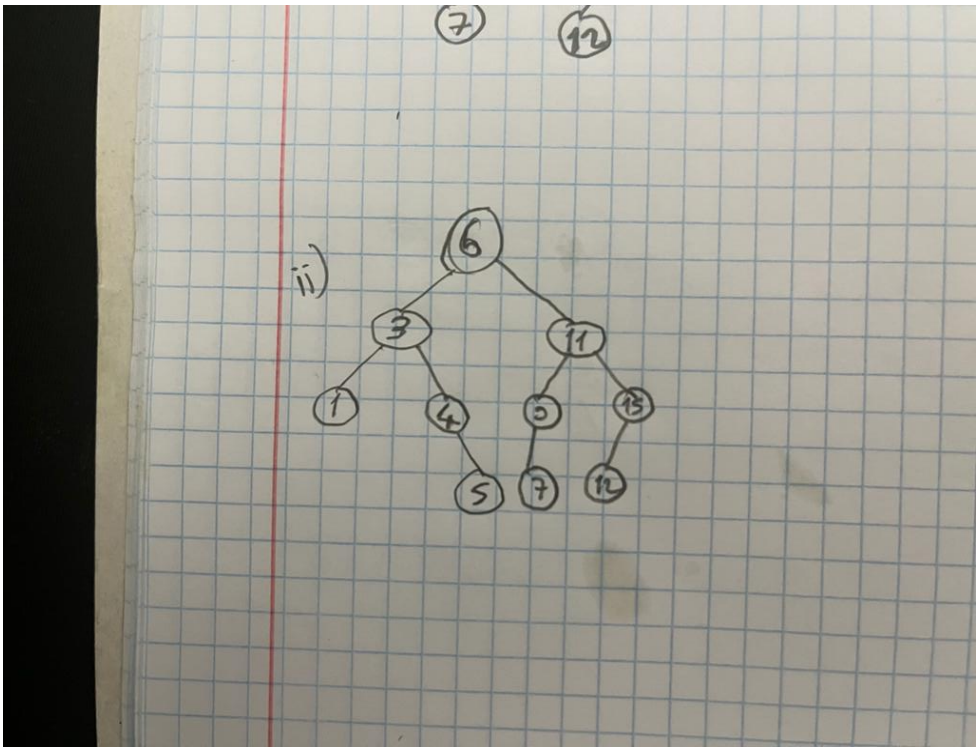
İçindekiler

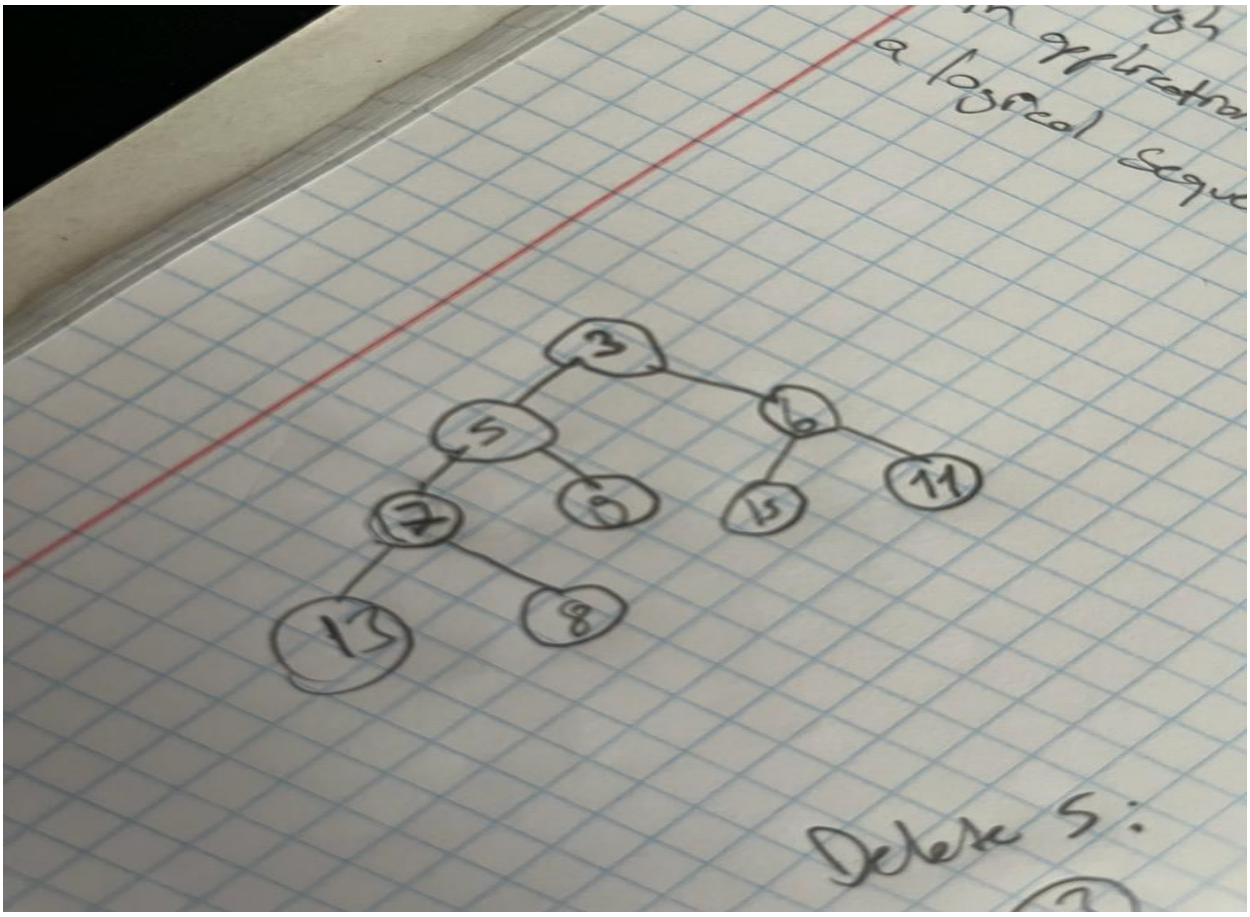
GRAPHS, GRAPH ALGORITHMS, TREES and OTHER SUBJECTS

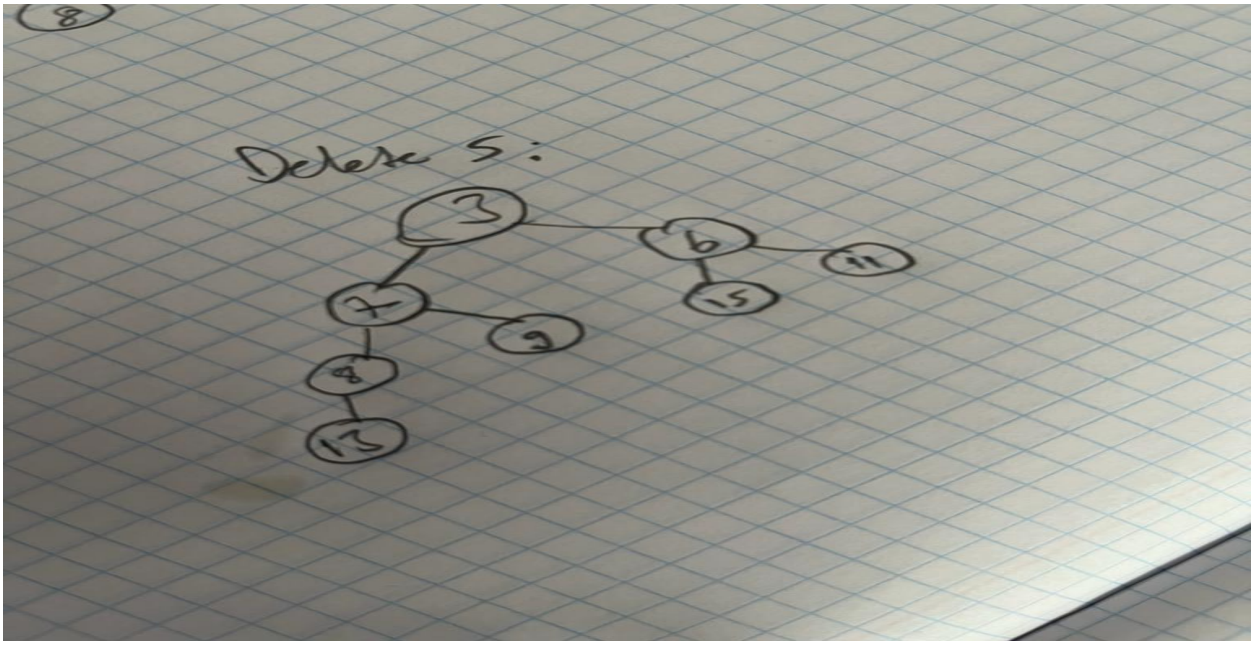
1.a Drawing of new AVL Trees after inserting requested values





1.b Drawing of new Heaps after inserting and removing requested values





2.a B-Tree insertion method (or AVL-Tree insertion method/Red-Black tree/Huffman encoding tree) code

```
                        public void insert(final int key) {
  Node r = root;
  if (r.n == 2 * T - 1) {
    Node s = new Node();
    root = s;
    s.leaf = false;
    s.n = 0;
    s.child[0] = r;
    split(s, 0, r);
    _insert(s, key);
  } else {
    _insert(r, key);
  }
}

// insert node
final private void _insert(Node x, int k) {

  if (x.leaf) {
    int i = 0;
    for (i = x.n - 1; i >= 0 && k < x.key[i]; i--) {
      x.key[i + 1] = x.key[i];
    }
    x.key[i + 1] = k;
    x.n = x.n + 1;
  } else {
    int i = 0;
```

2.b Explanation of B-Tree insertion method steps

`insert` metodu, verilen anahtarı eklemek için kullanılır. Eğer kök düğümdeki anahtar sayısı limiti aşılıyorsa, kök bölünerek yeni bir kök oluşturulur ve ağacın yüksekliği arttırılır. Daha sonra **`_insert`** metodu çağrılır.

- **`_insert`** metodu, anahtarın ekleneceği düğümü bulur ve eğer bu düğüm aşılıyorsa bölünmeyi gerçekleştirir.
- Eğer düğüm bir yaprak düğümü ise, sıralı bir şekilde anahtarları yerleştirir.
- Eğer düğüm bir iç düğüm ise, önce uygun çocuk düğümüne geçer. Eğer çocuk düğüm aşılıyorsa, **`split`** metodunu çağırarak bölünmeyi sağlar ve uygun çocuk düğümüne geçer.
- Son olarak, **`_insert`** metodu rekürsif olarak devam eder, uygun düğümde anahtarı ekler.

.2.c B-Tree insertion test

```
10 8 9 11 15 17 20 BUILD SUCCESSFUL (total time: 0 seconds)
```

3.a Dijkstra's SP source code and tests

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp5
{


    public class DijkstraShortestPath
    {
        private const int Infinity = int.MaxValue;
        public void Main()
        {
            int[,] graph = {
            {0, 4, 0, 0, 0, 0, 0, 8, 0},
            {4, 0, 8, 0, 0, 0, 0, 11, 0},
            {0, 8, 0, 7, 0, 4, 0, 0, 2},
            {0, 0, 7, 0, 9, 14, 0, 0, 0},
            {0, 0, 0, 9, 0, 10, 0, 0, 0},
            {0, 0, 4, 14, 10, 0, 2, 0, 0},
            {0, 0, 0, 0, 0, 2, 0, 1, 6},
            {8, 11, 0, 0, 0, 0, 1, 0, 7},
            {0, 0, 2, 0, 0, 0, 6, 7, 0}
        };

            int startNode = 0;
            int[] distances = Dijkstra(graph, startNode);
            Console.WriteLine("Shortest distances from node " + startNode + " to
all other nodes:");
            for (int i = 0; i < distances.Length; i++)
            {
                Console.WriteLine($"Node {i}: {distances[i]}");
            }
        }

        private static int[] Dijkstra(int[,] graph, int startNode)
        {
            int n = graph.GetLength(0);
            int[] distances = new int[n];
            bool[] visited = new bool[n];

            for (int i = 0; i < n; i++)
            {
                distances[i] = Infinity;
                visited[i] = false;
            }

            distances[startNode] = 0;

            for (int count = 0; count < n - 1; count++)
            {
                int minDistance = FindMinDistance(distances, visited);
                visited[minDistance] = true;

                for (int i = 0; i < n; i++)
                {
                    if (!visited[i] && graph[minDistance, i] != 0 &&
```

```
Shortest distances from node 0 to all other nodes:
Node 0: 0
Node 1: 4
Node 2: 12
Node 3: 19
Node 4: 21
Node 5: 11
Node 6: 9
Node 7: 8
Node 8: 14
```

3.b Prim’s MST source code and tests

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp5
{
    using System;

    public class PrimsMinimumSpanningTree
    {
        private const int Infinity = int.MaxValue;

        public void Main()
        {
            int[,] graph = {
            {0, 2, 0, 6, 0},
            {2, 0, 3, 8, 5},
            {0, 3, 0, 0, 7},
            {6, 8, 0, 0, 9},
            {0, 5, 7, 9, 0}
        };

            int[] parent = PrimsMST(graph);

            Console.WriteLine("Edges in the minimum spanning tree:");
            for (int i = 1; i < graph.GetLength(0); i++)
            {
                Console.WriteLine($"Edge {parent[i]} - {i}, Weight: {graph[i,
parent[i]]}");
            }
            Console.WriteLine();
        }


        private static int[] PrimsMST(int[,] graph)
        {
            int n = graph.GetLength(0);
            int[] parent = new int[n];
            int[] key = new int[n];
            bool[] mstSet = new bool[n];

            for (int i = 0; i < n; i++)
            {
                key[i] = Infinity;
                mstSet[i] = false;
            }

            key[0] = 0;
            parent[0] = -1;

            for (int count = 0; count < n - 1; count++)
            {
                int u = MinKey(key, mstSet);
                mstSet[u] = true;

                for (int v = 0; v < n; v++)
                {
```

```
Edges in the minimum spanning tree:
Edge 0 - 1, Weight: 2
Edge 1 - 2, Weight: 3
Edge 0 - 3, Weight: 6
Edge 1 - 4, Weight: 5
```

3.c BFT or DFT source code and tests

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp5
{
    internal class DepthFirstTraverse
    {
     class StackX
     {
        private const int SIZE = 20;
        private int[] st;
        private int top;

        public StackX()
        {
            st = new int[SIZE];
            top = -1;
        }

        public void Push(int j)
        {
            st[++top] = j;
        }

        public int Pop()
        {
            return st[top--];
        }

        public int Peek()
        {
            return st[top];
        }

        public bool IsEmpty()
        {
            return (top == -1);
        }
    }

    class Vertex
    {
        public char Label { get; }
        public bool WasVisited { get; set; }

        public Vertex(char lab)
        {
            Label = lab;
            WasVisited = false;
        }
    }

    class Graph
    {
        private const int MAX_VERTS = 20;
```


```
Visits: ABCDE
```

3.d Filled version of the given Big-O table

	Dijkstra's SP	Prim's MST	BFT	**Heap Deletion**
Big-O (Zaman Karmaşıklığı Big-O Notasyonuna Göre)	O(E + V log V)	O(V^2)	O(n)	O(logn)

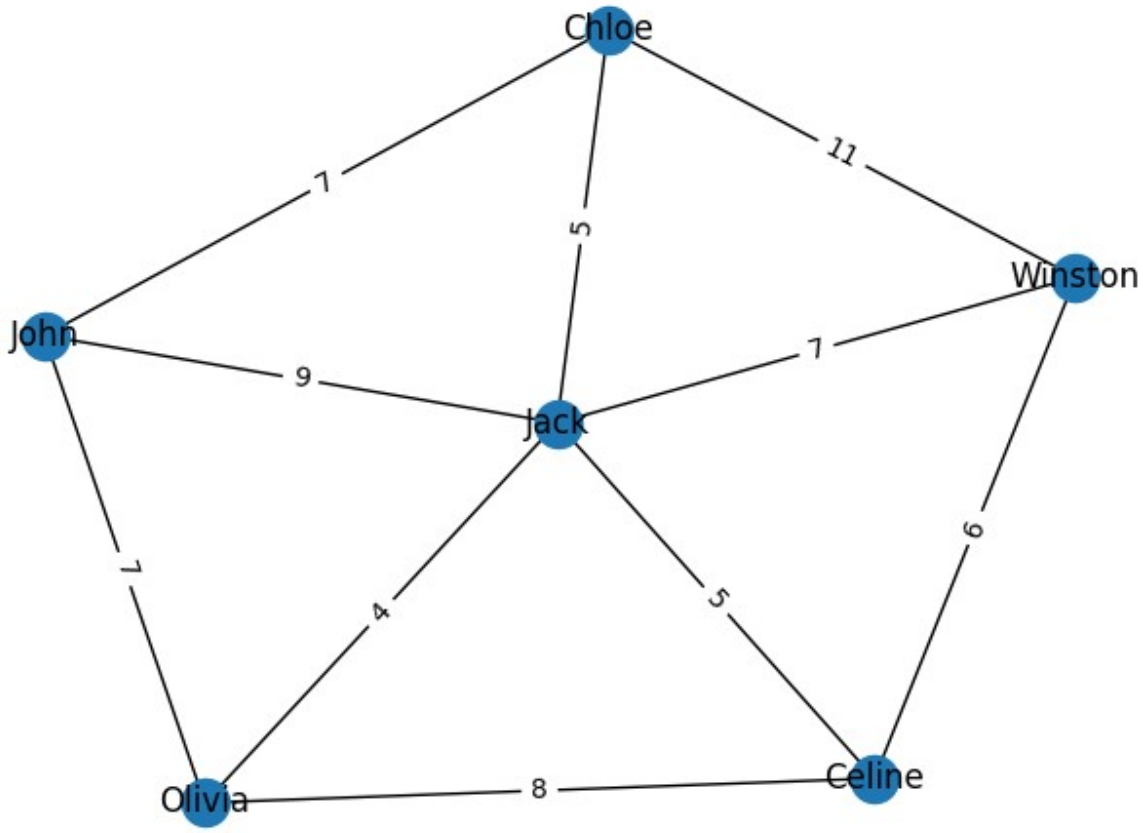
4.a Source code for drawing the given graph in python environment and screenshots

```
import networkx as nx
import matplotlib.pyplot as plt

# Aile üyelerini ve ilişkilerini tanımlayalım
nodes = {
    "Olivia": {},
    "Celine": {},
    "John": {},
    "Chloe": {},
    "Jack": {},
    "Winston": {},
}

edges = [
    ("Olivia", "Celine", {"weight": 8}),
    ("Olivia", "John", {"weight": 7}),
    ("Olivia", "Jack", {"weight": 4}),
    ("Celine", "Jack", {"weight": 5}),
    ("Celine", "Winston", {"weight": 6}),
    ("John", "Chloe", {"weight": 7}),
    ("John", "Jack", {"weight": 9}),
    ("Jack", "Chloe", {"weight": 5}),
    ("Chloe", "Winston", {"weight": 11}),
    ("Jack", "Winston", {"weight": 7}),
]

# Aile ağacı grafiğini oluşturalım
G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)
```


Chloe
Winston
John
Jack
Olivia
Celine
7
5
11
9
7
7
4
5
6
8

4.b Computing distances of nodes

```
for person1 in nodes:
    for person2 in nodes:
        if person1 != person2:
            shortest_path = nx.dijkstra_path(G, source=person1,
target=person2)
            shortest_distance = nx.dijkstra_path_length(G,
source=person1, target=person2)
            print(f"En kısa yol {person1} ile {person2}
arasında: {shortest_path}, Uzaklık: {shortest_distance}")
```

```
En kısa yol Olivia ile Celine arasında: ['Olivia', 'Celine'], Uzaklık: 8
En kısa yol Olivia ile John arasında: ['Olivia', 'John'], Uzaklık: 7
En kısa yol Olivia ile Chloe arasında: ['Olivia', 'Jack', 'Chloe'], Uzaklık: 9
En kısa yol Olivia ile Jack arasında: ['Olivia', 'Jack'], Uzaklık: 4
En kısa yol Olivia ile Winston arasında: ['Olivia', 'Jack', 'Winston'], Uzaklık: 11
En kısa yol Celine ile Olivia arasında: ['Celine', 'Olivia'], Uzaklık: 8
En kısa yol Celine ile John arasında: ['Celine', 'Jack', 'John'], Uzaklık: 14
En kısa yol Celine ile Chloe arasında: ['Celine', 'Jack', 'Chloe'], Uzaklık: 10
En kısa yol Celine ile Jack arasında: ['Celine', 'Jack'], Uzaklık: 5
En kısa yol Celine ile Winston arasında: ['Celine', 'Winston'], Uzaklık: 6
En kısa yol John ile Olivia arasında: ['John', 'Olivia'], Uzaklık: 7
En kısa yol John ile Celine arasında: ['John', 'Jack', 'Celine'], Uzaklık: 14
En kısa yol John ile Chloe arasında: ['John', 'Chloe'], Uzaklık: 7
En kısa yol John ile Jack arasında: ['John', 'Jack'], Uzaklık: 9
En kısa yol John ile Winston arasında: ['John', 'Jack', 'Winston'], Uzaklık: 16
En kısa yol Chloe ile Olivia arasında: ['Chloe', 'Jack', 'Olivia'], Uzaklık: 9
En kısa yol Chloe ile Celine arasında: ['Chloe', 'Jack', 'Celine'], Uzaklık: 10
En kısa yol Chloe ile John arasında: ['Chloe', 'John'], Uzaklık: 7
En kısa yol Chloe ile Jack arasında: ['Chloe', 'Jack'], Uzaklık: 5
En kısa yol Chloe ile Winston arasında: ['Chloe', 'Winston'], Uzaklık: 11
En kısa yol Jack ile Olivia arasında: ['Jack', 'Olivia'], Uzaklık: 4
En kısa yol Jack ile Celine arasında: ['Jack', 'Celine'], Uzaklık: 5
En kısa yol Jack ile John arasında: ['Jack', 'John'], Uzaklık: 9
En kısa yol Jack ile Chloe arasında: ['Jack', 'Chloe'], Uzaklık: 5
En kısa yol Jack ile Winston arasında: ['Jack', 'Winston'], Uzaklık: 7
En kısa yol Winston ile Olivia arasında: ['Winston', 'Jack', 'Olivia'], Uzaklık: 11
En kısa yol Winston ile Celine arasında: ['Winston', 'Celine'], Uzaklık: 6
En kısa yol Winston ile John arasında: ['Winston', 'Jack', 'John'], Uzaklık: 16
En kısa yol Winston ile Chloe arasında: ['Winston', 'Chloe'], Uzaklık: 11
En kısa yol Winston ile Jack arasında: ['Winston', 'Jack'], Uzaklık: 7
```

4.c Traversing the graphs with DFS and BFS

```python
# DFS dolaşımı
start_node_dfs = "Olivia"
dfs_traversal = list(nx.dfs_edges(G, source=start_node_dfs))
print(f"DFS Dolaşımı ({start_node_dfs} başlangıç noktasıyla): {dfs_traversal}")

# BFS dolaşımı
start_node_bfs = "Olivia"
bfs_traversal = list(nx.bfs_edges(G, source=start_node_bfs))
print(f"BFS Dolaşımı ({start_node_bfs} başlangıç noktasıyla): {bfs_traversal}")
```

```
DFS Dolaşımı (Olivia başlangıç noktasıyla): [('Olivia', 'Celine'), ('Celine', 'Jack'), ('Jack', 'John'), ('John', 'Chloe'), ('Chloe', 'Winston')]
BFS Dolaşımı (Olivia başlangıç noktasıyla): [('Olivia', 'Celine'), ('Olivia', 'John'), ('Olivia', 'Jack'), ('Celine', 'Winston'), ('John', 'Chloe')]
```

4.d Purposes of computing SP, BFS and MST

Single-Source Shortest Path (SP):

- Amaç: Herhangi iki kişi arasındaki en kısa yolları ve bu yolların uzunluklarını bulmak.
- Kullanım Alanları:
  - İki kişi arasındaki en kısa yol, iletişim süresini (çağrı süresini) minimize edebilir.
  - İki kişi arasındaki en kısa yol, bir kişinin diğerine ulaşma süresini veya mesafesini belirlemek için kullanılabilir.

Breadth-First Search (BFS):

- Amaç: Ağacı genişletmek ve belirli bir başlangıç düğümünden diğer düğümlere kadar olan en kısa yolları bulmak.
- Kullanım Alanları:
  - BFS, bir kişinin sosyal ağda kaç kişiyle dolaylı olarak bağlantılı olduğunu belirlemede kullanılabilir.
  - Aile üyelerinin genel bağlantılarını anlamak için kullanılabilir.

Minimum Spanning Tree (MST):

- Amaç: Ağdaki tüm düğümleri içeren, kenar ağırlıklarının toplamını minimize eden bir alt ağ (ağaç) bulmak.
- Kullanım Alanları:
  - MST, aile üyeleri arasındaki en güçlü ve önemli bağlantıları belirlemek için kullanılabilir.
  - MST, ailedeki iletişim yollarını optimize ederek toplam iletişim maliyetini minimize etmeye yardımcı olabilir.

4.e Method for accessing to the doctor

```python
import networkx as nx
```

```
nodes = {
    "Olivia": {},
    "Celine": {},
    "John": {},
    "Chloe": {},
    "Jack": {},
    "Winston": {},
}

edges = [
    ("Olivia", "Celine", {"weight": 8}),
    ("Olivia", "John", {"weight": 7}),
    ("Olivia", "Jack", {"weight": 4}),
    ("Celine", "Jack", {"weight": 5}),
    ("Celine", "Winston", {"weight": 6}),
    ("John", "Chloe", {"weight": 7}),
    ("John", "Jack", {"weight": 9}),
    ("Jack", "Chloe", {"weight": 5}),
    ("Chloe", "Winston", {"weight": 11}),
    ("Jack", "Winston", {"weight": 7}),
]

G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)

start_node = "Olivia"

target_doctor = "Dr. Smith"

# SP algoritması uygula
shortest_paths = nx.single_source_dijkstra_path_length(G,
source=start_node, weight="weight")

# Ulaşmak istenilen doktor düğümünün en kısa yolunun uzunluğu
shortest_distance_to_doctor = shortest_paths.get(target_doctor,
float("inf"))

# Ulaşmak istenilen doktor düğümünün en kısa yolu
shortest_path_to_doctor = nx.single_source_dijkstra_path(G,
source=start_node, target=target_doctor)
```

```
print(f"{start_node}'dan {target_doctor}'a en kısa yol:
{shortest_path_to_doctor}, Uzaklık:
{shortest_distance_to_doctor}")
```

SP algoritmalarının karmaşıklığı genellikle O(V^2) veya O(V*log(V)) şeklinde ifade edilir, burada V düğüm sayısını temsil eder..

//Explain how you should modify the method for the given requirement.

Eğer her kişinin belli bir yüzde ile telefonu aç(a)mayacağını düşünüyorsak, algoritmayı bu durumu dikkate alacak şekilde güncellememiz gerekir. Bu durumu ele almak için ağırlıkları güncelleyerek telefonun açılma olasılıklarını göz önüne alabiliriz. Telefonun açılma olasılığına göre ağırlıkların güncellenmesi, daha gerçekçi bir tahminle en kısa yolu bulmamıza yardımcı olabilir.

5.a. Brief comparison of Prim's and Kruskal's algorithms

Prim's Algorithm and Kruskal's Algorithm are both widely used for finding the minimum spanning tree (MST) in connected, undirected graphs with weighted edges. While both are greedy algorithms, they differ in their approach. Prim's Algorithm starts with a single vertex and incrementally grows the MST by adding the closest vertex at each step, relying on a priority queue or min-heap for efficient edge selection. In contrast, Kruskal's Algorithm initially considers all edges and iteratively selects the smallest one that doesn't create a cycle, often employing a disjoint-set data structure. Prim's Algorithm is generally more efficient on dense graphs, whereas Kruskal's Algorithm excels on sparse graphs. Additionally, Prim's requires a designated starting vertex, while Kruskal's is more versatile, not dependent on a specific starting point. The choice between them depends on the characteristics of the graph in question and the desired algorithmic properties.

5.b Implementation of Trie Data Structure and Insert Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp5
{
    public class TrieNode
    {
        public Dictionary<char, TrieNode> children { get; set; }
        public bool isEndOfWord { get; set; }

        public TrieNode()
        {
            children = new Dictionary<char, TrieNode>();
            isEndOfWord = false;
        }
    }

    public class Trie
    {
        private TrieNode root;

        public Trie()
        {
            root = new TrieNode();
        }

        public void insert(string word)
        {
            TrieNode current = root;

            foreach (char c in word)
            {
                if (!current.children.ContainsKey(c))
                {
                    current.children[c] = new TrieNode();
                }

                current = current.children[c];
            }

            current.isEndOfWord = true;
        }

        public bool search(string word)
        {
            TrieNode node = searchNode(word);
            return node != null && node.isEndOfWord;
        }

        private TrieNode searchNode(string word)
        {
            TrieNode current = root;

            foreach (char c in word)
            {
                if (current.children.ContainsKey(c))
```

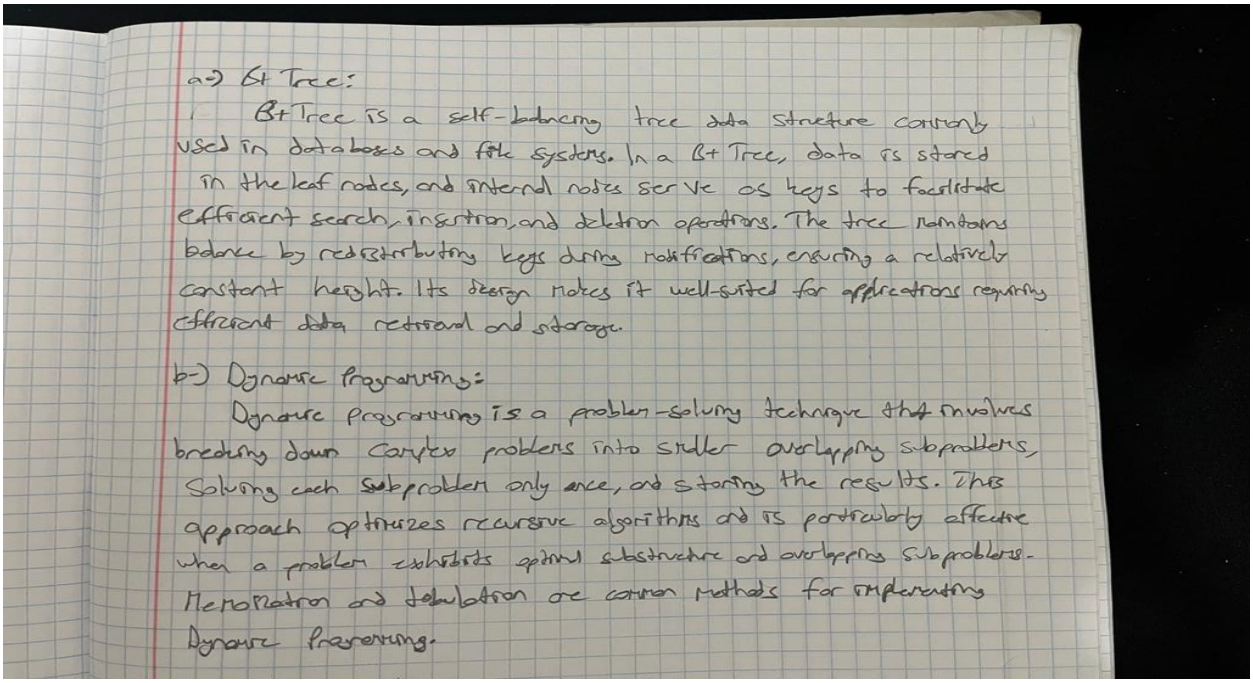
```
Trie trie = new Trie();
trie.insert("çanta");
trie.insert("kitap");
trie.insert("elma");
Console.WriteLine(trie.search("armut"));    // Çıktı: False
Console.WriteLine(trie.search("kitap"));   // Çıktı: True
Console.WriteLine(trie.search("çanta"));      // Çıktı: True
Console.WriteLine(trie.search("silgi"));   // Çıktı: False
Console.WriteLine(trie.search("elma"));     // Çıktı: True
```

```
False
True
True
False
True
```

`TrieNode` class represents a node in the Trie class. It contains a dictionary to store child nodes corresponding to characters and a boolean flag to indicate if it's the end of a word.

Trie class has a root node and includes the insert method, which inserts a word into the Trie by iterating through each character of the word and creating nodes as necessary.

5.c. Explanation of 4 Terms



exhibits optimal substructure and overlapping subproblems. Memoization and tabulation are common methods for implementing Dynamic Programming.

c-) Warshall's Algorithm:

Warshall's Algorithm, or the Floyd-Warshall Algorithm, is used for finding the shortest paths between all pairs of vertices in a weighted directed graph. It employs dynamic programming to iteratively update a matrix representing the shortest paths, considering all possible intermediate vertices. The algorithm accommodates both positive and negative edge weights in connected and disconnected graphs.

d-) Topological Sorting:

Topological Sorting is an arrangement of vertices in a directed acyclic graph such that for every directed edge (u,v), vertex u precedes vertex v in the ordering. This linear ordering, achieved through algorithms like Kahn's or depth-first search, is valuable in applications such as task scheduling and dependency resolution, ensuring a logical sequence without directed cycles

Self-assessment Table

	Points	Estimated Grade	Explanation
1 a) AVL Tree	**10**	10	Yapıldı. AVL ağacına istenen değerler eklenip, çıkarıldı.
1 b) Heap	**10**	10	Yapıldı. Heap veri yapısına istenen değer eklenip çıkarıldı.
2) B-Tree Insertion / AVL Tree Insertion / Red-Black Trees / Huffman Encoding Tree	**10**	10	Yapıldı. Java dilinde B-tree ekleme metodu yazıldı.
3 a) Dijkstra's shortest path code + test	**4**	4	Yapıldı. Dijkstra algoritması ile hesaplandı.
3 b) Prim's MST code + test	**4**	4	Yapıldı. Prim algoritması ile hesaplandı.
3 c) BFT or DFT code + test	**3**	3	Yapıldı. DFT ile dolaşıldı.
3 d) Filling Big-O Table	**4**	4	Yapıldı. Big-O tablosu zaman karmaşıklığına göre hesaplandı.
4 a) Graph Drawing	**3**	3	Yapıldı. Graf yapısı çizildi.
4 b) Finding Shortest Paths with Dijkstra's	**3**	3	Yapıldı. En kısa yol Dijkstra algoritması ile hesaplandı.
4 c) DFS and BFS	**3**	3	Yapıldı. DFS ve BFS dolaşım yöntemleriyle dolaşıldı.
4 d) Thinking and Writing aims of given algorithms	**3**	3	Yapıldı. Algoritmanın hangi amaçlarla kullanılabileceği yazıldı.

4 e) Real Life Application	**3**	3	Yapıldı. Çözüm önerildi.
5 i) Comparison (Prim's & Kruskal's Algorithm)	**5**	5	Yapıldı. Avantajları ve dezavantajları yazıldı.
5 ii) Trie Data Structure and Insertion Method	**5**	5	Yapıldı. Trie veri yapısı düğümü vee leman ekleme metodu yazıldı.
5 iii) Explanations of 4 terms	**10**	10	B+Tree, Warshall, Dynamic programming ve topological sorting açıklandı.
Demo Video for Source Codes and Tests of Q2 Q3a, Q3b, Q3c and Q5.ii .	**10**	10	Yapıldı. İstenen soruların videosu yapan kişiler tarafından açıklandı.
Self-assessment Table	**10**	9	Yapıldı.
Total	**100**	99	