



EGE UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

204 DATA STRUCTURES (3+1)

2020–2021 FALL SEMESTER

PROJECT-4 REPORT

(GRAPHS, GRAPH ALGORITHMS, TREES and OTHER SUBJECTS)

DELIVERY DATE

16/02/2021 – Up to 14:30

PREPARED BY

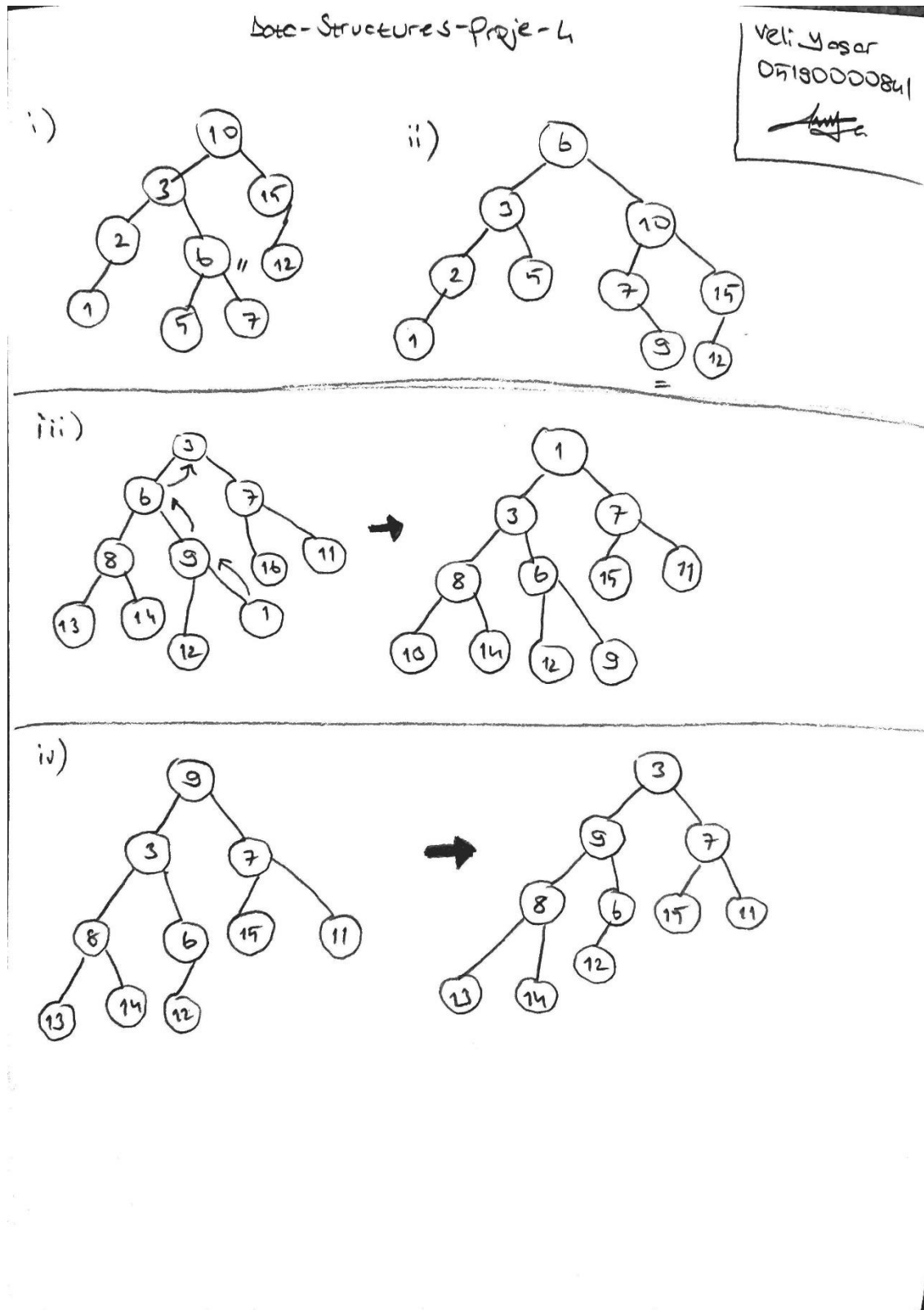
05190000841, Veli Yaşar

İçindekiler

GRAPHS, GRAPH ALGORITHMS, TREES and OTHER SUBJECTS	2
1. AVL Tree Insertions (drawing), Heap Node Insertion and Removing.....	2
2. B-Tree Insertion Method [Alternatives: AVL-Tree Insertion, Red-Black Trees, Huffman Encoding Tree]	2
2.1 Kalemle Yazılan Algoritma Görüntüsü.....	3
2.2 Yöntemin (ekleme işleminin) anlatımı / açıklaması / adımları	6
2.3 Kaynak Kod	6
2.4 (İşletim + Test) Ekran Görüntüsü	9
3. Huffman Encoding Tree Generation.....	10
4. Graph Algorithms	10
4.1 Dijkstra's Shortest Path [Source code + Screenshot for test]	11
4.2 Prim's MST [Source code + Screenshot for test]	13
4.3 BFT (Breadth-First Traverse) or DFT (Depth-First Traverse) [Source code + Screenshot for test]	15
4.4 Big-O Table (Time Complexities)	17
5. Graph Drawing and Finding Shortest Path in Python.....	17
5.1 Graph Drawing.....	17
5.2 Finding Shortest Path	18
5.3 Vertex Removing and repeating previous steps (Drawing and Finding SP)	18
6. Comparison of MST Algorithms.....	18
7. Definitions	18

GRAPHS, GRAPH ALGORITHMS, TREES and OTHER SUBJECTS


1. AVL Tree Insertions (drawing), Heap Node Insertion and Removing



2. B-Tree Insertion Method [Alternatives: AVL-Tree Insertion, Red-Black Trees, Huffman Encoding Tree]

2.1 Kalemle Yazılan Algoritma Görüntüsü

- AVL-Tree-Insertion -

Veli Yasar
07150000841


```
class Node {
    int key, height;
    Node left, right;
    Node(int d) {
        key = d;
        height = 1;
    }
}

class AVLTree {
    Node root;
    int Height(Node N) {
        if (N == null) return 0;
        return N.height;
    }
    int Max(int a, int b) {
        return a > b ? a : b;
    }
    Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;
        x.right = y; // rotation
        y.left = T2;
        y.height = Max(Height(y.left), Height(y.right)) + 1;
        x.height = Max(Height(x.left), Height(x.right)) + 1;
        // yükseklik güncelleme ^
        return x;
    }
}
```

Date-Structures- AVL Tree - Insertion

Veli Yasar
0518000008
41


```
Node left.rotate (Node x) {
```

```
    Node y = x.right;
```

```
    Node T2 = y.left;
```

```
    x.left = x
```

```
    y.right = T2 // Rotasyon
```

```
    x.height = Max (Height(x.left), Height(x.right)) + 1;
```

```
    y.height = Max (Height(y.left), Height(y.right)) + 1;
```

```
    return y; }
```

// yükseklikleri güncelle

```
int GetBalance (Node N) {
```

```
    if (N == null) return 0;
```

```
    return Height (N.left) - Height (N.right); }
```

```
Node Insert (Node node, int key) {
```

```
    // Standart BST ekleme işlemi
```

```
    if (node == null) return (new Node(key));
```

```
    if (key < node.key) node.left = Insert (node.left, key);
```

```
    else if (key > node.key) node.right = Insert (node.right, key);
```

```
    else return node;
```

```
    // ağırlar için yükseklik güncelleme
```

```
    node.height = 1 + Max (Height (node.left), Height (node.right));
```

DS - AVL Tree - Insertion -

Veli Yasar
05190000841



```
int balance = GetBalance(node);
```

```
// Düzümde dengenin bozulması durumunda yapılacaklar
```

```
// Sol-Sol durumu
```

```
if (balance > 1 && key < node->left->key) return RightRotate(node);
```

```
// Sağ-Sağ durumu
```

```
if (balance < -1 && key > node->right->key) return LeftRotate(node);
```

```
// Sol-Sağ durumu
```

```
if (balance > 1 && key > node->left->key) node->left = LeftRotate(node->left);  
return RightRotate(node);
```

```
// Sağ-Sol durumu
```

```
if (balance < -1 && key < node->right->key) node->right = RightRotate(node->right);  
return LeftRotate(node);
```

```
return node;
```

```
// eğer denge bozulmadıysa eski düğüm döndürülür.
```

```
}
```

```
void PreOrder(Node node) {
```

```
    if (node != null) {
```

```
        CW(node->key + " ");
```

```
        PreOrder(node->left);
```

```
        PreOrder(node->right); } }
```

```
}
```

2.2 Yöntemin (ekleme işleminin) anlatımı / açıklaması / adımları

AVL Ağacı, sol ve sağ alt ağaçların yükseklikleri arasındaki farkın tüm düğümler için birden fazla olamayacağı kendi kendini dengeleyen bir İkili Arama Ağacıdır(BST). AVL Ağacındaki her ekleme işleminde standart BST'lerden farklı olarak AVL özelliğini bozan düğümler için dengeleme işlemi yapılır. e.g. Left/Right Rotation.

Bu uygulama, yeni bir düğüm eklemek için özyinelemeli BST insertion kullanır. Burada tüm ancestor'lara aşağıdan yukarıya doğru birer pointer atarız. Özyinelemeli kod yukarı hareket ederek yeni eklenen düğümün tüm ancestor'larını dolaşır.

- 1) Normal BST ekleme işlemi gerçekleştirilir.
- 2) Geçerli düğüm, yeni eklenen düğümün atalarından biri olmalıdır. Mevcut düğümün yüksekliği güncellenir.
- 3) Mevcut düğümün denge faktörü (sol alt ağaç yüksekliği eksi sağ alt ağaç yüksekliği) alınır.
- 4) Denge faktörü 1'den büyükse, mevcut düğüm dengesizdir veya Sol-Sol ya da Sol-Sağ durum söz konusudur. Sol durumda olup olmadığını kontrol etmek için, yeni eklenen anahtar, sol alt ağaç kökündeki anahtarla karşılaştırılır.
- 5) Denge faktörü -1'den küçükse, mevcut düğüm dengesizdir ve ya Sağ-Sağ durumda ya da Sağ-Sol durum söz konusudur. Sağ-Sağ durumda olup olmadığını kontrol etmek için, yeni eklenen anahtar sağ alt ağaç kökündeki anahtarla karşılaştırılır.

2.3 Kaynak Kod

//Alıntı: [geeksforgeeks.org](https://www.geeksforgeeks.org)

```
class Program
{
    public static void Main(string[] args)
    {
        var tree = new AVLTree();
        var random = new Random();

        for (int i = 0; i < 6; i++)
        {
            tree.root = tree.Insert(tree.root, random.Next(10, 99));
        }

        Console.WriteLine("Ağacın Preorder dolaşılması: ");
        tree.PreOrder(tree.root);
        Console.WriteLine();

        Console.ReadKey();
    }
}

public class Node
{
    public int key, height;
    public Node left, right;

    public Node(int d)
    {
        key = d;
    }
}
```

```

        height = 1;
    }
}

public class AVLTree
{
    internal Node root;

    // Ağacın yüksekliğini bul
    private int Height(Node N)
    {
        if (N == null)
            return 0;

        return N.height;
    }

    private int Max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    private Node rightRotate(Node y)
    {
        Node x = y.left;
        Node T2 = x.right;

        // Rotasyon işlemi
        x.right = y;
        y.left = T2;

        // Yükseklik güncelleme
        y.height = Max(Height(y.left), Height(y.right)) + 1;
        x.height = Max(Height(x.left), Height(x.right)) + 1;

        // Yeni kök
        return x;
    }

    Node leftRotate(Node x)
    {
        Node y = x.right;
        Node T2 = y.left;

        // Rotasyon işlemi
        y.left = x;
        x.right = T2;

        // Yükseklikleri güncelle
        x.height = Max(Height(x.left), Height(x.right)) + 1;
        y.height = Max(Height(y.left), Height(y.right)) + 1;

        // Yeni kök
        return y;
    }

    // N düğümü için denge faktörü döndür
    private int GetBalance(Node N)
    {
        if (N == null)
            return 0;
    }
}

```



```

        return Height(N.left) - Height(N.right);
    }

    public Node Insert(Node node, int key)
    {
        // Standart BST ekleme işlemi
        // Birden fazla aynı değere izin verilmiyor
        if (node == null)
            return (new Node(key));

        if (key < node.key)
            node.left = Insert(node.left, key);
        else if (key > node.key)
            node.right = Insert(node.right, key);
        else
            return node;

        // Ancestor için yükseklik güncelleme
        node.height = 1 + Max(Height(node.left), Height(node.right));

        int balance = GetBalance(node);

        // Düğümde dengenin bozulması durumunda yapılacaklar

        // Sol-Sol durumu
        if (balance > 1 && key < node.left.key)
            return rightRotate(node);

        // Sağ-Sağ durumu
        if (balance < -1 && key > node.right.key)
            return leftRotate(node);

        // Sol-Sağ durumu
        if (balance > 1 && key > node.left.key)
        {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

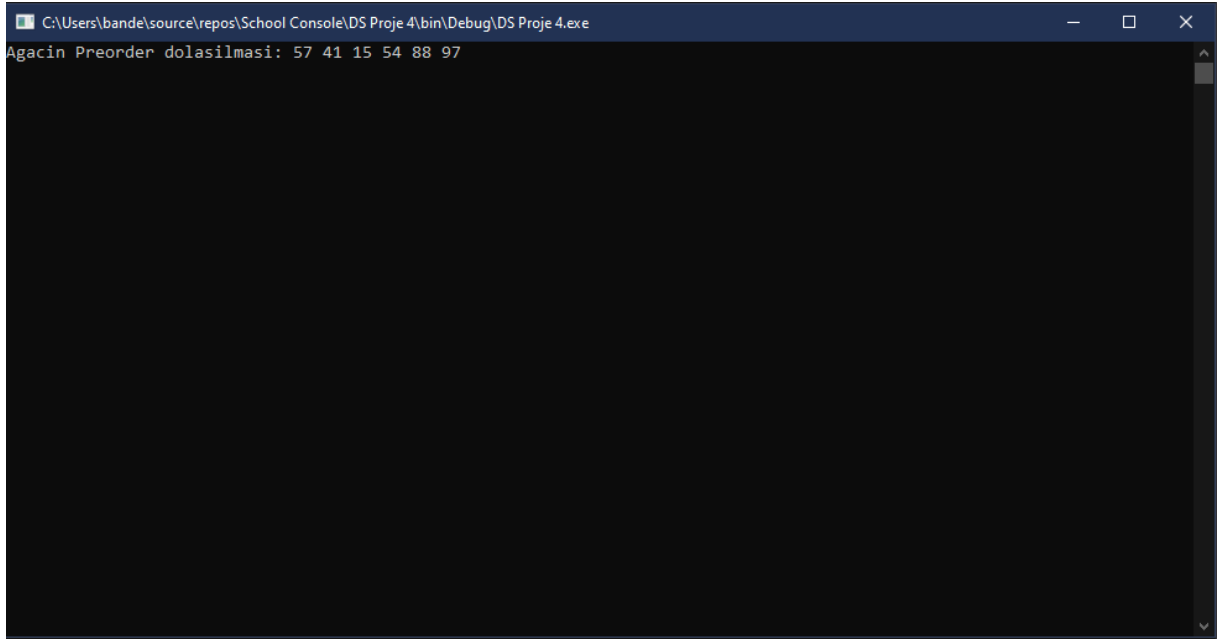
        // Sağ-Sol durumu
        if (balance < -1 && key < node.right.key)
        {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        return node;
    }

    // PreOrder dolaşım
    internal void PreOrder(Node node)
    {
        if (node != null)
        {
            Console.Write(node.key + " ");
            PreOrder(node.left);
            PreOrder(node.right);
        }
    }
}

```

2.4 (İřletim + Test) Ekran Görüntüsü

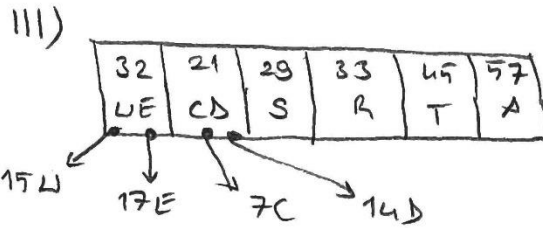


3. Huffman Encoding Tree Generation

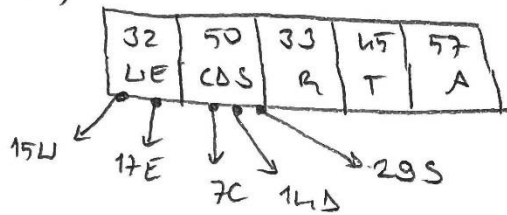
Huffman - Encoding - Tree

Veli Yager
05190000841

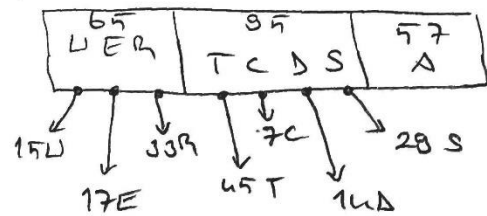
3)



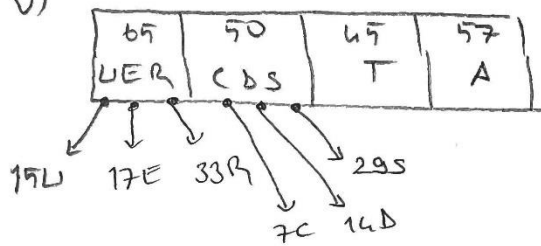
IV)



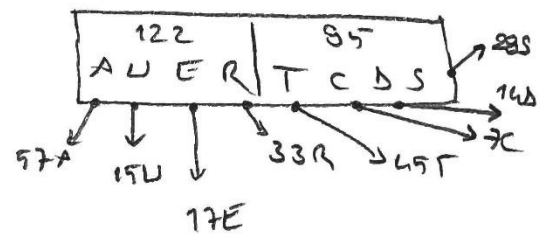
VI)



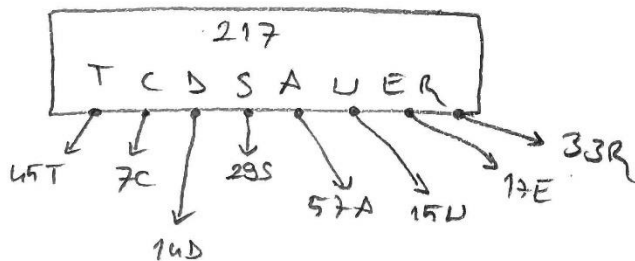
V)



VII)



VIII)



4. Graph Algorithms

4.1 Dijkstra's Shortest Path [Source code + Screenshot for test]

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

namespace ConsoleApp1
{
    class Program
    {
        public static int INF = 1000;

        static void Main(string[] args)
        {
            int N = 5;
            int SRC = 2;

            int[,] cost = {
                { INF, 5, 3, INF, 2},
                { INF, INF, 2, 6, INF},
                { INF, 1, INF, 2, INF},
                { INF, INF, INF, INF, INF},
                { INF, 6, 10, 4, INF} };

            int[] distances = new int[N];

            int[] previous = Distance(N, cost, distances, SRC);

            for (int i = 0; i < distances.Length; ++i)
                if (distances[i] != INF)
                    Console.WriteLine(distances[i]);
                else Console.WriteLine("INF");

            int DEST = 1;
            Console.WriteLine("\n Shortest path from " + SRC + " to " + DEST + "
(straight):");
            printShortestPathStraight(DEST, previous);
            Console.WriteLine("\n\n Shortest path from " + SRC + " to " + DEST + "
(reverse) :");
            printShortestPathReverse(DEST, previous);
            Console.ReadKey();
        }

        public static int[] Distance(int N, int[,] cost, int[] D, int src)
        {
            int w, v, min;

            bool[] visited = new bool[N];

            int[] previous = new int[N]; //for tracking shortest paths (güzergah)

            //initialization of D[], visited[] and previous[] arrays according to src
            node
            for (v = 0; v < N; v++)
            {
                if (v != src)
                {
                    visited[v] = false;

```

```

        D[v] = cost[src, v];
        if (D[v] != INF) //there is a connection between src and v
        {
            previous[v] = src;
        }
        else //no path from source
        {
            previous[v] = -1;
        }
    }
    else
    {
        visited[v] = true;
        D[v] = 0;
        previous[v] = -1;
    }
}

// Searching for shortest paths
for (int i = 0; i < N; ++i)
{
    min = INF;
    for (w = 0; w < N; w++)
        if (!visited[w])
            if (D[w] < min)
            {
                v = w;
                min = D[w];
            }

    visited[v] = true;

    for (w = 0; w < N; w++)
        if (!visited[w])
            if (min + cost[v, w] < D[w])
            {
                D[w] = min + cost[v, w];
                previous[w] = v; //update the path info
            }
}

return previous;
}

public static void printShortestPathStraight(int dest, int[] previous)
{
    Stack<int> pathStack = new Stack<int>();

    int current = dest;
    pathStack.Push(current);

    while (previous[current] != -1)
    {
        current = previous[current];
        pathStack.Push(current);
    }

    if (pathStack.Count == 1)
    {
        Console.WriteLine(" NO PATH");
        return;
    }
}

```

```

    }

    while (pathStack.Count > 0)
    {
        Console.WriteLine(" -> " + pathStack.Pop());
    }
}

public static void printShortestPathReverse(int dest, int[] previous)
{
    int current = dest;
    Console.WriteLine(dest + " <- ");

    while (previous[current] != -1)
    {
        current = previous[current];
        Console.WriteLine(current + " <- ");
    }
}
} } }

```

```

INF
1
0
2
INF

Shortest path from 2 to 1 (straight):
-> 2 -> 1

Shortest path from 2 to 1 (reverse) :
1 <- 2 <-

```

4.2 Prim's MST [Source code + Screenshot for test]

```

class PrimsMST
{
    public static 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 } };

        primMST(graph);

        Console.ReadKey();
    }

    // Number of vertices in the graph
    static int V = 5;

    // A utility function to find the vertex with minimum key value
    // from the set of vertices not yet included in PrimsMST
    static int minKey(int[] key, bool[] mstSet)
    {
        // Initialize min value
        int min = int.MaxValue, min_index = -1;
    }
}

```

```

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min)
            {
                min = key[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed PrimsMST stored in
    parent[]
    static void printMST(int[] parent, int[,] graph)
    {
        Console.WriteLine("Edge \tWeight");
        for (int i = 1; i < V; i++)
            Console.WriteLine(parent[i] + " - " + i + "\t" + graph[i,
    parent[i]]);
    }

    // Function to construct and print PrimsMST for a graph represented
    // using adjacency matrix representation
    static void primMST(int[,] graph)
    {
        // Array to store constructed PrimsMST
        int[] parent = new int[V];

        // Key values used to pick minimum weight edge in cut
        int[] key = new int[V];

        // To represent set of vertices included in PrimsMST
        bool[] mstSet = new bool[V];

        // Initialize all keys as INFINITE
        for (int i = 0; i < V; i++)
        {
            key[i] = int.MaxValue;
            mstSet[i] = false;
        }

        // Always include first 1st vertex in PrimsMST.
        // Make key 0 so that this vertex is picked as first vertex
        // First node is always root of PrimsMST
        key[0] = 0;
        parent[0] = -1;

        // The PrimsMST will have V vertices
        for (int count = 0; count < V - 1; count++)
        {
            // Pick the minimum key vertex from the set of vertices
            // not yet included in PrimsMST
            int u = minKey(key, mstSet);

            // Add the picked vertex to the PrimsMST Set
            mstSet[u] = true;

            // Update key value and parent index of the adjacent
            vertices of the picked vertex.

```

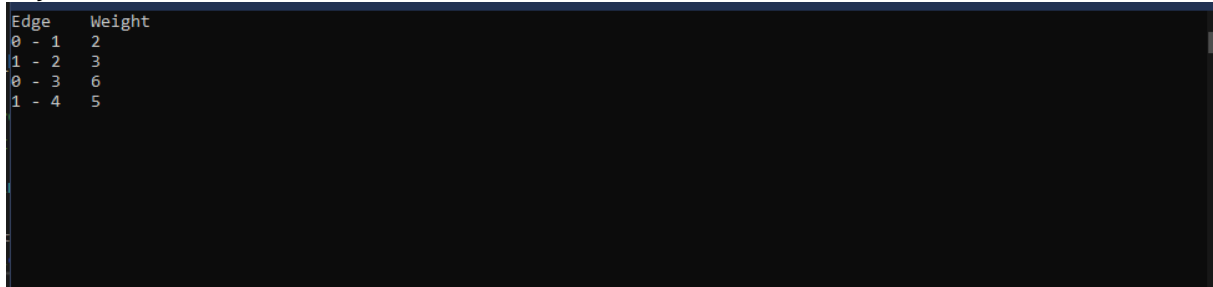
```

// Consider only those vertices which are not yet included
in PrimsMST
for (int v = 0; v < V; v++)

    // graph[u][v] is non zero only for adjacent
    // mstSet[v] is false for vertices not yet included
    // the key only if graph[u][v] is smaller than
    key[v]
    if (graph[u, v] != 0 && mstSet[v] == false &&
graph[u, v] < key[v])
    {
        parent[v] = u;
        key[v] = graph[u, v];
    }

    printMST(parent, graph);
}
}

```



Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

4.3 BFT (Breadth-First Traverse) [Source code + Screenshot for test]

```

class Graph
{
    static void Main(string[] args)
    {
        var graph = new Graph(4);

        graph.AddEdge(0, 1);
        graph.AddEdge(0, 2);
        graph.AddEdge(1, 2);
        graph.AddEdge(2, 0);
        graph.AddEdge(2, 3);
        graph.AddEdge(3, 3);

        Console.WriteLine("Following is Breadth First Traversal(starting from vertex
2): \n");
        graph.BFS(2);

        Console.ReadKey();
    }

    // Nnumber of vertices
    private int V;

    //Adjacency Lists
    LinkedList<int>[] adj;

    public Graph(int V)
    {
        adj = new LinkedList<int>[V];
    }
}

```



```

        for (var i = 0; i < adj.Length; i++)
        {
            adj[i] = new LinkedList<int>();
        }
        this.V = V;
    }

    // Function to add an edge into the graph
    public void AddEdge(int v, int w)
    {
        adj[v].AddLast(w);
    }

    // Prints BFS traversal from a given source s
    public void BFS(int s)
    {
        // Mark all the vertices as not visited(By default set as false)
        var visited = new bool[V];
        for (var i = 0; i < V; i++)
            visited[i] = false;

        // Create a queue for BFS
        var queue = new LinkedList<int>();

        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.AddLast(s);

        while (queue.Any())
        {
            // Dequeue a vertex from queue and print it
            s = queue.First();
            Console.Write(s + " ");
            queue.RemoveFirst();

            // Get all adjacent vertices of the dequeued vertex s.
            // If a adjacent has not been visited, then mark it visited and
enqueue it
            var list = adj[s];

            foreach (var val in list)
            {
                if (visited[val]) continue;
                visited[val] = true;
                queue.AddLast(val);
            }
        }
    }
}

```

```

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

```

4.4 Big-O Table (Time Complexities)

	Dijkstra's SP	Prim's MST	BFT	Heap Insertion
Big-O (Zaman Karmaşıklığı Big-O Notasyonuna Göre)	$O(n^2)$	$O(n^2)$	$O(n)$	$O(\log n)$

5. Graph Drawing and Finding Shortest Path in Python

5.1 Graph Drawing

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

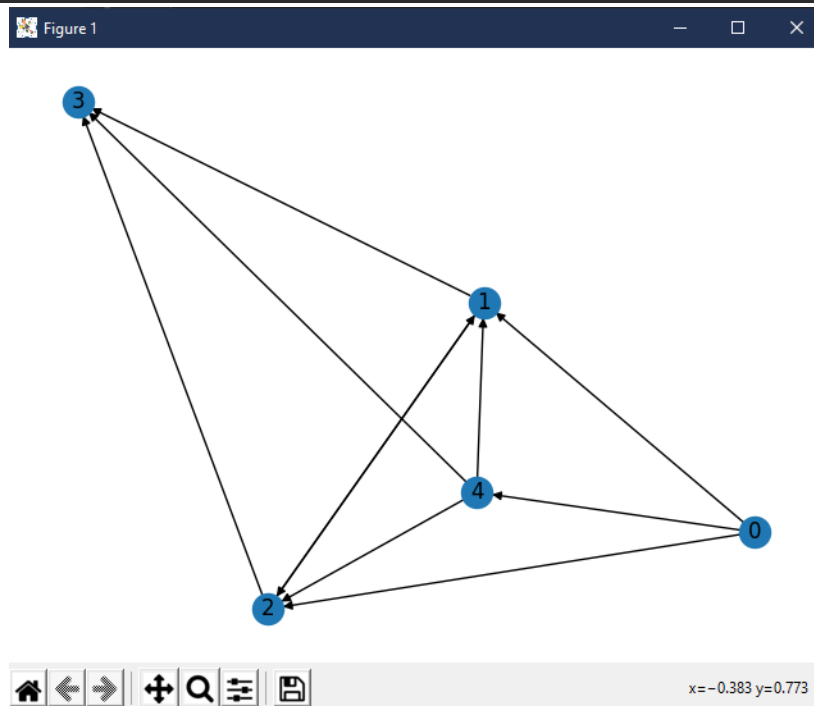
G = nx.DiGraph()

list_nodes = [0, 1, 2, 3, 4]
G.add_nodes_from(list_nodes)
print("Düğümmler: ", G.nodes())

list_arcs = [(0, 1, 5.0), (0, 4, 2.0), (4, 3, 3.0), (2, 3, 2.0), (1, 2, 2.0),
             (2, 1, 1.0), (4, 2, 10.0), (1, 3, 6.0), (0, 2, 3.0), (4, 1, 6.0)]
G.add_weighted_edges_from(list_arcs)
print("Köşeler: ", G.edges)

nx.draw(G, with_labels=1)
plt.show()
```

```
main (1) x
"C:\Program Files (x86)\Python37-32\python.exe" C:/Users/bande/PycharmProjects/EverDisposed/main.py
Düğümmler: [0, 1, 2, 3, 4]
Köşeler: [(0, 1), (0, 4), (0, 2), (1, 2), (1, 3), (2, 3), (2, 1), (4, 3), (4, 2), (4, 1)]
```



5.2 Finding Shortest Path

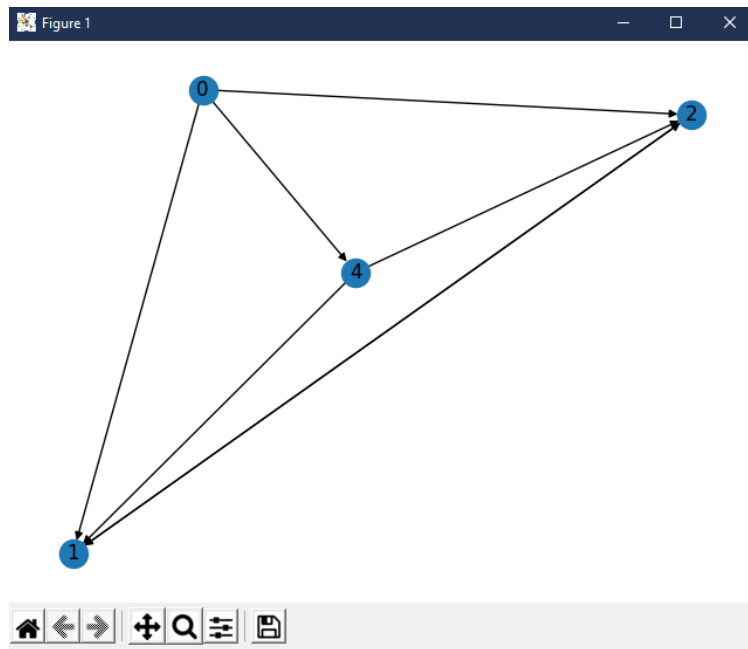
```
shortest_pathTo_0 = nx.dijkstra_path(G, source=4, target=0)
shortest_pathTo_1 = nx.dijkstra_path(G, source=4, target=1)
shortest_pathTo_2 = nx.dijkstra_path(G, source=4, target=2)
shortest_pathTo_3 = nx.dijkstra_path(G, source=4, target=3)

print(shortest_pathTo_0, shortest_pathTo_1, shortest_pathTo_2, shortest_pathTo_3)
```

5.3 Vertex Removing and repeating previous steps (Drawing and Finding SP)

```
G.remove_node(3)

nx.draw(G, with_labels=1)
plt.show()
```



6. Comparison of MST Algorithms

Prim's algoritması $O(n^2)$ zaman karmaşıklığına sahipken Kruskal's $O(\log n)$ 'dir. Prim's en küçük uzaklığı bulmak için birden fazla düğümü kontrol ederken Kruskal's sadece bir düğümü kontrol eder. Prim's algoritması daha yoğun graph'larda hızlı çalışırken Kruskal's algoritması daha seyrek graph'larda daha hızlı çalışır. Prim's graph'taki herhangi bir köşeden MST'yi oluşturmaya başlarken Kruskal's en küçük ağırlığı taşıyan köşeden MST'yi oluşturmaya başlar.

7. Definitions

1. Dynamic Programming

Dinamik programlama algoritmaları alandan ödün verilerek zamandan kazanılmasını sağlar. Karmaşık bir problemi tekrarlanan alt problemlere bölerek, her bir alt problemi yalnız bir kere çözüp daha sonra bu çözümü kaydederek karmaşık problemin çözümünde kullanma yöntemidir. Optimizasyon problemlerinin çözümünde yaygın olarak kullanılır.

Örneğin, Fibonacci sayılarında n . sayının döndürülmesi için kullanılan genel algoritma üstel zaman karmaşıklığına sahipken dinamik optimize edildiğinde $O(n)$ karmaşıklığına sahip olarak performans artışı sağlar. Bunu tekrarlanan alt problemleri hatırlayıp eşleştirerek gerçekleştirir.

2. Warshall's Algorithm

Ağırlıklı graph'larda en kısa yolları bulmaya yarayan $O(n^3)$ zaman karmaşıklığına sahip algoritmadır. Warshall's bir dinamik programlama örneğidir. Yani ana problemi çözmek için problemi parçalara ayırarak cevaplar oluşturur. Sonra bu alt cevapları birleştirerek ana problemi çözmeye çalışır. Bazı en kısa yol bulma algoritmaları gibi tek bir kaynak kullanmak yerine girilen çizgedeki her köşe çifti için en kısa yolları bulmayı sağlar. Bu sayede çok duraklı problemler için elverişli çözüm yolu sunar.

3. Quadratic Probing

Hash Table'da meydana gelebilen hash çakışmasını(farklı hash'lerin aynı değere sahip olması) çözmeye yarar. İlk hash indeksini alıp parabol içinde açık bir yuva bulunana kadar rastgele bir polinomun ardışık değerlerini ekleyerek çalışır. Open-addressing çizelgelerinde etkili bir algoritma olarak çalışır. Kusursuz olmasa da lineer çözümlerin sebep olduğu kümelenmeleri önlemekte daha iyidir. Ayrıca bazı referans bölgelerini koruduğu için iyi bir önbellek sağlar.

4. B+Tree

B-Tree gibi dinamik çok düzeyli endeksleme uygulamasında kullanılır. B-Tree'nin endeksleme için kullandığı veri işaretçilerinin belleği tıkayarak sistemi yavaşlatması ve B-Tree'de işlenebilecek düğüm girdi sayısının azalması gibi dezavantajları gidermeyi sağlar. B+Tree bunu işaretçileri sadece yaprak düğümlerinde saklayarak gerçekleştirir. Sonuç olarak diskten kayıtları çekerken daha hızlı davranır.

5. 2-3-4 Tree

Sözlüklerde kullanılan kendi kendini dengeleyen bir veri yapısıdır. Sayılar her çocuklu düğümün kaç çocuğu olduğunu gösterir: 2-düğüm bir elemana ve iki çocuğu sahiptir, 3-düğüm iki elemana ve üç çocuğa sahiptir, 4-düğüm üç elemana ve dört çocuğa sahiptir.

2-3-4 ağaçları B-Tree gibi $O(\log n)$ zamanda arama, ekleme ve silme işlemlerini yapabilirler. 2-3-4 ağaçlarının bütün yaprak düğümleri aynı derinliktedir ve veriler sıralanmış halde tutulur. Red-Black Tree'nin eşdeğeri konumundadırlar ve birbirlerine dönüştürülebilirler.