

**EGE UNIVERSITY**

**FACULTY OF ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

**204 DATA STRUCTURES (3+1)**

**2023–2024 FALL SEMESTER**

**PROJECT-4 REPORT**

**(GRAPHS, GRAPH ALGORITHMS, TREES and OTHER SUBJECTS)**

**DELIVERY DATE**

06/01/2024

**PREPARED BY**

05220000981 Burak Taşkın

05210000278 İsmail Erkan

İçindekiler

[1.a Drawing of new AVL Trees after inserting requested values 4](#_Toc154511673)

[1.b Drawing of new Heaps after inserting and removing requested values 5](#_Toc154511674)

[2.a B-Tree insertion method (or AVL-Tree insertion method/Red-Black tree/Huffman encoding tree) code 6](#_Toc154511675)

[2.b Explanation of B-Tree insertion method steps 8](#_Toc154511676)

[2.c B-Tree insertion test 8](#_Toc154511677)

[3.a Dijkstra’s SP source code and tests 8](#_Toc154511678)

[3.b Prim’s MST source code and tests 11](#_Toc154511679)

[3.c BFT or DFT source code and tests 13](#_Toc154511680)

[3.d Filled version of the given Big-O table 17](#_Toc154511681)

[4.a Source code for drawing the given graph in python environment and screenshots 17](#_Toc154511682)

[4.b Computing distances of nodes 18](#_Toc154511683)

[4.c Traversing the graphs with DFS and BFS 19](#_Toc154511684)

[4.d Purposes of computing SP, BFS and MST 20](#_Toc154511685)

[4.e Method for accessing to the doctor 21](#_Toc154511686)

[5.a. Brief comparison of Prim’s and Kruskal’s algorithms 23](#_Toc154511687)

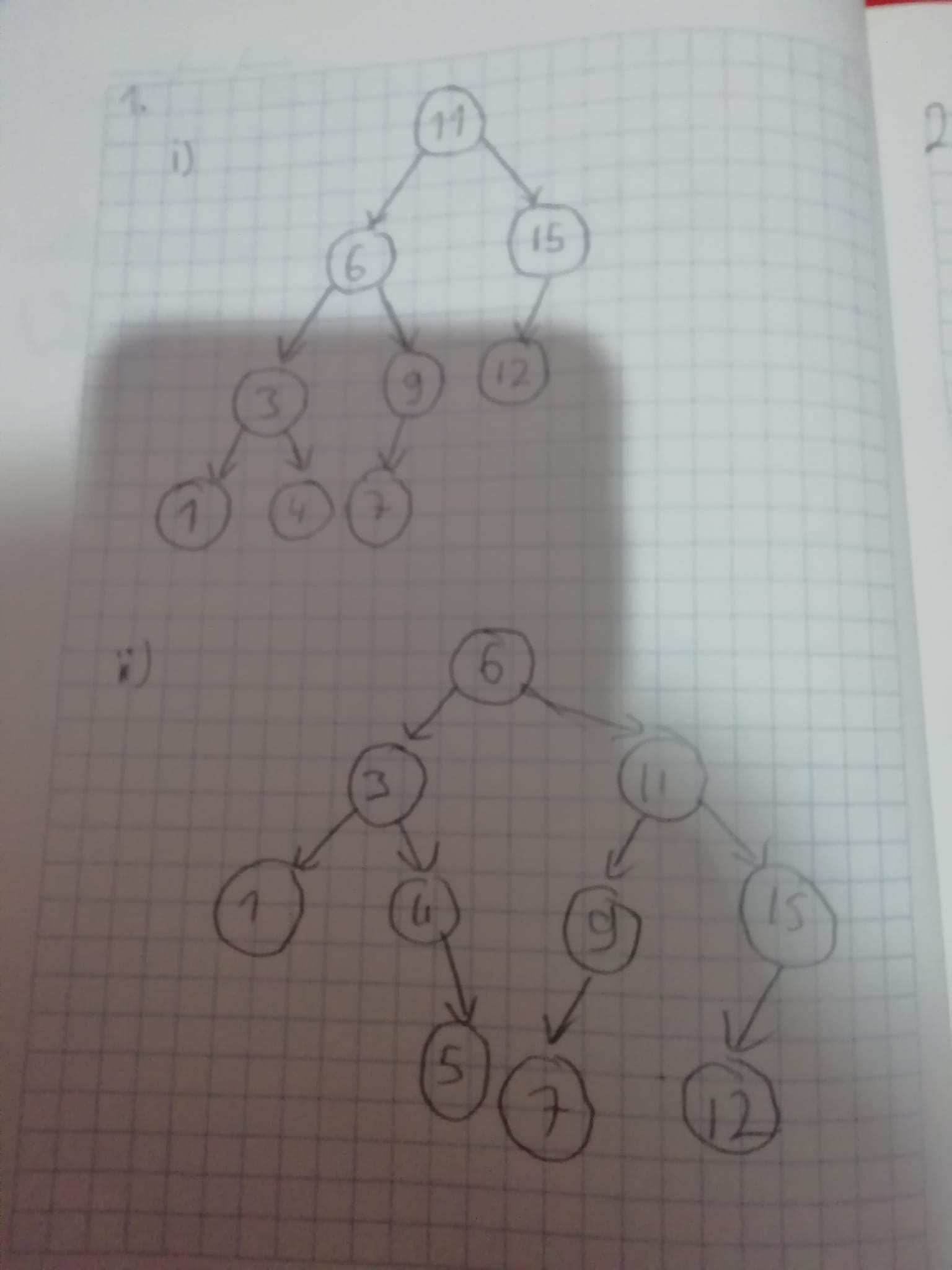
[5.b Implementation of Trie Data Structure and Insert Method 23](#_Toc154511688)

[5.c. Explanation of 4 Terms 24](#_Toc154511689)

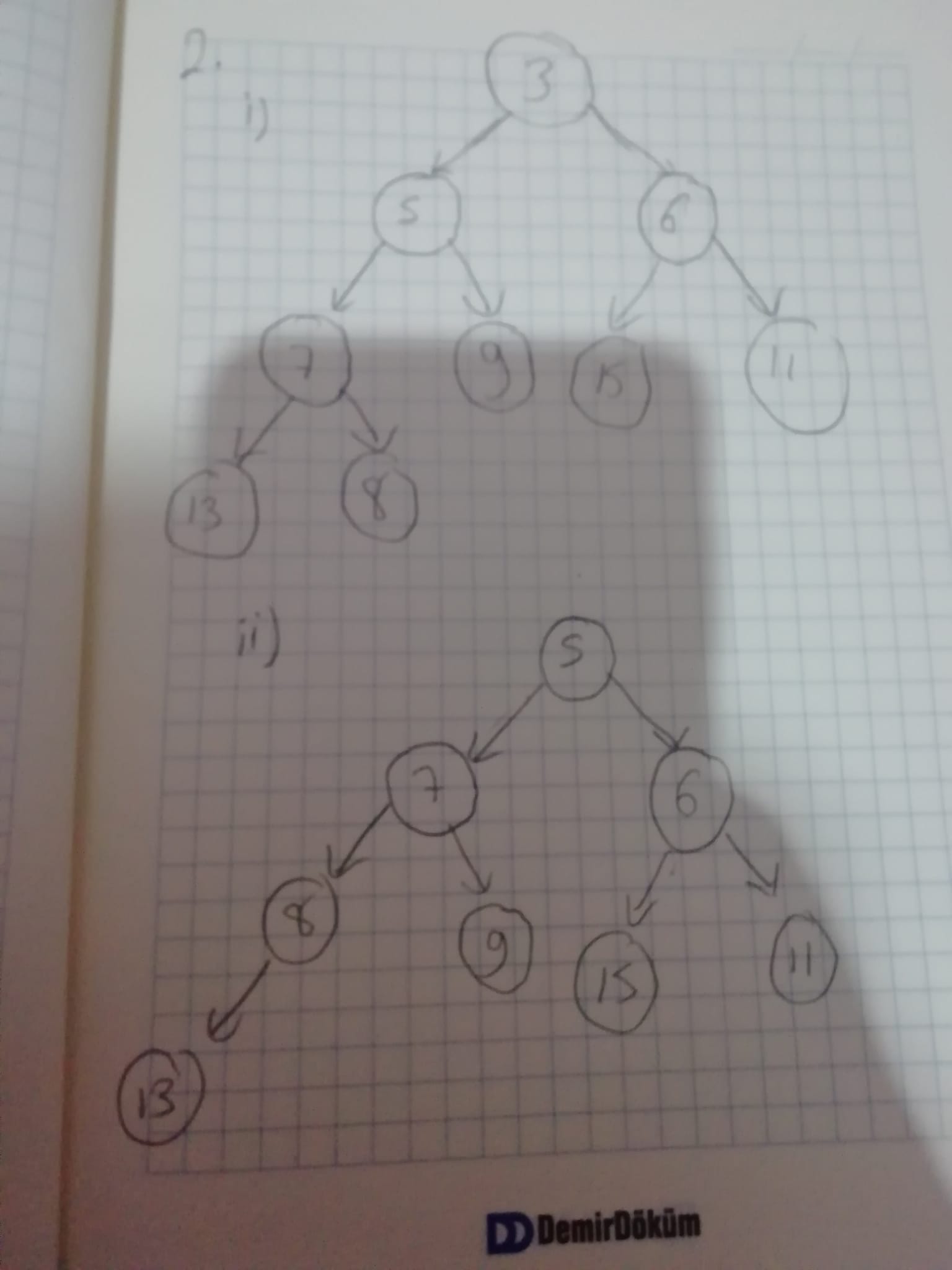
[Self-assessment Table 25](#_Toc154511690)

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(int key)  {  if (root == null)  {  root = new BTreeNode(degree, true);  root.keys[0] = key;  root.n = 1;  }  else  {  if (root.n == 2 \* degree - 1)  {  BTreeNode newRoot = new BTreeNode(degree, false);  newRoot.children[0] = root;  SplitChild(newRoot, 0);  int i = 0;  if (newRoot.keys[0] < key)  {  i++;  }  InsertNonFull(newRoot.children[i], key);  root = newRoot;  }  else  {  InsertNonFull(root, key);  }  }  }  private void InsertNonFull(BTreeNode node, int key)  {  int i = node.n - 1;  if (node.leaf)  {  while (i >= 0 && key < node.keys[i])  {  node.keys[i + 1] = node.keys[i];  i--;  }  node.keys[i + 1] = key;  node.n = node.n + 1;  }  else  {  while (i >= 0 && key < node.keys[i])  {  i--;  }  i++;  if (node.children[i].n == 2 \* degree - 1)  {  SplitChild(node, i);  if (key > node.keys[i])  {  i++;  }  }  InsertNonFull(node.children[i], key);  }  } |

## 2.b Explanation of B-Tree insertion method steps

## Bu kod, C# programlama dilinde bir B-Tree (B-ağacı) veri yapısını uygular. İlk olarak, BTreeNode sınıfı, B-Tree düğümlerini temsil eder. Her düğümün, mevcut anahtar sayısı (n), anahtarları içeren bir dizi (keys), alt düğümleri içeren bir dizi (children), ve yaprak düğüm olup olmadığını belirten bir bayrak (leaf) bulunmaktadır.

## BTree sınıfı, B-Tree'nin genel yapısını oluşturur ve kök düğümü ile ağacın derecesini tutar. Derece, her düğümün içerebileceği maksimum alt düğüm sayısını belirtir.

## B-Tree'ye eleman eklemek için Insert metodu kullanılır. Eğer ağaç boşsa, yeni bir kök oluşturulur ve anahtar eklenir. Ağaç doluysa, kökün bölünmesi gerekebilir. Bölünme durumunda, yeni bir kök oluşturulur ve mevcut kök bu yeni kökün ilk alt düğümü olarak atanır.

## InsertNonFull metodu, bir düğüme anahtar ekler. Eğer düğüm yaprak düğüm ise, anahtarı uygun konuma ekler. Eğer yaprak düğüm değilse, uygun alt düğüme gidilir ve gerekirse bölünme işlemi yapılır.

## SplitChild metodu, bir düğümün belirli bir alt düğümünü böler. Yeni bir çocuk düğüm oluşturulur ve ortanca anahtar yukarı taşınır. Anahtar ve çocuk dizileri güncellenir.

## Son olarak, PrintTree metodu, oluşturulan B-Tree'yi seviyeli olarak ekrana yazdırır. Her seviyede bulunan anahtarları ve alt düğümleri gösterir.

## Main metodu, örnek bir B-Tree oluşturur ve belirtilen anahtarları ekler. Ardından, oluşturulan ağacı seviyeli olarak ekrana yazdırır

## 2.c B-Tree insertion test

## B-Tree:

## Level 1, Key: 5

## Level 1, Key: 6

## Level 1, Key: 7

## Level 0, Key: 10

## Level 1, Key: 12

## Level 1, Key: 17

## Level 1, Key: 20

## Level 1, Key: 30

## 3.a Dijkstra’s SP source code and tests

|  |
| --- |
| // source code  *package* proje4.dijsktra;  *import* java.util.\*;  *public class* WeightedNode *implements* Comparable<WeightedNode> {    *public* String name;  *public* ArrayList<WeightedNode> neighbors = *new* ArrayList<WeightedNode>();  *public* HashMap<WeightedNode, Integer> weightMap = *new* HashMap<>();  *public boolean* isVisited = *false*;  *public* WeightedNode parent;  *public int* distance;  *public int* index;  *public* DisjointSet set;     *public* WeightedNode(String name, *int* index) {  *this*.name = name;  distance = Integer.MAX\_VALUE;  *this*.index = index;  }    *public* WeightedNode(String name) {  *this*.name = name;  distance = Integer.MAX\_VALUE;  }   *@Override  public* String toString() {  *return* name;  }   *@Override  public int* compareTo(WeightedNode o) {  *return this*.distance - o.distance;  }   }    *package* proje4.dijsktra;  *import* java.util.\*;  *public class* WeightedGraph {  ArrayList<WeightedNode> nodeList = *new* ArrayList<WeightedNode>();   *public* WeightedGraph(ArrayList<WeightedNode> nodeList) {  *this*.nodeList = nodeList;  }   *void* dijkstra(WeightedNode node) {  PriorityQueue<WeightedNode> queue = *new* PriorityQueue<>();  node.distance = 0;  queue.addAll(nodeList);  *while*(!queue.isEmpty()) {  WeightedNode currentNode = queue.remove();  *for* (WeightedNode neighbor : currentNode.neighbors) {  *if*(queue.contains(neighbor)) {  *if* (neighbor.distance > currentNode.distance + currentNode.weightMap.get(neighbor)) {  neighbor.distance = (currentNode.distance + currentNode.weightMap.get(neighbor));  neighbor.parent = currentNode;  queue.remove(neighbor);  queue.add(neighbor);  }  }  }  }   *for* (WeightedNode nodeToCheck : nodeList) {  System.out.print("Node " +nodeToCheck+", distance: "+nodeToCheck.distance+", Path: ");  pathPrint(nodeToCheck);  System.out.println();  }  }   *public static void* pathPrint(WeightedNode node) {  *if* (node.parent != *null*) {  pathPrint(node.parent);  }  System.out.print(node.name + " ");  }   *public void* addWeightedEdge(*int* i, *int* j, *int* d) {  WeightedNode first = nodeList.get(i);  WeightedNode second = nodeList.get(j);  first.neighbors.add(second);  first.weightMap.put(second,d);  }    *public void* floydWarshall(){    }                                     } |

//Insert your test screenshot or output here

Printing from source A:

Node A, distance: 0, Path: A

Node B, distance: 2, Path: A B

Node C, distance: 5, Path: A C

Node D, distance: 3, Path: A B D

Node E, distance: 5, Path: A B E

Node F, distance: 13, Path: A C F

Node G, distance: 14, Path: A B E G

## 3.b Prim’s MST source code and tests

|  |
| --- |
| // source code  *package* proje4.dijsktra;   *import* java.util.ArrayList; *import* java.util.PriorityQueue;  *public class* Prims {     ArrayList<WeightedNode> nodeList = *new* ArrayList<WeightedNode>();    *public* Prims(ArrayList<WeightedNode> nodeList) {  *this*.nodeList = nodeList;  }    *// Prim's algorithm from source node  void* prims(WeightedNode node) {  *for* (*int* counter = 0; counter < nodeList.size(); counter++) {  nodeList.get(counter).distance = Integer.MAX\_VALUE;  }  node.distance=0;  PriorityQueue<WeightedNode> queue = *new* PriorityQueue<>();  queue.addAll(nodeList);  *while* (!queue.isEmpty()) {  WeightedNode presentNode = queue.remove();  *for* (WeightedNode neighbor : presentNode.neighbors) {  *if* (queue.contains(neighbor)) {    *if* (neighbor.distance > presentNode.weightMap.get(neighbor)) {  neighbor.distance = presentNode.weightMap.get(neighbor);  neighbor.parent = presentNode;  queue.remove(neighbor);  queue.add(neighbor);  }  }  }  }  *int* cost = 0;   *for* (WeightedNode nodeToCheck : nodeList) {  System.out.println("Node " + nodeToCheck + ", key: " + nodeToCheck.distance + ", Parent: " + nodeToCheck.parent);  cost = cost + nodeToCheck.distance;  }   System.out.println("\nTotal cost of MST: " + cost);  }    *public void* addWeightedUndirectedEdge(*int* i, *int* j, *int* d) {  WeightedNode first = nodeList.get(i);  WeightedNode second = nodeList.get(j);  first.neighbors.add(second);  second.neighbors.add(first);  first.weightMap.put(second,d);  second.weightMap.put(first, d);  }  } |

//Insert your test screenshot or output here

Running prims algo on the graph

Node A, key: 15, Parent: E

Node B, key: 5, Parent: A

Node C, key: 6, Parent: D

Node D, key: 8, Parent: B

Node E, key: 0, Parent: null

Total cost of MST: 34

## 3.c BFT or DFT source code and tests

|  |
| --- |
| // source code  *package* graph\_ds.adj.list;  *import* java.util.ArrayList;  *public class* GraphNode {   String name;  *int* index;  *boolean* isVisited = *false*;  GraphNode parentNode;  *public* ArrayList<GraphNode> neighbors = *new* ArrayList<GraphNode>();    *public* GraphNode(String name, *int* index){  *this*.name = name;  *this*.index = index;      }         }  *package* graph\_ds.adj.list;  *import* javax.xml.namespace.QName; *import* java.util.ArrayList; *import* java.util.LinkedList; *import* java.util.Queue; *import* java.util.Stack;  *public class* Graph {   ArrayList<GraphNode> nodeList;   *public* Graph(ArrayList<GraphNode> nodeList){  *this*.nodeList = nodeList;   }    *public void* addUnDirectedEdge(*int* i, *int* j){  GraphNode firstNode = nodeList.get(i);  GraphNode secondNode = nodeList.get(j);  firstNode.neighbors.add(secondNode);  secondNode.neighbors.add(firstNode);  }   *public* String toString() {  StringBuilder s = *new* StringBuilder();  *for* (*int* i = 0; i < nodeList.size(); i++) {  s.append(nodeList.get(i).name + ": ");  *for* (*int* j =0; j < nodeList.get(i).neighbors.size(); j++) {  *if* (j == nodeList.get(i).neighbors.size()-1 ) {  s.append((nodeList.get(i).neighbors.get(j).name) );  } *else* {  s.append((nodeList.get(i).neighbors.get(j).name) + " -> ");  }  }  s.append("\n");  }  *return* s.toString();  }     *public void* dfsVisit(GraphNode node){  Queue<GraphNode> queue = *new* LinkedList<GraphNode>();  queue.add(node);  GraphNode currentNode = queue.remove();  System.out.print(currentNode.name + " ");  *for*(GraphNode graphNode: currentNode.neighbors){  *if*(!graphNode.isVisited){  System.out.print(graphNode.name +" ");  graphNode.isVisited = *true*;  queue.add(graphNode);  *for*(*var* gn: graphNode.neighbors){  *//System.out.println(gn.name);* }  }  }  }   *public void* dfs(){  *for*(*var* gNode: nodeList){  *if*(!gNode.isVisited){  dfsVisit(gNode);  }   }  }    *public void* bfsVisit(GraphNode node){  Stack<GraphNode> stack = *new* Stack<GraphNode>();  stack.push(node);  *while*(!stack.isEmpty()){  GraphNode currentNode = stack.pop();  currentNode.isVisited = *true*;  System.out.print(currentNode.name + " ");  *for*(GraphNode neighbour : node.neighbors){  *if*(!neighbour.isVisited){  stack.push(neighbour);  neighbour.isVisited = *true*;  }  }   }  }  *public void* bfs(){  *for* (GraphNode graphNode: nodeList){  *if*(!graphNode.isVisited){  bfsVisit(graphNode);  }  }  }   *public static void* pathPrint(GraphNode node){  *if*(node.parentNode != *null*){  pathPrint (node.parentNode);  }  System.out.print (node.name + " ");  }  *public void* BFSForSSSPP(GraphNode node){  LinkedList<GraphNode> queue = *new* LinkedList<GraphNode> ();  queue.add (node);  *while*(!queue.isEmpty ()){  GraphNode currentNode = queue.remove (0);  currentNode.isVisited = *true*;  System.out.print ("Printing path for node: " + currentNode.name+": ");  pathPrint (currentNode);  System.out.println ();  *for*(GraphNode neighbour: currentNode.neighbors){  *if*( !neighbour.isVisited ){  queue.add (neighbour);  neighbour.isVisited = *true*;  neighbour.parentNode = currentNode;  }  }  }                              }                             } |

//Insert your test screenshot or output here

For dfs

## A: B -> C -> D

## B: A -> E

## C: A -> D

## D: A -> C -> E

## E: B -> D

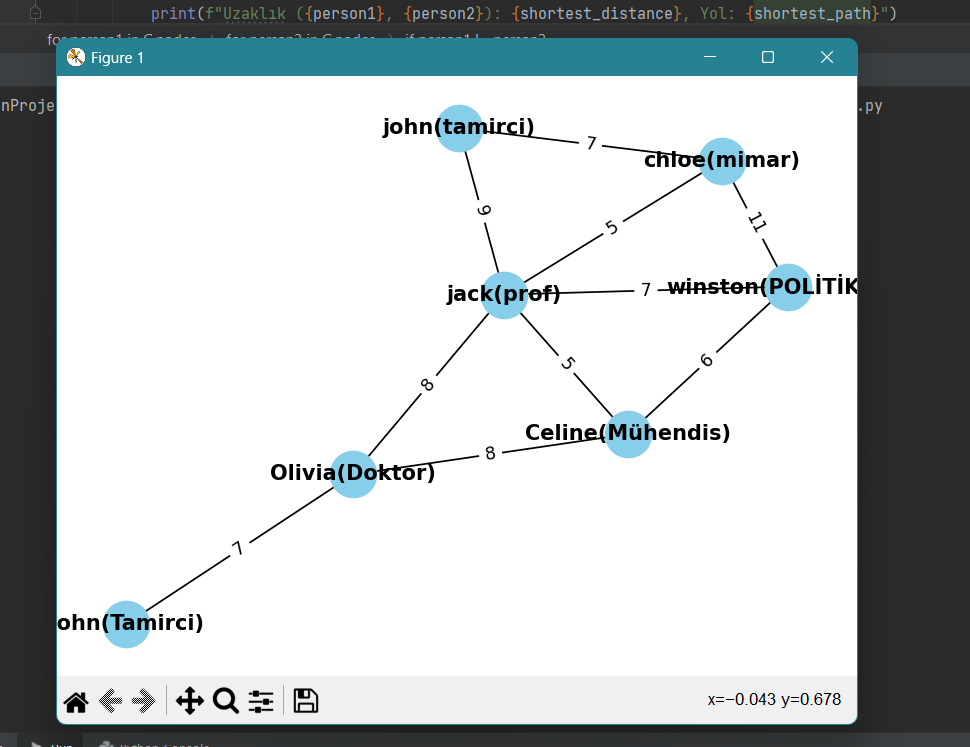
## A B C D E 3.d Filled version of the given Big-O table

//Fill the table and insert here.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Dijkstra’s SP** | Prim’s MST | BFT | **Heap Deletion** |
| **Big-O** (Zaman Karmaşıklığı  Big-O Notasyonuna Göre) | O((V + E) log V) | O(ElogV) | O(|E| + |V|) | O(n log n). |

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

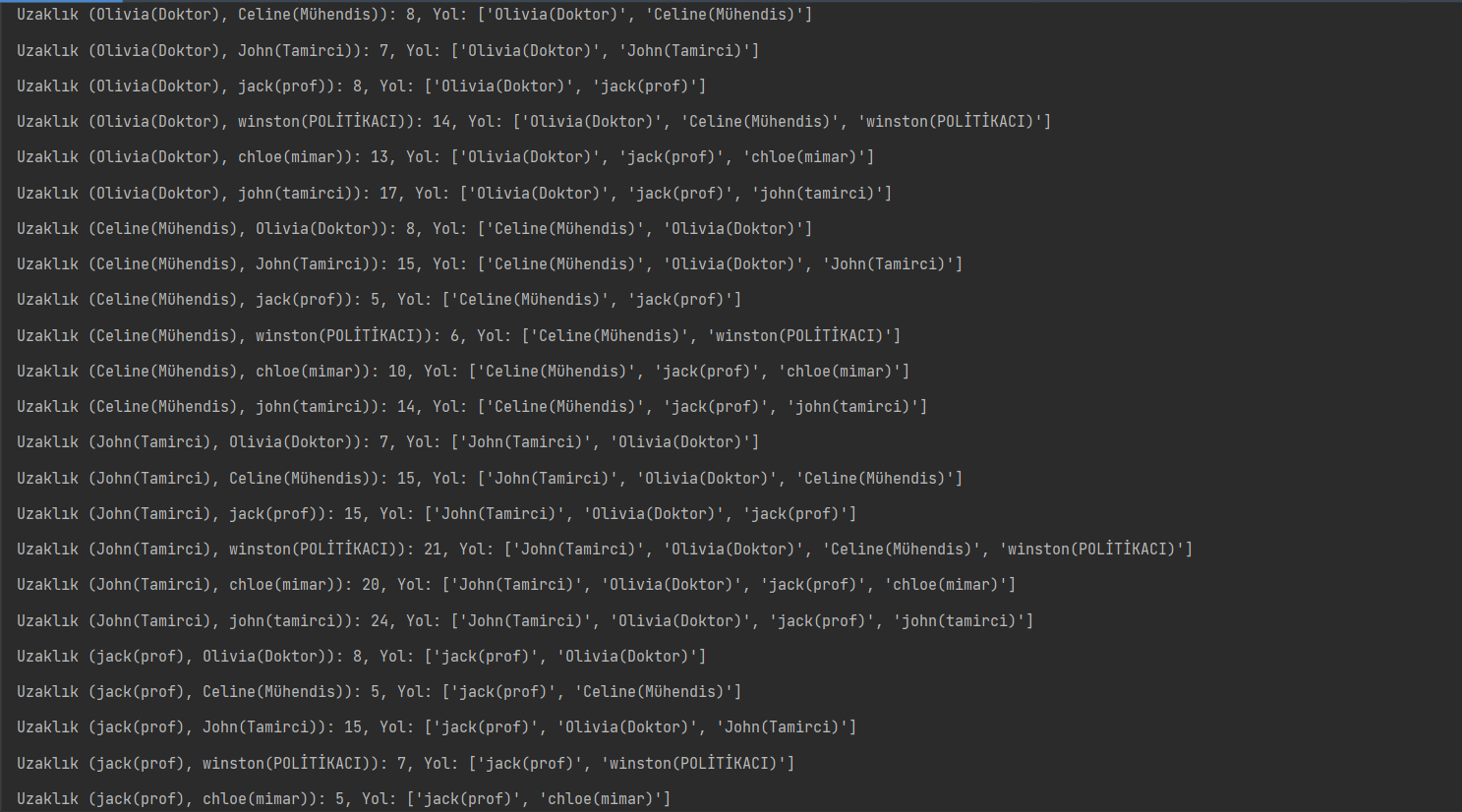
|  |
| --- |
| // source code  pos = nx.spring\_layout(G) nx.draw(G, pos, with\_labels=True, font\_weight='bold', node\_size=700, node\_color='skyblue', font\_color='black') labels = nx.get\_edge\_attributes(G, 'weight') nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels) plt.show() |

//Insert your test screenshot here

## 4.b Computing distances of nodes

|  |
| --- |
| // source code  for person1 in G.nodes:  for person2 in G.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"Uzaklık ({person1}, {person2}): {shortest\_distance}, Yol: {shortest\_path}") |

//Insert your results here



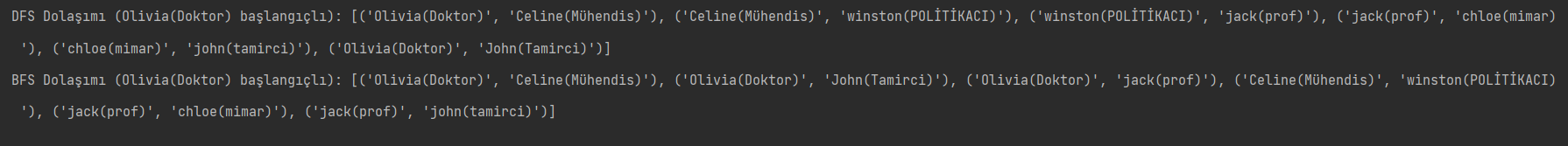
## 

## 

## 4.c Traversing the graphs with DFS and BFS

|  |
| --- |
| // source code  #DFS start\_node\_dfs = "Olivia(Doktor)" dfs\_result = list(nx.dfs\_edges(G, source=start\_node\_dfs)) print(f"DFS Dolaşımı ({start\_node\_dfs} başlangıçlı): {dfs\_result}")  #BFS start\_node\_bfs = "Olivia(Doktor)" bfs\_result = list(nx.bfs\_edges(G, source=start\_node\_bfs)) print(f"BFS Dolaşımı ({start\_node\_bfs} başlangıçlı): {bfs\_result}") |

//Insert your results here



## 4.d Purposes of computing SP, BFS and MST

//Explain how computing SP, BFT and MST for this graph can be useful.

1. **Shortest Path (SP) - En Kısa Yol Bulma:**
   * **Amaçlar:**
     + İki kişi arasındaki en kısa iletişim yolu (en kısa çağrı süresi) bulmak.
     + İki kişi arasındaki en hızlı iletişim yolu üzerindeki kişileri belirlemek.
   * **Uygulama Alanları:**
     + İki kişi arasındaki iletişim süresini en aza indirmek veya optimize etmek.
     + İki kişi arasındaki etkileşimi hızlandırmak.
2. **Breadth-First Search (BFS) - Genişlik Öncelikli Arama:**
   * **Amaçlar:**
     + Verilen bir başlangıç kişisinden diğer kişilere olan mesafeleri ve ilişkileri keşfetmek.
     + Sosyal ağ içindeki genel yapıyı anlamak.
   * **Uygulama Alanları:**
     + Sosyal ağdaki kişiler arasındaki genel bağlantıları anlamak.
     + Belirli bir kişi ile diğer kişiler arasındaki sosyal uzaklıkları belirlemek.
3. **Minimum Spanning Tree (MST) - Minimum Kapsayan Ağaç:**
   * **Amaçlar:**
     + Verilen ağırlıklı bir graf üzerinde minimum ağırlıklı kapsayan ağacı bulmak.
     + Ağırlıklı graf üzerindeki temel ilişkileri ortaya çıkarmak.
   * **Uygulama Alanları:**
     + Verilen ağdaki temel ilişkileri belirlemek ve vurgulamak.
     + İlişkiler arasındaki en önemli veya en düşük maliyetli bağlantıları keşfetmek.

## 4.e Method for accessing to the doctor

|  |
| --- |
| // Algorithm  Verilen bir kişinin en kısa sürede ulaşabileceği doktoru bulmak için en uygun algoritma Dijkstra algoritmasıdır. Dijkstra, bir başlangıç düğümünden diğer tüm düğümlere olan en kısa yolları bulan bir algoritmadır. Bu durumda, başlangıç düğümü verilen kişi, hedef düğüm ise doktor olacaktır.  İşte Dijkstra algoritmasının temel adımları:   1. Başlangıç düğümü için uzaklık değeri sıfır olarak ayarlanır, diğer tüm düğümler için sonsuz (∞) olarak atanır. 2. Başlangıç düğümünden diğer düğümlere olan tahmini en kısa yol uzunlukları güncellenir. 3. Henüz ziyaret edilmemiş düğümler arasında en küçük tahmini en kısa yola sahip düğüm seçilir ve bu düğümü ziyaret edilmiş olarak işaretleriz. 4. Seçilen düğümün komşularının tahmini en kısa yolları güncellenir. 5. Hedef düğüme ulaşılıncaya kadar 3-4 adımları tekrar edilir. |

//Explain the complexity of the method.

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

Ancak, verilen bir kişinin doktora en kısa sürede ulaşması durumunda, telefon aramalarında bir yüzde ile başarı oranını düşünmek gerekebilir. Eğer aranan kişinin belirli bir yüzde ile telefonu açma olasılığı varsa, bu durumu göz önünde bulundurarak bir olasılıklı Dijkstra algoritması uygulanabilir.

Bu durumu ele almak için, her bir kenarın ağırlığına arama başarısını ifade eden bir olasılık ekleyebilirsiniz. Örneğin, her arama başarısız olduğunda bir miktar zamanın geçtiği, ancak belirli bir olasılıkla da başarıya ulaşıldığı bir model kullanılabilir. Bu durumda, her bir kenarın ağırlığı, belirli bir başarı olasılığı ile çarpılmalıdır.

Örneğin, bir kenarın ağırlığı w ve başarı olasılığı p ise, yeni ağırlık w'yi şu şekilde hesaplayabiliriz: w' = w / p.

Dijkstra algoritmasının karmaşıklığı, genellikle önceden belirlenen bir düğüm kümesi üzerinde O(V^2) veya önceden belirlenmemiş bir küme üzerinde O((V + E) \* log(V)) olarak ifade edilir. Burada V düğüm sayısını, E kenar sayısını temsil eder.

Belirli bir yüzde ile telefonu açma durumunda algoritma ve karmaşıklığını değiştirmek için, her bir kenarın ağırlığını belirli bir başarı olasılığı ile çarpmak gerekecektir. Bu durumda, olasılıklı Dijkstra algoritması, her kenarın ağırlığını bu olasılıkla çarparak en kısa yolları hesaplayacakt

## 5.a. Brief comparison of Prim’s and Kruskal’s algorithms

## Prim ve Kruskal algoritmaları, minimum gergeske (Minimum Spanning Tree - MST) bulma problemine çözüm getiren iki farklı algoritmadır. Prim Algoritması, başlangıç düğümünden başlayarak her adımda mevcut ağacı genişletir ve en küçük ağırlıklı kenarı ekler. Bu yöntem, genellikle bağlı grafiklerde daha etkili çalışır ve öncelikle Fibonacci heap veya minimum öncelik kuyruğu gibi veri yapıları kullanır. Öte yandan, Kruskal Algoritması, kenarları ağırlıklarına göre sıralar ve en küçük ağırlıklı kenarı ekler. Bu yöntem, genellikle daha fazla paralelleştirme olanağı sunar ve bağlı olmayan grafiklerde veya ağırlıkların birbirine yakın olduğu durumlarda etkili olabilir. Prim daha hızlı çalışabilir, özellikle bağlı bir grafikte, ancak Kruskal daha genel uygulama alanlarına sahip olabilir. Seçim, uygulama gereksinimlerine, grafik yapısına ve performans kriterlerine bağlı olarak yapılmalıdır.

## 5.b Implementation of Trie Data Structure and Insert Method

|  |
| --- |
| using System;  using System.Collections.Generic;  public class TrieNode  {  public char Value { get; set; }  public Dictionary<char, TrieNode> Children { get; set; }  public TrieNode(char value)  {  Value = value;  Children = new Dictionary<char, TrieNode>();  }  }  public class Trie  {  private TrieNode root;  public Trie()  {  root = new TrieNode('\0');  }  public void Insert(string word)  {  TrieNode currentNode = root;  foreach (char c in word)  {  if (!currentNode.Children.ContainsKey(c))  {  currentNode.Children[c] = new TrieNode(c);  }  currentNode = currentNode.Children[c];  }  }  public static void Main(string[] args)  {  Trie trie = new Trie();  string[] words = { "ana", "ananas", "merhaba", "merhabalar" };  foreach (string word in words)  {  trie.Insert(word);  }  Console.WriteLine("Kelimeler Trie'ye eklendi.");  }  } |

Output:

Kelimeler Trie'ye eklendi.

## Bu C# kodu, Trie veri yapısını oluşturan iki ana sınıf içerir: TrieNode ve Trie. TrieNode, Trie ağacındaki her bir düğümü temsil eder ve bir karakter değeriyle birlikte çocuk düğümleri tutan bir sözlük içerir. Trie sınıfı, Trie veri yapısını yönetir ve Insert metodu aracılığıyla verilen kelimeleri Trie ağacına ekler. Kodun Main metodu, belirli kelimeleri Trie'ye ekleyerek temel işleyişi gösterir. Örneğin, "ana", "ananas", "merhaba" ve "merhabalar" kelimeleri Trie'ye eklenmiştir. Bu örnek, Trie veri yapısının temel ekleme işlemlerini basit bir şekilde gösterir.

## 5.c. Explanation of 4 Terms

1. B+ Tree:

B+ Tree, özellikle disk tabanlı veritabanlarında ve dosya sistemlerinde kullanılan bir denge ağaç veri yapısıdır. Bu ağaç türü, sıralı veri erişimini optimize etmeye yönelik birçok avantaj sunar. B+ Tree, yaprak düğümlerinde gerçek veriyi barındırırken, iç düğümler sıralama anahtarlarını içerir. Bu özellik, sıralı veri erişimini kolaylaştırır ve disk tabanlı sistemlerde etkili bir şekilde kullanılır.

1. 2-3-4 Tree:

2-3-4 Tree, bir denge ağaç yapısıdır ve her iç düğümde 2, 3 veya 4 çocuk bulunur. Bu ağaç türü, veriyi dengelemek ve hızlı arama işlemleri sağlamak amacıyla tasarlanmıştır. 2-3-4 Tree'de veri, yaprak düğümlerde saklanır ve her seviyede düğüm sayısı sabittir. Bu, ağacın dengeli kalmasını sağlar.

1. Quadratic Probing:

Quadratic Probing, özellikle hash tablolarında kullanılan bir çakışma çözme yöntemidir. Bir anahtarın hedeflenen konumunda bir çakışma olması durumunda, bu yöntem çakışmanın olmadığı bir sonraki boş konumu arar. Quadratic Probing, lineer probing'e benzer ancak boş konumu ararken daha karmaşık bir formül kullanır. Bu, çakışma çözme stratejilerinden biridir ve hash tablolarının performansını etkiler.

1. Warshall’s Algorithm:

Warshall Algoritması, graf teorisinde kullanılan bir algoritmadır. İki nokta arasındaki doğrudan yolları bulmanın ötesinde, bu algoritma graf üzerindeki tüm noktalar arasındaki erişilebilirlik durumunu belirlemek için kullanılır. Algoritma, bir graf matrisini kullanır ve bu matris üzerindeki transitif kapanımı hesaplar. Warshall Algoritması, ilişkisel grafiklerde, yol bulma ve ulaşılabilirlik problemlerinde yaygın olarak kullanılır.

## Self-assessment Table

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Points** | **Estimated Grade** | **Explanation** |
| **1 a) AVL Tree** | **10** | 10 | İstenildiği gibi yapıldı |
| **1 b) Heap** | **10** | 10 | İstenildiği gibi yapıldı |
| **2) B-Tree Insertion / AVL Tree Insertion / Red-Black Trees / Huffman Encoding Tree** | **10** | 10 | İstenildiği gibi yapıldı |
| **3 a) Dijkstra’s shortest path code + test** | **4** | 4 | İstenildiği gibi yapıldı |
| **3 b) Prim’s MST code + test** | **4** | 4 | İstenildiği gibi yapıldı |
| **3 c) BFT or DFT code + test** | **3** | 3 | İstenildiği gibi yapıldı |
| **3 d) Filling Big-O Table** | **4** | 4 | İstenildiği gibi yapıldı |
| **4 a) Graph Drawing** | **3** | 3 | İstenildiği gibi yapıldı |
| **4 b) Finding Shortest Paths with Dijkstra’s** | **3** | 3 | İstenildiği gibi yapıldı |
| **4 c) DFS and BFS** | **3** | 3 | İstenildiği gibi yapıldı |
| **4 d) Thinking and Writing aims of given algorithms** | **3** | 3 | İstenildiği gibi yapıldı |
| **4 e) Real Life Application** | **3** | 3 | İstenildiği gibi yapıldı |
| **5 i) Comparison (Prim’s & Kruskal’s Algorithm)** | **5** | 5 | İstenildiği gibi yapıldı |
| **5 ii) Trie Data Structure and Insertion Method** | **5** | 5 | İstenildiği gibi yapıldı |
| **5 iii) Explanations of 4 terms** | **10** | 10 | İstenildiği gibi yapıldı |
| **Demo Video for Source Codes and Tests of Q2 Q3a, Q3b, Q3c and Q5.ii .** | **10** | 10 | İstenildiği gibi yapıldı |
| **Self-assessment Table** | **10** | 10 | İstenildiği gibi yapıldı |
| **Total** | **100** | 100 | İstenildiği gibi yapıldı |