Algoritmer og Datastrukturer E2024

Simone Ingvild Lebech

Afleveringsdato: 29. November 2024

Underviser: Ole Dolriis

Contents

1 Øvelse 1

Design en algoritme der tager en string som input og returnerer det oftest forekommende ord. Antag der altid er en blank imellem to ord og den sidste karakter altid er et punktum.

1.1 Kode for Hyppigst forekommende ord

```
import java.util.HashMap;
import java.util.Map;

public class MostFrequentWord {
    public static String findMostFrequentWord(String text) {
        if (text == null || text.isEmpty()) {
            return "";
        }

        // Opret HashMap til at tælle forekomster af hvert ord
        Map<String, Integer> wordCount = new HashMap<>();

        // Split teksten på mellemrum, kommaer og punktummer og konverter til lowercase
        String[] words = text.toLowerCase().split("[\\s,.!]+");

        // Tæl forekomster af hvert ord
        for (String word : words) {
            if (!word.isEmpty()) {
                wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
            }
        }

        // Find ordet der forekommer mest
        String mostFrequentWord = "";
        int maxCount = 0;
        for (Map.Entry<String, Integer> entry : wordCount.entrySet()) {
            if (entry.getValue() > maxCount) {
                maxCount = entry.getValue();
                mostFrequentWord = entry.getKey();
            }
        }
```

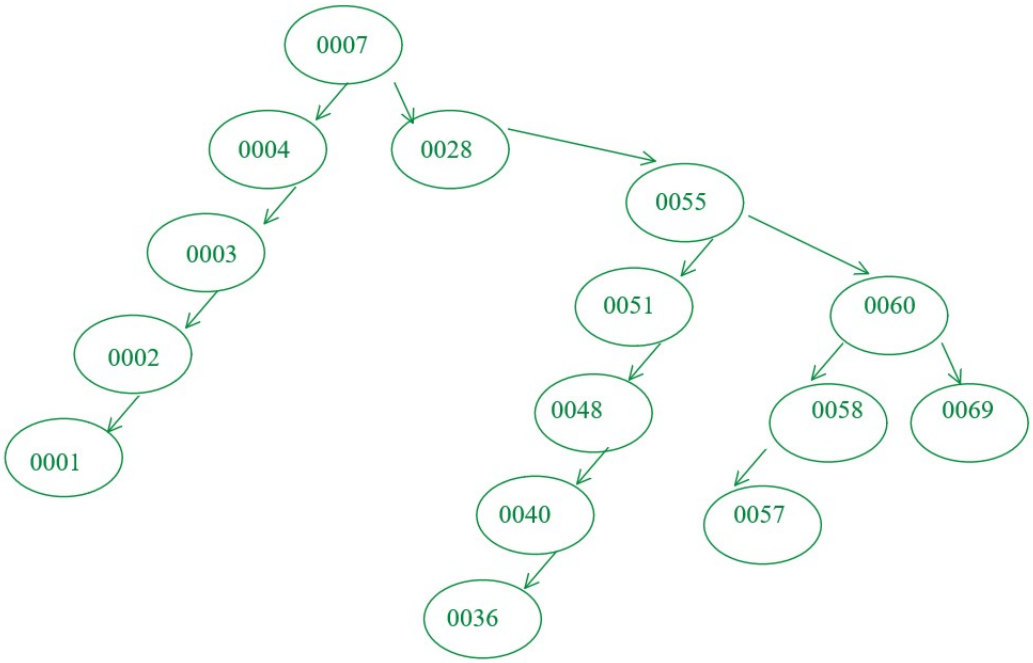
```
        return mostFrequentWord;
    }

    public static void main(String[] args) {
        String text = "I have a dog, a cat and four chickens";
        String mostFrequentWord = findMostFrequentWord(text);
        System.out.println("The most frequent word is: " + mostFrequentWord);
    }
}
```

1.1.1 Kodeforklaring

Koden definerer en klasse kaldet MostFrequentWord som indeholder en statisk metode findMostFrequentWord. Denne metode tager en streng som input og returnerer det ord som optræder flest gange, i den givne streng. Metoden findMostFrequentWord tjekker om input strengen er nul eller tom, og hvis det er tilfældet returneres en tom streng. Derefter laves et HashMap som gemmer ord og deres respektive antal. Input strengen deles så i ord, men ignorerer tegnsætning og omdanner alle ord til små bogstaver. Koden itererer over hvert ord, og øger antallet i HashMap. Efter den er gået gennem alle ord, itererer den gennem HashMap for at finde det ord som har et højeste antal forekomster. Til sidst returnerer metoden det ord som optræder flest gange. I main metoden, gives en eksempel streng, og findMostFrequentWord metoden kaldes for at bestemme det hyppigst optrædende ord i strengen. Resultatet printes i terminalen.

2 Øvelse 2

Skriv en metode til implementering af træet som kan returnere antallet af grene i træet. Antag at roden ikke kan være en del af en gren. Nedenstående figur forestiller et binært søgetræ



2.1 Kode til at konstruere det givne træ

```
public class TreeBuilder {
    public static void main(String[] args) {
        // Lav knuderne
        BinaryTreeNode root = new BinaryTreeNode(7);

        // Byg venstre undertræ
        root.left = new BinaryTreeNode(4);
        root.left.left = new BinaryTreeNode(3);
        root.left.left.left = new BinaryTreeNode(2);
        root.left.left.left.left = new BinaryTreeNode(1);

        // Byg højre undertræ
        root.right = new BinaryTreeNode(28);
        root.right.right = new BinaryTreeNode(55);
        root.right.right.left = new BinaryTreeNode(51);
        root.right.right.left.left = new BinaryTreeNode(48);
```

```
        root.right.right.left.left.left = new BinaryTreeNode(40);
        root.right.right.left.left.left.left = new BinaryTreeNode(36);
        root.right.right.right = new BinaryTreeNode(60);
        root.right.right.right.left = new BinaryTreeNode(58);
        root.right.right.right.left.left = new BinaryTreeNode(57);
        root.right.right.right.right = new BinaryTreeNode(69);

        // Test countBranches methoden
        int branchCount = root.countBranches();
        System.out.println("Number of branches: " + branchCount); // Expected output: 2
    }
}
```

2.1.1 Kodeforklaring

Koden bruges til at konstruere et binært træ med de værdier givet i opgaven. Metoden countBranches kaldes på rodknuden for at beregne antallet af grene i træet. En gren, i denne kontekst, er vejen fra roden til et blad. Det beregnede antal af grene printes så i terminalen.

2.2 Kode til implementering af træ

```
public class BinaryTreeNode {
    int value;
    BinaryTreeNode left;
    BinaryTreeNode right;

    // Constructor til at initialize knude med værdi
    public BinaryTreeNode(int value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }

    // Metode til at tælle grene
    public int countBranches() {
        return countBranches(this, true);
    }

    private int countBranches(BinaryTreeNode node, boolean isRoot) {
```

```
        if (node == null) {
            return 0;
        }

        int count = 0;
        if (!isRoot) {
            BinaryTreeNode onlyChild = getOnlyChild(node);
            if (onlyChild != null) {
                BinaryTreeNode grandchild = getOnlyChild(onlyChild);
                if (grandchild != null && grandchild.isLeaf()) {
                    count = 1;
                }
            }
        }

        return count + countBranches(node.left, false) + countBranches(node.right, false);
    }

    // Hjælpermetode til kun at få barnet af knuden
    private BinaryTreeNode getOnlyChild(BinaryTreeNode node) {
        if (node == null) {
            return null;
        }
        if (node.left != null && node.right == null) {
            return node.left;
        }
        if (node.right != null && node.left == null) {
            return node.right;
        }
        return null;
    }

    // Metode til at tjekke om knuden er et blad
    public boolean isLeaf() {
        return left == null && right == null;
    }
}
```

2.2.1 Kodeforklaring

Koden definerer en BinaryTreeNode klasse som repræsenterer en knude i et binært træ. Et binært træ er en data struktur hvor hver knude har max 2 børn: et højre og et venstre. Klassen indeholder også forskellige metoder. Konstruktøren initialiserer et nyt binært træ objekt, og tager en heltalsværdi som input og tildeler det til value feltet. Left og right referencerne sættes initielt til null. Metoden countBranches tæller rekursivt antallet af grene i undertræet rodfæstet i den nuværende knude. En gren defineres som vejen fra rod til et blad, og metoden tjekker om knuden er et blad. Hvis det er et blad returneres 0, og ellers kaldes metoden rekursivt på venstre og højre barn og der tilføjes 1 for hver ikke-null barn. Metoden getOnlyChild er en hjælpermtode som returnerer det eneste barn for en knude. Hvis knuden har to børn, eller ingen, returneres null. Metoden isLeaf tjekker om knuden er et blad, altså at den ingen børn har. Det forudsætter at både venstre og højre børneknude er null.

2.3 Supplerende opgave 1: Karakteriser træet

Træet er binært, men ikke fuldt da ikke alle knuder har to børn. Da alle niveauer ikke er helt udfyldt er det heller ikke perfekt. Kigger man på det numerisk, ses at der er 15 knuder, 6 blade (knuder uden børn) og 9 indre knuder (knuder med mindst et barn). Træet er 6 højt, da dette er den længste sti fra rod til blad. Den optimale højde for et binært træ med n knuder er $\lceil \log_2(n+1) \rceil$ For 15 knuder som her, er den optimale højde da: $\lceil \log_2(16) \rceil = 4$

2.4 Supplerende opgave 2: Transformer søgetræ

Et binært søgetræ er en datastruktur hvor hver knudes venstre barn har en mindre værdi end knuden selv, og hvert højre barn har en større værdi. En prioritetskø er en abstrakt datastruktur, der opretholder en samling ag elementer, som hver har en prioritet tilknyttet. Operationerne på en prioriteteskø er oftest at tilføje et element og at finde elementet med højest prioritet. For at omdanne fra den ene til den anden vil man oftest udføre en in-order traversal af det binære søgetræ og indsætte elementerne i prioriteteskøen.

- Initialiser en tom prioriteteskø: Vælg en passende implementering (min-heap eller max-heap alt efter om vi ønsker stigende eller faldende prioritet)
- Udfør in-order traversal for hver knude i træet. Da in-order traversal besøger knuderne i stigende rækkefølge i et binært søgetræ vil elementerne indsættes i køen i den ønskede rækkefølge.

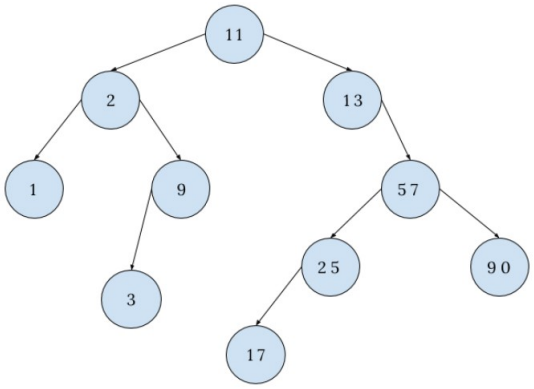
Tidskompleksiteten vil nu se ud som følgende

- In-order traversal: O(n), hvor n er antal knuder i træet
- Indsættelse i prioritetskø: Afhænger af implementering, men for min-heap vil man typisk have $O(\log n)$

Samlet fås altså: $O(n \log n)$ da der udføres indsættelse i prioriteteskøen.

3 Øvelse 3

Nedenstående figur forestiller et binært søgetræ som ikke er et AVL-træ. List rækkefølgen knuderne vil blive besøgt i en in-order og i en level-order traversering.



Ved in-order gives en sorteret liste af knuder, derfor: 1,2,3,9,11,13,17,25,57,90.

Ved level-order besøges knuderne niveau for niveau, fra venstre mod højre og derfor: 11,2,13,1,9,57, 3,25,90,17.

3.1 Internal path

Her tales om summen af afstanden fra roden til hver knude. For dette træ er den relativt høj. Man går gennem hver knude og tæller antal trin tilbage til rode. Dermed fås: $1 \rightarrow 11 : 2, 2 \rightarrow 11 : 1, 9 \rightarrow 11 : 2, 3 \rightarrow 11 : 3, 13 \rightarrow 11 : 1, 57 \rightarrow 11 : 2, 25 \rightarrow 11 : 3, 17 \rightarrow 11 : 4, 90 \rightarrow 11 : 3$. Lægges disse sammen fås 21.

3.2 Omarrangering

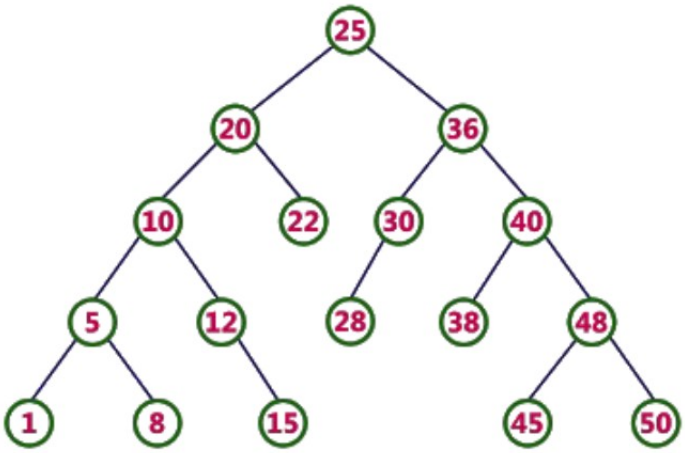
Ubalancen i træet er ved knude 13, hvor højden af det højre subtræ er 3, og det venstre subtræ er 0. Da der er forskel i højden mellem de to subtræer, bryder vi AVL-træets balancekrav, fordi forskellen i højden for enhver knude højst må være 1. For at omarrangere træet kan man rotere det højre subtræ, en højre rotation ved knude 13 flytter 57 op og 13 ned, så træet balanceres.

3.3 AVL-træ før seneste operation?

AVL-træer er selvbalancerende binære søgetræer, dvs. at når en knude indsættes eller slettes, udføres rotationer for at opretholde balancen. Hvis et træ så ikke er AVL, betyder det, at der på et tidspunkt er en operation der ikke blev fulgt af en balancering. Derfor kunne træet altså godt have været et AVL-træ før den seneste operation, forudsat at vi i stedet havde indsat knude 3 eller slettet knude 12.

4 Øvelse 4

Nedenstående figur forestiller et binært søgetræ. List rækkefølgen knuderne besøges i en post-order og i en pre-order traversering.



I en post-order traversing besøges venstre undertræ, højre undertræ, og derefter roden. Altså bliver rækkefølgen:1,8,5,12,15,10,22,20,28,30,38,45,48,40,36,25. Hvis man derimod bruger pre-order besøger man rod, venstre undertræ og derefter højre undertræ. Dermed bliver rækkenfølgen så: 25,20,10,5,1,8,12,15,22,36,30,28,40,38,45,48,50

4.1 Internal Path Length

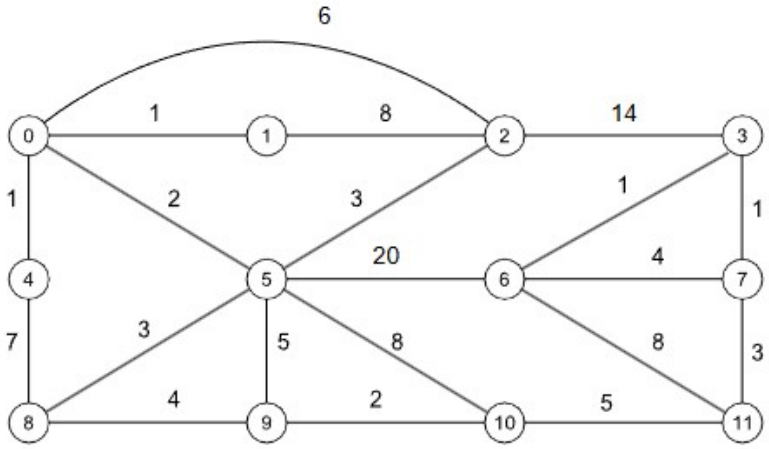
Summen af afstanden fra roden til hver knude. Her tælles antallet af trin fra roden til hver knude og tallene summeres. For venstre undertræ: $1 \rightarrow 25:4, 8 \rightarrow 25:4, 5 \rightarrow 25:3, 15 \rightarrow 25:4, 12 \rightarrow 25:3, 10 \rightarrow 25:2, 20 \rightarrow 25:1, 22 \rightarrow 25:2$ For højre undertræ: $45 \rightarrow 25:4, 50 \rightarrow 25:4, 48 \rightarrow 25:3, 40 \rightarrow 25:2, 38 \rightarrow 25:3, 36 \rightarrow 25:1, 28 \rightarrow 25:3, 30 \rightarrow 25:2$
Samlet fås altså; $23 + 22 = 45$

4.2 Er det et AVL træ?

Et AVL-træ forudsætter at forskellen i højde mellem venstre og højre undertræ højest er 1. For at afgøre om dette er tilfældet undersøges højderne af undertræerne derfor. Hvis forskellen er større end 1 for nogen af knuderne, som her, er det altså ikke et AVL-træ.

5 Øvelse 5

Figuren viser en graf. Find et minimum spanning tree for grafen. Svaret skal være en liste af edges/kanter, der viser i hvilken rækkefølge algoritmen vil etablere forbindelse mellem knuderne. Angiv også træets totale væg samt hvilken algoritme der er brugt.



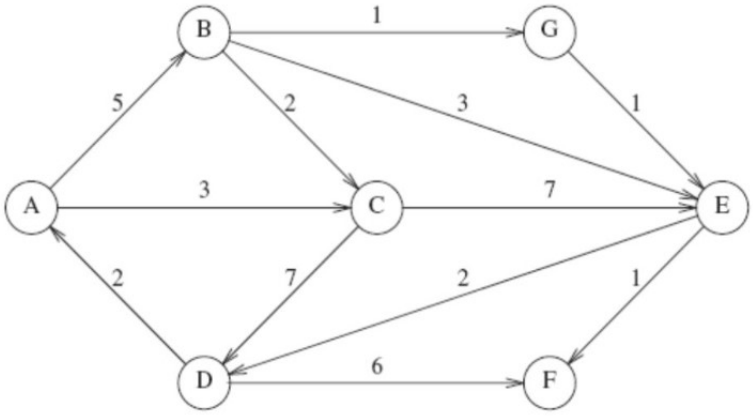
Det ses at der er tale om en vægtet, urettet graf og at der skal findes et MST. MST er en delgraf, der forbinder alle knuder med den mindst mulige samlede vægt. Hertil kan bruges Prim's algoritme hvor man starter fra en vilkårlig knude og udvider træet ved at tilføje den kant med laveste vægt, der forbinder en knude i træet til en knude udenfor.

1. Start: MST = 0
2. Mindste kant: Kanten (0,4) med vægt 1 $\rightarrow$ MST = $0, 4$
3. Mindste kant: Kanten (4,8) med vægt 3 $\rightarrow$ MST = $0, 4, 8$
4. Mindste kant: Kanten (8,9) med vægt 4 $\rightarrow$ MST = $0, 4, 8, 9$
5. Mindste kant: Kanten () med vægt $\rightarrow$ MST = 0,4,8,9

Kanter: (0,4), (4,8), (8,9), (9,5), (5,6), (6,10), (10,11), (6,3), (3,2), (2,1)

6 Øvelse 6

Figuren viser en graf. Tabellen i opgaven viser startkonfigurationen for en traversering af den givne graf med anvendelse af Dijkstras algoritme. Vis slutkonfigurationen for en traversering startende i knude/vertex A.



Vi starter i A og kigger mod B, hvor vi ser distancen er 5 som nu tilføjes til prioriteteskøen. For hver naboknude beregner man distancen til startknuden, forudsat at stien går gennem den nuværende kant. Derfor kigges fra A til C som har distancen 3. Da der er kortere afstand fra A til C end fra A til B, kigger vi nu fra C knuden og naboerne herfra. Vi ser at C til D har afstanden 7, og at C til E også har afstanden 7. Da der ikke er flere naboer til C trækkes nu den korteste distance ud fra prioriteteskøen, her B. B til C har distancen 2, mens B til E har distancen 3. Vi allerede har besøgt E, men ser at afstanden fra B til E er kortere end fra C til E, og derfor lægges nu en ny værdi for E i prioriteteskøen. Herefter kigges fra B til G, hvor distancen er 1. Vi starter da i G, som har distancen 1 til E. Igen ændres værdien for E. Nu startes i E, og der ses fra E til F som har distancen 1, og fra E til D som har distancen 2. D har tidligere været besøgt, men E til D er kortere end C til D, og derfor ændres værdien fra 7 til 2. Nu startes i F som havde den korteste distance fra E, og der ses at den ikke har nogen udgående kanter. Altså trækkes punktet fra prioriteteskøen med den korteste distance, her D. Fra D til A ses at distancen er 2, mens den fra D til F er 6.

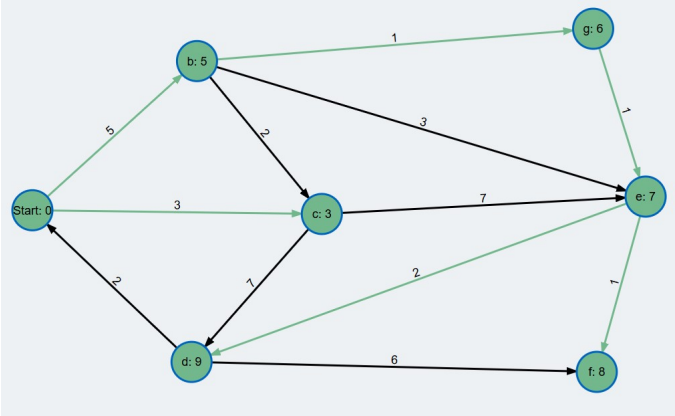


Figure 1: Grafisk fremstilling

v	known	d_v	p_v
A	true	0	0
B	true	5	A
C	true	3	A
D	true	9	E
E	true	7	G
F	true	8	E
G	true	6	B

Table 1: Skema over sti