Uppgift 9 för betyg C

Simon Moradbakti

HT 2024

Introduktion

I den parallela uppgiften som går denna vecka som fungerar enligt depth first principen, alltså att man går så djupt ner i trädets grenar, och kramar vänster sida fram tills dess att antalet noder tar slut. Därefter flyttar man ett steg åt höger och gör samma sak igen. Den här rapporten ska gå igenom hur det skulle bli om man istället gör en så kallad breath-first metod som är ett alternativ. Som det låter går vi istället längst med nivåer av träder och analyserar varje nivå. Lite som en hiss i ett våningshus. För detta är en bra idé att använda sig av en kö. Det kommer förses bilder och ytterliggare förklaringar av detta samt kod, för att detta ska framgå tydligare.

one level at a time

Om man följer instruktionerna som ges för den här deluppgiften så får man följande bild:

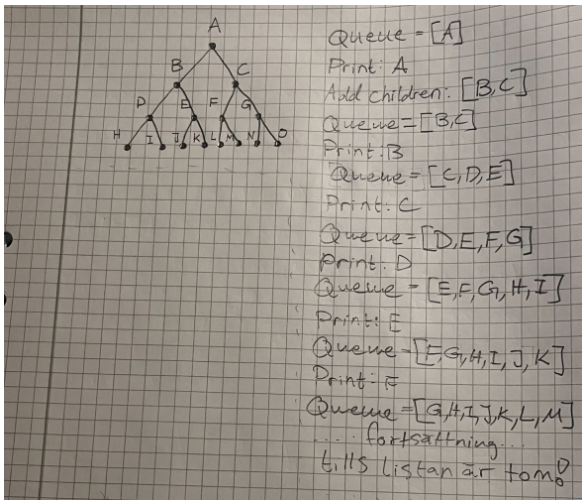

Figure 1: Handskriven bild

I bilden syns ett träd med noderna A-O. Hade depth-first körts på detta träd hade vårat utskrivna värde blivit: A, B, D, H, I, J, E, K, L, C, F, M, N, G, O. Dvs inte det värdet vi vill ha som är A, B, C, D, E, F, G, H, I, J, K, L, M, N, O. Till höger i bilden visas detta hur det rent konkret sker. Vi skapar en lista med det element som är vår rot (i detta fall A) och printar det. Sedan lägger vi till våra "children" eller noder som kommer ut från vår rot. Detta blir en ny kö som sedan fortsätter efter att roten har avlägsnats. Detta fortsätter tills hela trädets löv har blivit utskrivna. Detta är den bästa typen av lösning för detta typ av träd, men behöver inte nödvändigtvis vara det för alla typer. Hade trädet sett annourlunda ut, till exempel att löven var ordnade på ett sätt att depth first principen är bättre så är det att föredra. Vi kan också dra slutsatsen att tidskomplexiteten för detta typ av träd blir O(n), längden på trädet avgör hur lång tid det tar!

the queue

Denna deluppgift gav ingen kod att utgå från, men den gav ett tips. Dvs att man kan använda kod från en tidigare uppgift som hjälpmedel. Detta var den uppgift som går parallelt med denna där man med hjälp av stacken skapade en kö för att göra depth first. Efter revidering av denna kod blev delar av den nya koden detta:

```
public static void main(String[] args)
        {
        Node root = new Node("A");
        root.left = new Node("B");
        root.right = new Node("C");
        root.left.left = new Node("D");
        root.left.right = new Node("E");
        root.right.left = new Node("F");
        root.right.right = new Node("G");
        root.left.left.left = new Node("H");
        root.left.left.right = new Node("I");
        root.left.right.left = new Node("J");
        root.left.right.right = new Node("K");
        root.right.left.left = new Node("L");
        root.right.left.right = new Node("M");
        root.right.right.left = new Node("N");
        root.right.right.right = new Node("O");
        }
```

här så skapas självaste trädet med noder likt bilden ovan. Notera att vi använder oss av "Node" då detta kan hantera strängar (som är ord eller bokstäver) och inte uteslutande int värden som en kö gör. Vi kan även se att vi använder oss av just String[] i första raden. Delar av Breath first Traversal metoden implementerades såhär:

```
public String next() {
    if (queue.isEmpty()) {
        return null;
    }
    Node current = queue.poll();
    if (current.left != null) {
        queue.add(current.left);
    }
    if (current.right != null) {
        queue.add(current.right);
    }
    return current.value;
}
```

Kortfattat så tar queue.poll() bort den första noden ur kön. Medans queue.add(current.left) och queue.add(current.right) har som uppgift att lagra eventuella noder längst bak i vår kö. Anledningen till detta är att det mer eller mindre ganaterar att vi först går igenom/traverserar noder på samma nivå som den tidigare noden först.

a lazy sequence

I den sista deluppgiften fick man som uppdrag att implementera en sekvens som returnerar ett värde, ett efter ett. Detta kommer att konstruera ett träd och vi ska därefter svara på en del frågor. Delar av de komplementerande kodraderna för sekvensen Sequence blev följande:

```
class Sequence {
    private Queue<Node> nodeQueue;
    public Sequence(Node root) {
        nodeQueue = new LinkedList<>();
        if (root != null) {
            nodeQueue.add(root);
        }
    }
    public boolean hasNext() {
        return !nodeQueue.isEmpty();
    }
    public String next() {
        if (nodeQueue.isEmpty()) {
            return null;
        }
    public Sequence sequence() {
    return new Sequence(root);
}
```

När den fullständiga koden körs så får vi ett träd, det extraherar de 3 första värderna tar en paus och fortsätter sedan med 2 till. När det kommer till vad som händer när man lägger till värden under pausen, alltså under tiden som koden exekveras så kan det uppfattas som ganska självklart. Ingenting kommer ske under själva ekeveringen då ekveringen skedde **innan** förändringen skedde och det är denna version (och ingen annan) som kommer exekveras ut. Samma sak sker dock inte om man istället skulle välja att ta bort värden under exekveringen. Detta då en referens till en borttagen nod kan finnas i kön vilket mer eller mindre skulle förstöra hela kön.

Sammanfattning

Denna rapport har behandlat ett alternativt sätt att traversera ett träd med den så kallade breath first principen som går efter nivåer i ett träd istället för att krama vänstra grenen och flytta till höger enligt depth first principen. Rapporten diskuterar när de olika versionerna kan vara som bäst och vad som skiljer de åt kodmessigt. Den har även berört vad som händer ifall man ändrar koden mitt i en exekvering under en pausning och vad det kan få för konsekvenser.