Assignment 6 - Binary Tree

Jesper Hesselgren

Oktober 10, 2024

Introduktion

I denna rapport ska vi undersöka den mer komplexa datastrukturen träd, med särskilt fokus på binära träd. Ett binärt träd är en typ av trädstruktur där varje nod har högst två grenar: en vänster och en höger gren. Vi kommer att utforska hur man implementerar ett binärt träd, lägger till en nod och söker efter specifika noder i trädet, vi ska sedan jämföra våran sökmetod `lockup` metod med binary search. Vidare ska vi undersöka hur man kan traverser och skriva ut hela trädet i *in-order* med hjälp av våran egen implementerade stack från assignment 3.

Implementation av binärt träd

Vårt binära träd implementeras genom huvudklassen `BinaryTree` och den inre klassen `Node`, som representerar noderna i trädet. `BinaryTree` innehåller attributet `root`, vilket är en referens till trädets första nod. `Node`-klassen innehåller tre attribut: `value`, `left` och `right`. `Value` lagrar nodens data, som i detta fall är av typen `Integer`, medan `left` och `right` är referenser till nodens vänstra respektive högra barn. Om en nod saknar en vänster- eller högerreferens, är den ett löv i trädet, och dess `left`- eller `right`-referens är satt till `null`.

Det binära trädet vi implementerar kommer att vara ett sorterat binärt träd som inte tillåter dubletter. Alla noder med värden som är mindre än rotens värde placeras till vänster, och alla noder med större värden placeras till höger. Därmed kommer den minsta noden att finnas längst till vänster i trädet och den största längst till höger.

Add metoden

Den rekursiva metoden, arbetar genom att göra upprepade funktionsanrop som traverserar trädet nedåt. Det första metoden börjar med är att kontrollera om värdet redan finns. Om så är fallet, returnerar metoden utan att

göra något, eftersom dubbletter inte är tillåtna. Om värdet är mindre än nodens värde, kallas add rekursivt på den vänstra subnoden, eller om vänster nod saknas, lägger vi till en ny nod där. På liknande sätt, om värdet är större, går vi till höger subnod eller skapar en ny nod om ingen höger nod finns. Rekursionen fortsätter nedåt i trädet tills den når en position där den nya noden kan placeras.

Den iterativa metoden, implementerar samma logik men utan rekursion. Istället använder vi en while-loop för att traversera genom trädet. Vi börjar med att kontrollera om trädet är tomt och om så är fallet, skapar vi en ny rotnod. Annars startar vi vid roten och rör oss nedåt genom trädet. För varje nod kontrolleras om värdet ska placeras till vänster eller höger. Om rätt position är tom, placeras noden där. Om inte, fortsätter metoden genom att uppdatera den aktuella noden till `left` eller `right` beroende på värdets storlek.

Lock-up metoden

`Lock-up` metoden söker efter ett specifikt värde i trädet på ett iterativt sätt. Den börjar vid rotnoden och använder en while-loop för att traversera trädet. För varje nod jämförs dess värde med det sökta värdet, om värdena är lika returnerar metoden true, vilket indikerar att värdet finns i trädet. Om det sökta värdet är mindre än nodens värde, flyttas loopen till vänster subnod. Om värdet är större, flyttas det till höger subnod. Om loopen når en null-nod betyder det att vårt värde inte finns och metoden returnerar false, vilket betyder att värdet inte finns i trädet. Här nedan kan vi se implementation av `lock-up` metoden.

```
        public boolean lockUp(Integer value) {

            Node current = root;
            while (current != null) {
                if (current.value == value) {
                    return true;
                } else if (value < current.value) {
                    current = current.left;
                } else {
                    current = current.right;
                }
            }
            return false;
    }
```

Binärsökning vs Lock-up

I assignment 3 såg vi att `binärsökning` hade en tidskomplexitet på *O(log(n))*. `lock-up`-metoden har, precis som binärsökning, en tidskomplexitet på *O(log(n))*, under förutsättning att trädet är balanserat. Om trädet däremot är obalanserat och mer liknar en länkad lista, vilket kan hända om vi lägger till noder i storleksordning försämras tidskomplexiteten istället och blir *O(n)*.

Traversera trädet med explicit stack (in-order)

Vi kan traversera vårt träd med en explicit stack istället för rekursion. Stacken som används för att traversera trädet är vår dynamiska arraystack från assignment tre. Metoden `stackPrintTree` börjar genom att initiera stacken och sätta den aktuella noden till trädets rot. Därefter kör vi igenom hela trädet nedåt till den noden längst ner till vänster(minst talet), och pushar på vägen ner varje nod på stacken.

När vi har nått längst ner till den mest vänstra noden, börjar vi arbeta oss uppåt. Metoden poppar en nod från stacken och skriver ut dess värde. Om denna nod har ett höger barn, flyttar metoden till höger och fortsätter sedan nedåt till vänster, samtidigt som varje vänsternod pushas på stacken. Om noden däremot saknar höger barn, poppar vi nästa nod från stacken och fortsätter på samma sätt tills stacken är tom och alla noder har besökts.

`StackPrintTree` metoden säkerställer att noderna skrivs ut i in-order-sekvens, det vill säga (vänster, rot, höger).Stacken hanterar själv ordningen utan att behöva använda rekursiva anrop. Nedan kan vi se själva implementationen över hur vi traverserar trädet och skriver ut noderna.

```
public void stackPrintTree() {

    dynamicStack stack = new dynamicStack(20);
    Node current = root;

    while (current.left != null) {
        stack.push(current);
        current = current.left;
    }

    while (current != null) {
        System.out.println(current.value);

        if (current.right != null) {
            current = current.right;
```

```
                while (current.left != null) {
                    stack.push(current);
                    current = current.left;
                }
            } else {
                if (stack.isEmpty())
                    return;
                current = stack.pop();
            }
        }

    }
```

Slutsats/diskussion

I denna rapport har vi utforskat datastrukturen träd och kollat på hur man implementerar ett binärt träd. Vi har undersökt hur man lägger till och söker efter noder med både rekursiva och iterativa metoder. Vi kollar hur vi kan använda en explicit stack för att traversera trädet i *in-order*, istället för att använda en rekursiv metod och programmeringsspråkets implicita stack.

Att traversera trädet iterativt jämfört med att använda rekursiv programmering har sina fördelar. För ett mycket djupt eller stort träd kan det vara fördelaktigt att välja en iterativ metod istället för en rekursiv, eftersom man då undviker risken för en stack-overflow till följd av för många funktionsanrop. Å andra sidan kan rekursiv programmering, särskilt för den som är erfaren, leda till mer strukturerad och lättläst kod. Det rekursiva tillvägagångssättet kan ge en tydligare struktur och göra koden mer lätt läst.