Träd

Mattias Sandberg

29 Sep 2022

Introduktion

I tidigare uppgift har vi arbetat med länkade datastrukturer och det är väldigt effektiva när det kommer till mer komplexa datastrukturer. Utöver länkade listor finns det trädstrukturer som har fått sitt namn då datastrukturen utgår från en rot som förgrenar sig till fler och fler grenar. Den grenen som inte utökar sig / fortsätter att förgrena sig kallas för ett löv. I denna uppgift så kommer vi att arbeta med så kallade "binary trees" vilket betyder att en gren kommer fördela sig i två om det inte är ett så kallat löv. Det operationerna som denna uppgift kommer att handla om är addera, söka, konstruera och ta bort föremål i trädstrukturen. Nedan har jag lagt in en simplifierad bild över hur ett träddiagram kan visualiseras från hemsidan ($https://sv.wikipedia.org/wiki/Binärt_sökträd$) . I det kommande uppgifterna så använder jag penna och papper för att kunna skriva och konstruera koden då det kan bli svårt att komma ihåg grenarna i huvudet och därmed tycker jag att det är bra att rita upp en datastruktur som nedan:

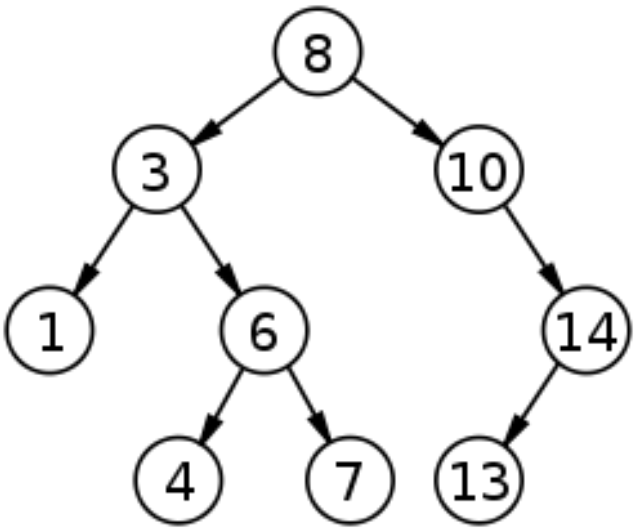

Figure 1: Simpliferad bild över Träd datastrukturer

Add operation

För att börja med denna uppgift så konstruerar jag ett binärt träd där varje nod har en nyckel och ett värde, en höger och vänster gren. Nycklarna är mappade till värden och vi antyder att varje nyckel är enbart mappad till ett specifikt värde. Vi använder en kapslad node för att beskriva trädets interna struktur, där trädet bara har en egenskap vilket är trädens rot. I trädstrukturer så är trädet tomt om roten är en noll-pekare. Nu så utgår vi från att trädet är strukturerat, dvs att alla noder med nycklar som är mindre än roten finns i den vänstra grenstrukturen och tvärtom. Den första operationen är en ”Add” operation vilket betyder att jag ska lägga till en ny nod (löv) till trädstrukturen som mappar nyckeln till värdet. Och om nyckeln redan är nuvarande så uppdaterar vi värdet på noden. Nedan kommer kodutdrag för hur jag implementerade min kod i java samt en graf över benchmarken då jag mätte exekveringstiden för ett ökande dataset:

```
private Node Add(Node cur, int val)
{
    if (cur == null)
    {
        return new Node(val);
    }
    if (val < cur.val)
    {
        cur.left = Add(cur.left, val);
    } else if (val > cur.val)
    {
        cur.right = Add(cur.right, val);
    } else {
        // Värdet existerar redan i detta fall
        return cur;
    }
    return cur;
}
```

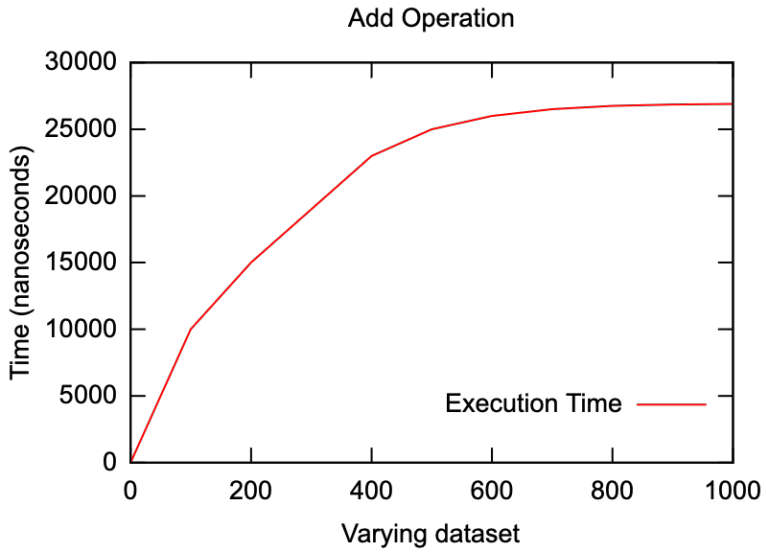

Figure 2: Graf över Add operation med träd datastrukturer

Tidskomplexiteten för att addera en ny nod är i det genomsnittliga fallet O(log(n)) vilket även min graf representerar som är taget från min benchmark. Om jag skulle ha haft otur med hur min datastruktur skulle ha sett ut så hade jag fått en graf som representerade Ordo O(N) då det är worst case senario för add/insert operationer i trädstrukturer.

Lookup operation

I denna uppgift så kommer vi att utföra en lookup operation på en binary tree structure. Detta betyder att vi ska söka och hitta värdet som tillhör en specifik nyckel. Om det inte skulle finnas någon nyckel så ska funktionen returnera null. Nedan kommer delar av koden som jag har implementerat samt en slutsats till resultatet

```
private boolean lookup(Node cur, int val)
{
    if (cur == null)
    {
        return false;
    }
    if (val == cur.val)
    {
```

```
            return true;
        }
        return val < cur.val
          : lookup(cur.right, val);
          ? lookup(cur.left, val)
}
```

Tidskomplexiteten är även för lookup operationen Ordo O(log(n)) för average case. Vilket är rimligt då den utför liknande operationer som förra programmet för att hitta värdet till dess angivna nyckel. Tidskomplexiteten för denna operation i worst case senario är Ordo O(n).

Depth first traversal

Det är väldigt ofta som man vill gå igenom alla värdena som finns i en trädstruktur och i denna uppgift så kommer vi att gå igenom hur man kan göra detta på ett lämpligt sätt. Då vi har det lägsta värdena till vänster i trädstrukturen kan vi gå igenom alla värden från vänster till höger dvs lägst till högst värde. Denna metod kallas för depth first och utgår från att man går ned så långt det går i den vänstra grenen innan man fortsätter till nästa grenar. För att göra detta och testa det så implementerar jag en printmetod med hjälp av den implicita stacken som jag har använt i flera uppgifter förut och som finns i java. När vi rekursivt har skrivit ut värdena i den vänstra grenen så hoppar vi tillbaka till den nuvarande noden och printar värdet och sedan rekursivt gör samma sak för det högra grenarna. Vi kan implementera inorder traversal utan rekursion genom att först initiera den nuvarande noden men en rot. Och medan cur ej är noll och stacken ej är tom ska funktionen fortsätta pusha vänstra grenar till stacken tills vi når det vänstra lövet. Sedan ska vi popa det vänstra lövet från stacken och sätta cur till den högra grenen av den popade noden. Detta representeras i kod nedan:

```
public void DepthFirstTraversal()
{
    Stack stack = new Stack<>();
    Node cur = r;

    while (cur != null || !stack.isEmpty())
    {
        while (cur != null)
        {
            stack.push(cur);
            cur = cur.left;
        }
```

```
        Node t = stack.pop();
        visit(t.value);
        cur = t.right;
    }
}
```

An iterator

Det första steget i denna uppgift är att deklarera en binarytree struktur för att sedan kunna implementera ett upprepande gränssnitt. Klassen ska sedan förses med en metod som returnerar en iterator. I denna uppgift så kommer vi att returnera en iterator som skapar en sekvens med integers, det vill säga värden av det noder i trädstrukturen. Nedan kommer kodsegment från min implementation av en iterator i ett binärt träd där jag implementerar en stack.

```
public class Iterator{
Stack<TN> stack;
public Iterator(TN r) {
stack = new Stack<TN>();
while (r != null) {
stack.push(r);
rot = r.left;
}
}
public boolean HasNext() {
return !stack.empty();
}
 public int Next() {
TN node = stack.pop();
int res = node.val;
if (node.right != null) {
node = node.right;
while (node != null) {
stack.push(node);
node = node.left;
}
}
return res;
}
}
```

Using a stack

I tidigare uppgifter har vi implementerat en stack som minnesallokering men när det kommer till våra iterationer så kan det uppstå problem. I Next() metoden så kan det uppstå problem om nästa nod inte är null eller ett blad (höger-rot). Då är den vänstra roten redan null eller så är den redan i stacken. Det vi behöver gör i vårt program är att följa den högra rotens högra nod. Och när vi går längs med dessa noder så pushar vi in det noderna i stacken för att sedan kunna returnera dessa värden. Tricket med detta är att vi måste ha i minnet att det inte ovillkorligt kommer att vara en "Parentnode" till vår nästa nod. Det vill säga att det kan vara ett värde som från stacken ska popas till den trädstrukturens högra gren-nät.

Till sist i min rapport så vill jag visualisera min förklaring. Detta ska jag göra genom att skapa en iteration med slumpmässiga tal för att sedan visa hur ytterliga värden läggs till. Trädet kommer att bibehålla sina ursprungliga värden och det jag kommer att visa hur det nya värdena ska läggas till, dvs från rätt nod på rätt sida (Right/Left) beroende på storlek på det nylagda talet. I exemplet nedan kommer är 166 "ParentNode" och då jag vill lägga till $42 < 166$ så kommer den välja den vänstra grenen och då $42 > 37$ kommer den skapa en nya "BladNode" på den högra sidan om 37. Därefter läggs det nya värdet in i vår Binarytree med implementation av stacken.

```
/*
                                166
                                /\
                               37 182
        //Lägger till 42//
                                166
                                /\
                               37 182
                                 \
                                 42
*/
```