Uppgift 8

Simon Moradbakti

HT 2024

Introduktion

I den här uppgiften ska arbeta med träd, och även binära sådanna. Träd är ännu mer komplicerade att arbeta med än länkade listor och passar därför till arbete som kräver mycket data samt minne. Namnet träd bygger på att vi har olika komponenter som tillsammans måste fungera tillsammans, dessa är roots, branches och leafs som alla bekant är komponenter för ett träd. Denna rapport kommer gå in på binära träd och vad som utmärker dem. Tidskomplexiteten kommer även utredas samt hur det funkar med stacken och DFT när man använder sig av träd. Detta kommer göras med så kallade benchmarks.

A binary tree

För att kunna besvara frågorna på denna deluppgift måste vi först koda vårt binära träd. Lyckligtvis behöver detta inte göras helt från början då vi fått en del kod att arbeta med. Vi följer instruktionerna och ser till att vårt träd har ett värde, en höger och en vänster branch. Vi ska använda oss av lookup add metoder. Med add menas att ett nytt leaf ska implementeras som har ett värde, som ska sorteras. Alltså likt en array som kan förlängas. Med lookup menas att vi kontrollerar om värdet finns och isf returnerna true eller false beroende på utfallet. Skribenten valde att skriva add metoden som följande:

```
public void add(Integer value) {
    root = addRecursive(root, value);
}
private Node addRecursive(Node current, Integer value)
    {
    if (current == null) {
        return new Node(value);
    }
    if (value < current.value) {
        current.left = addRecursive(current.left,
            value);
    } else if (value > current.value) {
```

```
        current.right = addRecursive(current.right,
            value);
    }
    return current;
}
```

lookup metoden implementerades såhär:

```
public boolean lookup(Integer key) {
    return lookupRecursive(root, key);
}
private boolean lookupRecursive(Node current, Integer
   key) {
    if (current == null) {
        return false;
    }
    if (key.equals(current.value)) {
        return true;
    }
    return key < current.value
            ? lookupRecursive(current.left, key)
            : lookupRecursive(current.right, key);
}
```

Nu kan vi köra ett benchmark på den fullständiga koden, rita upp vad vi får och sedan avgöra vad för tidskomplexitet blir! Nedan är en tabell på några mätvärden:

tid (ns)	element
1230000	10
2490000	100
7570000	1000
15600000	10000
73000000	100000

Table 1: Resultat av exekvering

Vi kan rita upp detta med hjälp av verktygen på excel och se vad detta ger oss för graf.

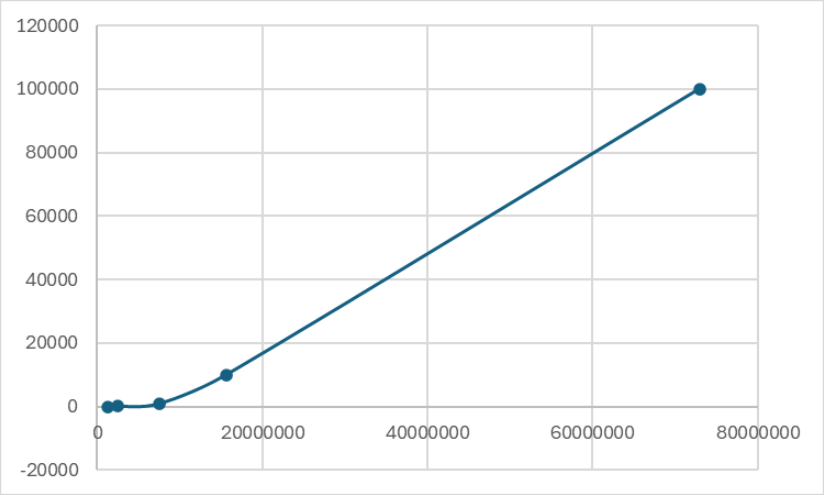

Figure 1: Grafen på tidskomplexiteten

Vi ser att för varje gång antalet element ökar med en faktor 10, så ökar tiden någonstans mellan 2 till 5 gånger. Vi kan alltså utesluta både en kvadratisk och linjär ökning vilket endast lämnar en logaritmisk tidskomplexitet, alltså O(log(n)). Om man hade haft en ordnad frekvens i ett binärt sökträd blir det obalanserat som en länkad lista vilket ger tidskomplexiteten O(n), en försämring! Jämförs lookup algoritmen med binär sökning så kommer man fram till att i en sorterad array så får binärsökning tidskomplexiteten O(log(n)), precis samma ger lookup algoritmen, blir trädet obalanserat blir tidskomplexiteten linjär och därför effektivare. Efter utförandet av det givna experimentet kom det fram till denna slutsats att den rekusiva metoden är mer konsis men i djupa träd kan ge stack overflow medans den iterativa metoden är längre med ger ej stack overflow.

An explicit stack

I den här deleuppgiften ska man använda sig av stack för att traversera ett binärt träd istället för rekursion. Vi börjar vid roten och därefter pushar vi ut alla noder som är i vår vänsta gren av trädet till vi kommer till en nod utan någon nod under sig. Gör vi det så börjar vi poppa noderna så de skrivs ut. Detta betyder i praktiken att det endast sker när hela trädet är bearbetat eller just traverserat. Vi flyttar sedan höger ut på vårt träd, om det går alltså. Vi upprepar processen. Vi gör detta tills hela trädet är traverserat och stacken är tom. På detta sätt kan vi gå igenom hela trädet utan att använda oss av rekusion. Rent kodmessigt skulle det kunna se ut såhär:

```
public void print() {
    Stack<Node> stack = new Stack<Node>();
```

```
    Node current = this.root;
    while (current != null || !stack.isEmpty()) {
        while (current != null) {
            stack.push(current);
            current = current.left;
        }
        current = stack.pop();
        System.out.println(current.value);
        current = current.right;
    }
}
```

Sammanfattning

Denna rapport undersökte binära träd. Den beskriver trädets olika komponenter och tidskomplexiteten för olika operationer. Benchmarks har utförst för att kunna bevisa tidskomplexiteten. Det har även gåtts igenom vad balanserade och obalanserade träd kan leda till. Rapporten berör även hur man kan använda stacken för att traverasera listan och beskriver hela dess process.