Träd

Hiba Qutbuddin Habib

Spring Fall 2002

Introduktion

Träd datastruktur är en mer komplex datastruktur i jämförlse med en länkade lista. Träd som datastruktur har sitt ursprung i en rot som delar sig i grener, dess grener i sin tur delas upp i flera grener. Den sista grenen som inte delar up sig kallas för löv. När man snakar om träd sturktur då brukar man referera till föräldrar och barn, alltså att varje rot är en föräder och varje förälder kan max ha två barn, alltså två grenar.

I den rapporten ligger fokus på binära träd, då en binärt träd är en träd som har en viss ordning, dvs på höger sida av roten ligger all värde som är större än själv roten medan på vänster sidan så finns det alla värde som är mindre än själv roten.

Binärt träd

I denna delen kommer vi att implementera en binärt träd. Klassen för binärt träd kommer att bygga på en annan klass som vi kallar för "Node". I denna klassen kommer vi att ha nyckel, värde, vänster gren samt höger gran. För varje specifik nyckel i träd, finns det ett givet värde. För att kunna lägga till noder i trädet, så behöver vi skapa en metod för det, denna metoden implementeras rekrusivt. Metoden lägger till en ny nod till trädet som mappar nyckelar till värde. Följande metod kommer att implementeras i två steg, först implemntera vi denna metoden i klassen nod, där vi lägger till noder i en viss ordning, detta implemteras enligt:

```
private void add(Integer k, Integer v) {
//om det redan samma nyckel existerar, uppdateras värdet
    if (this.key == k)
        this.value = v;
//kontrollera om k är mindre än nyckel i trädet
    if (this.key > k) {
//om k är mindre och vänster gren är tom så skapas en ny nod
        if (this.left == null) {
            this.left = new Node(k, v);}
```

```
// annars adderas en ny nod till vänstra grenen
//genom att anropa rekrusivt metoden add
         else {
            this.left.add(k, v); }}
// annars om k är större
         else {
//kontrolleras höger sidan om det är tom
            if (this.right == null) {
//så skapas en ny nod
            this.right = new Node(k, v); }
// annars adderar vi en ny nod i rätt ordning
         else {
            this.right.add(k, v);} }
```

Den andra metoden skapas i klassen för binär träd, då den anropar den första metoden i klassen node ifall en viss vilkor inte gäller. Föjlande metoden implementeras enligt;

```
 public void add(Integer k, Integer v) {
 // om vi inte ha någon träd
    if (root == null)
//då skapar vi rooten
        root = new Node(k, v);
//om vi har redan en träd så anropas metoden add från klassen nod
    else
        root.add(k, v); }
```

För en binärt träd kan vi även ha en annan metod, denna metod användes för sökning genom träd, alltså som argumnet får metoden en specifik nyckel som den ska hitta i trädet sedan returner metoden värdet i just den nyckel. Detta kan implementeras enligt:

```
public int lookup(Integer key) {
    Node current = this.root;
    while (current != null) {
        if (current.key == key)
System.out.println("The value at key: " + key +" is " +current.value);
        if (current.key < key)
            current = current.right;
        else
           current = current.left; }
        return -1;  }
```

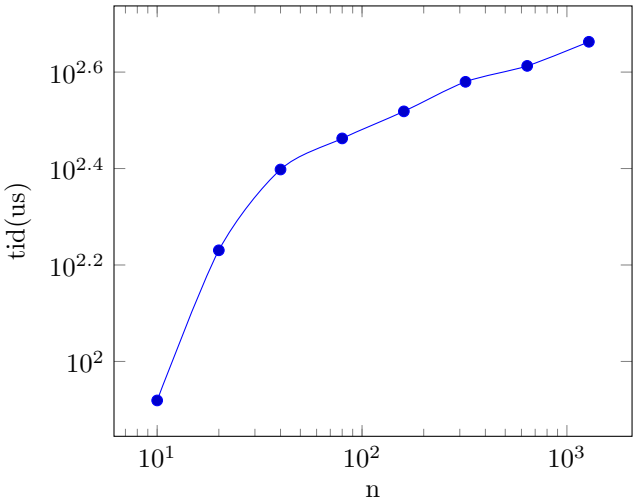
För att vi ska kunna få en bättre förståelse över tidskomplexitet för denna datastruktur, så kommer vi att mäta tiden när vi söker efter en specifik nyckel i trädet. Vi använder metoden ”Lookup” ovan. I ”Benchmarks” som används för tidmätningar kommer jag att använda slumpmässiga tal som nyckelar samt värdet, jag kommer också at ge metoden ”Lookup” random värde som nycklar.

Resultatet

Mätningar gav följande resultat:

n(längd)	**10**	**20**	**40**	**80**	**160**	**320**	**640**	**1280**
t(us)	83	170	250	290	330	380	410	460
t/log(n)	0.036	0.055	0.067	0.066	0.066	0.065	0.064	0.064
t/n	0.0083	0.0083	0.0062	0.0036	0.0021	0.0012	0.00065	0.00036

Table 1: Tidmätning för en binärt sökning träd



Tabellen och grafen ovan visar resultat över mätning för sökning efter en specifik nyckel i en binärt träd. I tabellen kan vi se att resultatet är mer logaritmisk än linjär då konstant t/n går mot noll ju större trädet blir, däremot värdet för $t/log(n)$ verkar gå mot ett stabilt värde. Detta innebär att vi får en logaritmisk algoritm, som tillhör mängden $O(log(n))$. I bästa fallet stämmer detta för en binärt sökningsträd, då med bästa fall menar vi att trädet är ganska balanserad vilket gör att vid sökning kan man bortse från nästan halva trädet. Dock ett binärt sökningsträd har också ett värsta fall, då är trädet ej balanserad vilket gör att när vi söker efter en element i den djupare delen så kommer man att behöva söka genom ungefär n element och då tillhör denna algoritmen mängden $O(n)$. I den "Benchmark" som jag använde för mätningen så användes det slumpmässiga samt oordnade tal som nyckelar, ifall vi skulle ha haft redan ordnade värde för nycklar då skulle vi ha fått en obalanserad binärt söknings träd vilket skulle ledde till ett värsta fall. När vi jämför denna algorimten med binärt söknings algoritm som vi utförde i en tidigare uppgift och då resulatet visade att binärt söknings algoritm tillhörde mängden $nlog(n)$, så kan vi se att binär sökning tar längre tid än binär träd sökning om trädet är ganska balanserad.

”Depth first traversal”

En binärt träd är ingen linjärt datastruktur, detta gör det svårare att gå genom trädet när man vill besöka varje nod. För denna anledningen kan vi hitta flera användbara metoder för att traversera en träd. En av metoderna kallas för ”Depth First”, då denna typen av traversering bygger på att först gå så djupt som möjligt i varje gren innan man börjar utforska andra ”syskon” grenar. Denna metoden kan tillämps på flera olika sätt, av dessa kan man nämna ”in-order, pre-order samt post-order”.

En iterator

An iterator används för att loopa genom en data struktur för att sedan returnerar innehållet av det. På vilket form denna innehållet ska presenteras skiljer det sig från en till en annan men det finns flera sätt att välja emellan. I denna uppgiften kommer iterator att gå genom noder och returnera värdet som de anhåller, vi använder metoden ”Depth first in-order” för att presentera dessa värde. För denna iterator kommer vi att använda en stack. För att kunna få en inorder traversering så kommer man behöva gå till vänster, därifrån kommer värderna att pushas in i stackan fram tills man når längst ner till löven. Om dessa grenar har högra grenar så pushas även de in. Med principen sist in först ut för stacken kommer alla värde att ”popas” ut en efter en tills vi kommer återigen till roten. Detsamma gäller för den högra grenen.

Iterator kommer att innehålla följande metoder:

```
 @Override
public boolean hasNext() {
    return !stack.isEmpty();}
@Override
public Integer next() {
    if (!hasNext())
    throw new NoSuchElementException();
    Integer key = stack.pop();
    return key; }
```

Från början pushas värderna från trädet inne i stacken. Metoden ”hasnext” anropar metoden ”isEmpty” från stackan för att kolla om stacken innehåller värde som man kan ”poppa”, denna metoden returnera antigen ”false” eller ”true”. När det gäller metoden ”next” så har den i uppgift att returnera värdera som finns i stacken. Om vi lägger till en ny värde till träd när vi redan ha anropat metoden next i iterator, så har vi förlorat värderna eftersom vi poppade allt och då behöver vi gå genom trädet på nytt så att vi kan få med den ny tllagde värdet i trädet.