Träd

Viktor Björkén

September 2022

1 Introduktion

I denna uppgift kommer vi att jobba med binära sök träd och dess olika funktioner. Vi kommer att bygga olika metoder för att söka och hitta i dessa träd samt jämföra tidskomplexiteten på dessa med metoder från föregående uppgifter.

Det binära trädet utgår ifrån en root och går sedan ner i grenar de grenar som pekar vidare ner på null kallas blad. Dessa är alltså de sista i trädets struktur.

2 Binärt träd

Vi kommer börja uppgiften med att konstruerar vårat binära träd, Detta gör vi genom att anväda den kod vi har fått.

Ett binärt träd fungerar genom följande att den vänstra delen alltid är ett mindre värde än sitt föreldar nod. Samma sak fast med större värden händer på höger sidan. Dessa grenar utgår från början från en så kallad ”root”. Alla grenar som går ut ska själva kunna ses som ett träd.

Ett (sorterat) binärt träd är en bra funktion att anväda då alla element ligger i ordning kommer det att vara enklare att hitta och söka.

I första deluppgiften ska vi addera in 2 metoder där den ena adderar en ny node, och en ”lookup” där vi ska söka och returnera det värde som har med rätt key att göra.

Vi implenterar dessa genonm följande kodrader.

Lookup metoden ser ut på följande

```
public Node search(Node root, int key) {
    if (root == null || root.data == key)
        return root;
    if (root.data < key)

        return search(root.right, key);
```

```
     return search(root.left, key);
}
```

”Addera” metoden gör vi på följande

```
public Node insert(Node root, int key) {

     if (root == null) {
         root = new Node(key);
         return root;
     }

     if (key < root.data)
         root.left = insert(root.left, key);
    else if (key > root.data)
        root.right = insert(root.right, key);
    return root;
}
```

Det vi ser är att vi har samma genomsnittligt tidskomplexitet på

$$O(log(n))$$

Detta är dessamma som våran förra uppgift där vi undersökte våran binär sökning.

När vi gör våran benchmark på våra följande metoder ser vi att det som skrivits ovan stämmer. i värsta fall för de olika metoderna får vi

$$O(n)$$

men ett genomsnitt på

$$O(log(n))$$

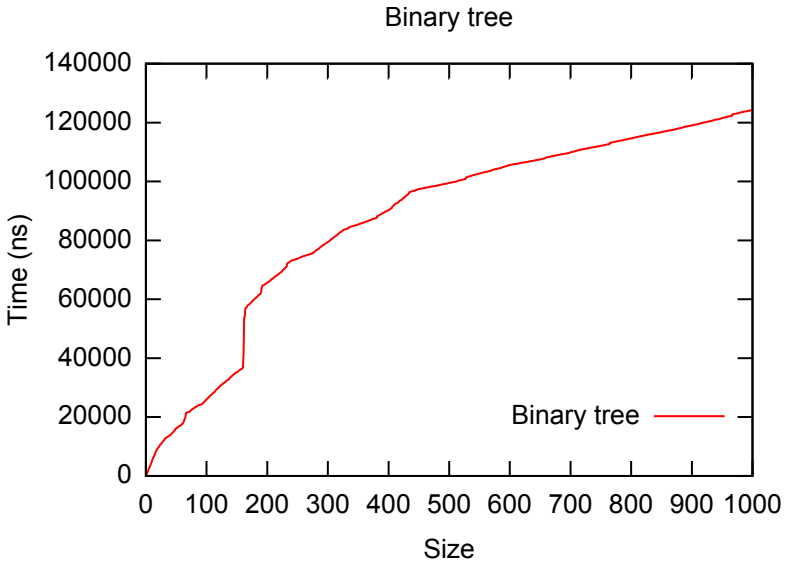

Figure 1: graf över binär trädets tidskomplexitet

3 Traversering

Det finns olika sorters sätt att med hjälp av travversering göra på binära träd. Dessa olika gör så att att träde blir uppbyggt/sorterad på olika sätt.

Vi kommer använda oss utav är ”djupet-först”/DFS, denna kommer gå igenom noderna på följande sätt.

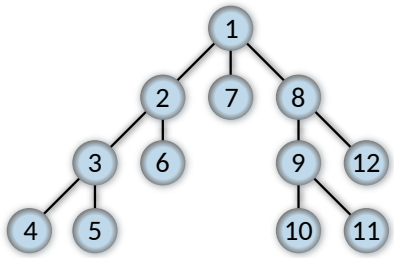

Figure 2: Djupet först sökning

Här kan vi se hur ”djupet-först” kommer att gå egenom noderna, där den börjar på rooten för att sedan börja ner till vänster och sedan gå mot den högra delen av trädet.

4 Iterator

En iterator kommer möjliggöra att vi har en datastruktur som kan ”hålla” trädet samtidigt som som vi ”håller” en postion där nästa element kan hittas.

Skillnanden mellan en implecit iterator och en explecit är att den implecita är något som i t.ex java redan är inbyggt för att itererar medans en explecit är något som vi som andvändare sjävla bygger.

För att implementera våran iterator kommer vi att behöva använda koden för ”TreeIterator” som vi fått och sedan bygga de funktioner som saknas. Vi göra detta med följande kod.

För våran hasNext() respektive next() funktion gör vi följande.

```
// hasNext()
  public boolean hasNext(){
        if(stack.empty()){
            return false;
        }return true;
    }
// next()
 public int next(){
        TreeNode node = stack.pop();
        pushToLeft(node.right);
        return node.val;
    }
```

När vi sedan kör denna iterator kommer alla värden på vänstra sidan att pushas till stacken. hasNext() funktionen kommer då att returna sant om det finnns element i stacken, next() kommer att returna det näst minsta numret i trädet.

5 Stack

När vi ska implementera våran ”treeIterator” börjar vi med att först initsiera våran stack. När vi gjort detta sätter vi våran nuvarande nod till att våra rooten. Därefter körts en while-loop så länge den inte är ”null”, när denna är igång kommer vi att addera vårat nuvarande värde i stacken och sedan flytta våran nuvarande till vänster.

Våran ”treeIteration” implementation ser då ut på följande sätt

```
class treeIterator {
    private Stack<Node> traversal;

    InorderIterator(Node root)
    {
        traversal = new Stack<Node>();
        moveLeft(root);
```

```
}

private void moveLeft(Node current)
{
    while (current != null) {
        traversal.push(current);
        current = current.left;
    }
}
```

Om vi skulle addera några få element efter att vi har börjat iterara kommer vi beroende på vilket element det är att hanteras olika. Är det ett element som skulle ha hamnt i en del av trädet som redan gott igenom iteratorn kommer detta värde att gå förlorat medans om det är ett värde som kommer efter kommer detta att kunna läggas in och sedan iteraras.