# Träd

Elis Gunnarsson

Fall 2022

## Introduktion

Länkade listor som vi utförde på förra veckans uppgift kan vara väldigt användbart men den blir som mest användbart när strukturen är lite mer komplext. Nästa steg är en trädstruktur. Trädstrukturen kommer från hur strukturen ser ut, där det är rötter som delas upp, och de nya rötterna i sin tur delas upp igen osv.

Vi kommer jobba med binära träd. En rot kommer alltid delas upp till två nya rötter, de som inte gör det blir ett blad. Vi kommer jobba med olika operationer som: construction, adding, searching for and removing item.

### Binärt träd

När vi ska konstruera det binära trädet så finns det några saker vi ska ta hänsyn till. Trädet ska bestå av noder, dessa noder ska ha en key (adress), ett värde, en rot som går till vänster samt en rot som går till höger (höger och vänster rot).

Nu ska vi använda två metoder, 1. add(Integer key, Integer value) och 2. lookup(integer key). Den första metoden lägger till en ny eller adderar en node till trädet. Ett extra blad kan man säga. Om värdet redan finns så kommer noden att uppdateras. Metod två letar efter om värdet redan finns i någon key. Om det inte finns returneras null.

Dessa två metoder ska vi implementera till ett träd, men trädet ska vara sorterat. Där vänster gren eller rot ska vara tal som är mindre än första key, och där högra grenen/roten ska vara tal som är större. När vi jämför lookup algortitmen med den binära sök algoritmen som vi genomförde på en av våra gamla uppgifter. Då kommer vi fram till att för lookup metoden så blir exekveringstiden O = log (n), vilket är densamma för den binära sök algoritmen i ”avarage” case.

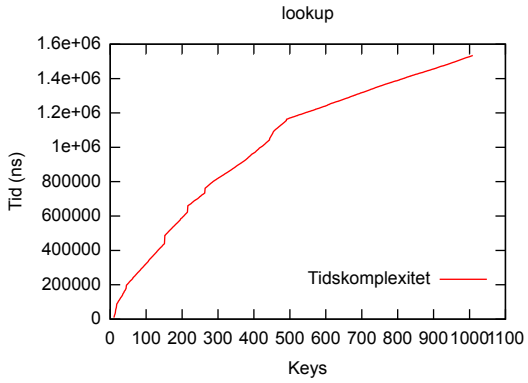


Figure 1: Tidskomplexitet - lookup

Om vi hade konstruerat det binära trädet utifrån sorterade sekvenser av nycklarna hade vi istället fått en annan tdiskomplexitet där O = N, dvs. linjär.

### Korsa trädet

Väldigt ofta vill du kunna gå igenom alla värdena som ditt träd innehåller. I vårt fall som vill sortera i storleksordning från väsnter till höger (minst till störst). Frågan är hur ska vi göra detta? Ett sätt vi kan göra det på är att gå längst med den vänstra grenen tills vi når det minsta värdet. Därefter går vi succesivt till högra grenen i storleksorning. Det engelska namnet för denna metoden är ”depth first” vilket innebär att du går så djupt till vänster först innan du överväger något annat.

Genom att lägga till en print metod till själva node klassen så printer den ut värdena i ”depth first”.

```
public void print() {
if(left != null)
left.print();
System.out.println(" key: " + key + "\tvalue: " + value);
if(right != null)
right.print();
}
```

### En iterator

Nu vill vi använda oss av ”depth first” metoden fast vi vill få ut ett värde åt gången istället för hela trädet när vi ska sortera det. Då kan vi använda oss av iterationer (upprepningar). Med iterationen vill vi få en datastruktur där vi lyckas vå ett värde man även möjligheten till nästa vid behov.

I java kan vi definera en class som impleneterar en iterator. där iteratorn har två metoder: hasNext() och next(), Där hasNext() returnerar ”true” om iterationen har fler värden och next() returnerar nästa värde i iterationen.
¨

När vi ska sortera hela trädet med hjälp av iterationer finns det två vägar vi kan gå. Den första är att vi hela tiden sparar det värdet vi ska byte i en lista (array) och sedan byter ut i listan hela tiden, dock är det väldigt ineffektivt. Det andra sättet är att vi använder oss utav en stack som hela tiden lokaliserar vart vi är.

```
public class Iterator {
Stack<TradNode> stack;

public Iterator(TradNode rot) {
stack = new Stack<TradNode>();
while (rot != null) {
stack.push(rot);
rot = rot.vanster;
}
}

public boolean hasNext() {
return !stack.tom();
}
 public int next() {
TradNode node = stack.pop();
int resultat = node.val;
if (node.hogert != null) {
node = node.hoger;
while (node != null) {
stack.push(node);
node = node.vanster;
}
}
return resultat;
}
}
```

Hur vi implimenterar iterationen av det binära trädet i en stack. Kortfattat så hur det fungerar kommer vi fylla på stacken med värden längst med hela vänstra roten tills vi kommer till ett blad. Därefter kommer vi med hjälp av next() metoden pusha våra värden ut till trädet till rätt ordningen. Med hjälp av hasNext() metoden kommer den se till så att så länge vi har värden

i stacken (true), så kommer next() metoden att köras. HasNext() metodens uppgift är att med hjälp av stacken hålla kolla på vart i trädet vi är, och se till så alla rötter får rätt värde.

### Använda stack

Utöver det jag nämnde i förra stycken om hur vi använder oss utifrån en stack till våra iterationer så kan det uppstå lite problem/klurigheter. Det uppstår om till exempel nästa node i next() metoden inte är ett blad eller null, dvs. en högerrot. (vänstra roten är då null eller eller redan i stacken). Då behöver vi följa från längs med den högra rotens vänstra node så långt ner till vänster som vi gjorde tidigare. Under tiden pushar vi in the noderna vi går förbi till stacken då vi kommer returnera dessa värdena senare.

Vi måste hela tiden ha i åtanke att noden vi får fram i stacken inte nödvändigtvis kommer vara en ”förälder-node” till nästkommande node, utan det kan vara ett värde som senare ska popas från stacken till en annan gren i det högra ledet av trädet.

Jag har i det tidgare styckena förklarat hur min träditeration fungerar. Men om vi istället skapar en iteration med några få värden, för att sedan lägga till ytterligare värden. Det som sker är att trädet kommer ha kvar sina ”startvärden” men de nytillkomna värden kommer läggas inom rätt gren och storlek. Säg ett ett av värden är mindre än den första högra grenen men större än alla de andra noderna i vänsterledet. Då kommer det skapas en ny ”bladnode”. Dvs. en node utan vidare grenar.

Ett försök till att visualisera mitt exempel. Säg vi har vår föräder-node högst upp som är 10. Höger gren har vi 20 och vänster har vi 5. Om vi då lägger till ett nytt värde, säg 7. Då kommer den första steget gå vänster, och sen skapa en ny ”bladnode” till höger om femman.

```
/*
    10
   /  \
 5     20

 *adderar 7*

    10
   /  \
 5     20
  \
   7
   */
```