Labbrapport Labb 3 D0012E

Författare
Adam Gatmon | adagat-1@student.ltu.se
David Malmblad | maldav-1@student.ltu.se
Albin Kullberg | albkul-0@student.ltu.se

LULEÅ TEKNISKA UNIVERSITET

15 december 2023

Innehåll

1 Inledning

I denna laboration utforskar vi binära sökträd, och hur man kan hålla dem balanserade vid insertion-operationer på trädet. Genom att ändra storleken av varje nodens delträd, strävar vi att förstå hur detta påverkar tidskomplexiteten.

2 Teori

2.1 Binärt Sökträd (BST)

Ett binärt sökträd är en hierarkisk datatruktur där varje nod har högst två barn. Den har den egenskapen att varje nod i det vänstra delträdet har ett värde mindre än nodens värde, och varje nod i det högra delträdet har ett värde större än nodens värde.

2.2 Sökträd D med Storleksbegränsningar

Sökträdet D har särskilda egenskaper där storleken av vänster och höger delträd för varje nod begränsas av en konstant c (där $\frac{1}{2} \leq c \leq 1$). Detta säkerställer en viss form av balans i trädet. Ett perfekt balanserat subträd är då ett träd där det totala antalet subnoder på höger respektive vänster sida aldrig överstiger hälften av hela subträdets totala antal noder. Ett lågt c-värde kommer tvinga trädet att ombalanseras (bygga om trädet till ett perfekt balanserat träd) oftare, men som följd säkerställa att trädet är balanserat, medan ett högt c-värde kommer låta trädet bli mer obalanserat innan en dyr ombalansering måste ske, men kommer som följd också öka söktiden genom trädet, i teorin upp till $O(n)$.

3 Metod

För att utföra och utvärdera prestanda för våran sökträdstruktur har vi implementerat den i programmeringsspråket Python. Detta inkluderar att säkerställa att varje nod har vänster och höger delträd vars storlek inte överstiger c gånger storleken av det delträd noden rotar i.

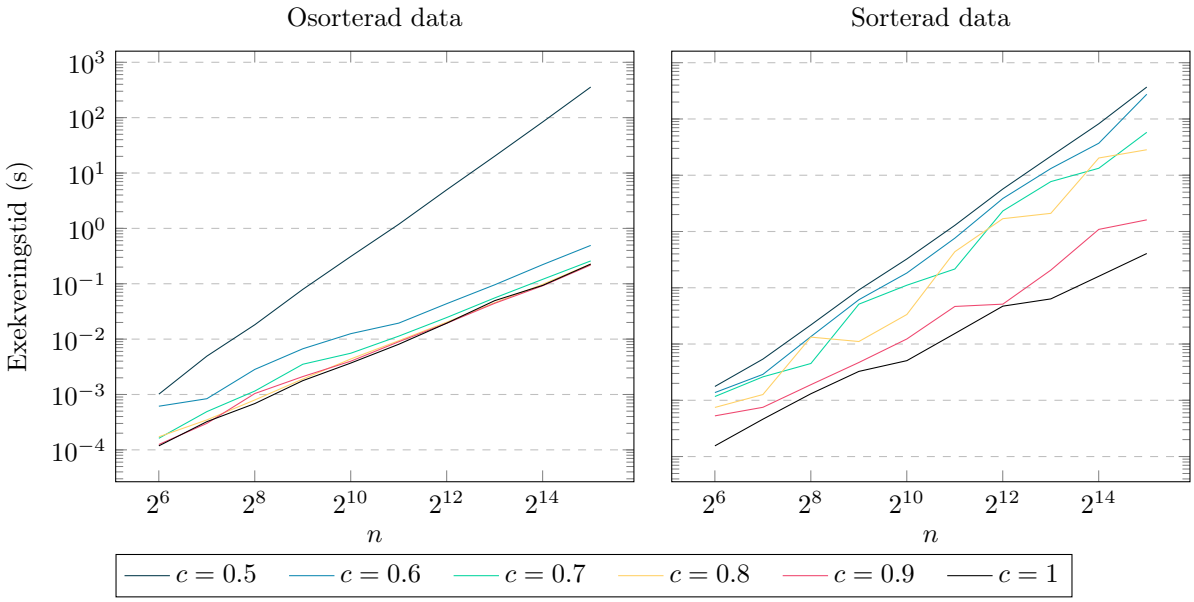
Implementera insättningsoperationen för trädet D. Vid varje insättning, kontrollera om storleksbegränsningarna är uppfyllda för varje nod. Om inte, ersätt det aktuella delträdet med ett perfekt balanserat binärt sökträd.

- Skapa olika testscenarier för att undersöka trädet D och jämföra det med ett standard BST.
- Genomför insättningar med olika värden av konstanten c.
- Mät och registrera tidskomplexiteten för sökoperationer och insättningar.

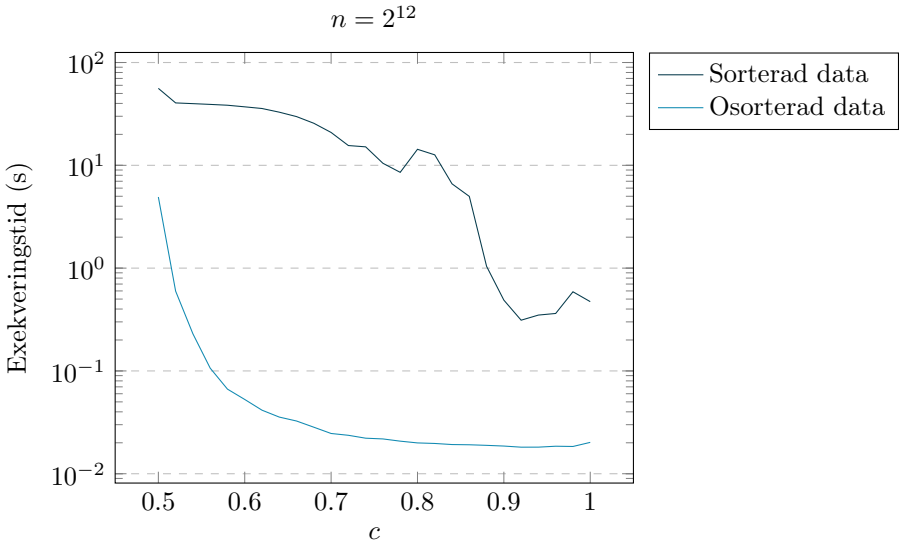
Utför experiment med både trädet D och ett standard binärt sökträd (BST). Under varierande förhållanden, särskilt med olika värden av konstanten c, samla in resultat och data för sökoperationer och insättningar. Mät och registrera tidskomplexiteten för sökoperationer och insättningar i båda träden under olika scenarier. Analysera hur prestandan varierar med olika värden av c och olika storlekar av träden.

4 Resultat

Flera olika tester genomfördes enligt metoden beskriven i del 3. Resultatet följer nedan i två figurer. Figur 1 visar den totala exekveringstiden för alla insert-operationer vid ett givet n och c. Figur 2 visar den totala exekveringstiden för alla insert-operationer vid ett bestämt $n = 2^{12}$, för alla värden av $c \in [\frac{1}{2}, 1]$ med ett steg av 0.02. Alla tester har körts med både helt slumpmässig data och sorterad data. För figur 2 har testerna repeterats 10 gånger för varje värde av c, och värdet som visas är snittresultatet av dessa.



Figur 1: Exekveringstid som funktion av n för olika värden av c



Figur 2: Exekveringstid som funktion av c

5 Diskussion

5.1 Felkällor

Begränsningar i datorresurser som minne och processorkraft kan påverka experimentens resultat, särskilt när det gäller skalbarhet. Kontrollera att datorresurserna är tillräckliga för att köra experimenten.

Valet av konstanten c kommer påverka trädet D's prestanda. Det är viktigt att testa tillräckligt många värden av c för att få insikt i hur trädet reagerar på olika storlekar och indata.

5.2 Slutsatser

Lite olika slutsatser kan dras av resultatet. Det första som är tydligt att se, framför allt i figur 2, är att värdet av c kan ha en enorm inverkan på exekveringstiden, framför allt när datan är sorterad. Den sorterade datan tar alltid mycket längre tid att inserta på vanligt sätt, av två anledningar:

1. Trädet som byggs är så obalanserat som möjligt, vilket gör att operationen[1] sker i $O(n)$ istället för $O(\log n)$. Detta beror på att sökandet efter den korrekta positionen att inserta på blir linjär (värdet vi ska inserta är alltid större än alla värden i trädet)

2. På grund av att trädet som byggs är så obalanserat som möjligt kommer ombalanseringar ske väldigt ofta, framför allt för låga värden av c. Ombalanseringen är dyr, då den alltid[2] sker i $O(n \cdot \log n)$, då hela listan av löv måste flyttas över i två separata sublistor, vilket sker $\log n$ gånger.

Den osorterade datan är helt slumpad, och kommer därför i genomsnitt skapa ett ganska balanserat träd även utan manuella ombalanseringar. Även detta märks tydligt i figur 2 genom att tidsfördelen med ett högt c-värde snabbt avtar över $c \approx 0.6$. Med låga c-värden sker ombalanseringen onödigt ofta, då trädet som skapas blir aningen slumpat och inte helt perfekt, men när lite "marginal" tillåts (d.v.s. högre värden av c) kommer ombalanseringar ske sällan då trädet som skapas ofta blir balanserat nog. Dessutom sker balanseringarna antagligen oftare på djupare nivå i trädet, vilket gör de ombalanseringar som faktiskt sker mycket billigare.

[1] Operationen sker alltid i $O(n)$ när $c = 1$. För andra värden på c kommer trädet tidvis att ombalanseras, vilket för söktiden närmare $O(\log n)$, då majoriteten av trädet "sprids ut på bredden". Trädets djup efter en ombalansering är aldrig mer än $\log n$.

[2] Värt att nämna är att *hela* trädet inte alltid ombalanseras, utan potentiellt bara ett subträd någonstans i trädet. Ombalanseringen sker alltså i $O(n \cdot \log n)$, där n är antalet löv i subträdet som ombalanseras.