Lab 2 TDA416

Lucas Ruud (lucasr), Niklas Baerveldt (nikbae)

February 2018

1 Resultat

Text	1	2	3	4	5
BST	74.860 (0ms - 10ms)	45.966 (0ms - 10ms)	17.796.870 (0ms - 10ms)	1.645.263 (0ms - 10ms)	1.661.868 (0ms - 10ms)
AVL	52.592 (0ms - 10ms)	31.186 (0ms - 10ms)	111.735 (0ms - 10ms)	40.969 (0ms - 10ms)	41.104 (0ms - 10ms)
SLC	2.996.585 30ms	425.266 (0ms - 10ms)	17.796.870 200ms	1.645.263 10ms	1.661.868 10ms
SPL	47.115 (0ms - 15ms)	13.610 (0ms - 15ms)	13.775 (0ms - 15ms)	5.493 (0ms - 15ms)	17.506 (0ms - 15ms)

Siffrorna högst upp i tabellcellerna står för hur många jämförelser som utfördes av en viss datastruktur och på vilken textfil. Siffrorna längst ner i tabellcellerna står för hur lång tid det tog att beräkna allt. I vissa celler så står det t.ex. 0ms - 10ms, detta betyder att det lägsta värdet som uppmättes när vi körde algoritmerna var 0ms och det näst lägsta 10ms.

2 BST

Det binära sökträdet, förkortat BST är en trädstruktur där varje nod i trädet har mellan 0 till 2 barn. Det är totalt ordnat, vilket innebär att ett element som är mindre är roten blir ett vänsterbarn till roten och alla element som är större blir ett högerbarn till roten.
Fördelarna med ett binärt sökträd är att operationer som t.ex. insert() och och find() teoretiskt sätt borde tillhöra $O(log(n))$ där n är antalet noder i trädet, detta innebär att man borde kunna utföra operationer på trädet relativt snabbt även om det är av en större storlek.
En annan fördel är, som tidigare nämnt att trädet är totalt ordnat, vilket underlättar när man vill traversera trädet.
Dock har BST nackdelen att det inte är självbalanserande, detta medför att de operationer som borde tillhöra $O(log(n))$ inte gör det, utan snarare tillhör $O(n)$ (linjär) tidskomplexitet. Detta beror på hur trädet ser ut. Om det t.ex. endast finns vänsterbarn så blir trädet i princip en dubbellänkad lista och då tillhör t.ex. operationen find() $O(n)$ i tidskomplexitet.

2.1 Resultat av BST

Om man tittar på de resultat som vi fick när vi analyserade olika texter med BST-strukturen så ser vi att det blir ett stort hopp i antalet jämförelser i text 3 (17.796.870). Om man kollar på texten i text 3 så ser man att det är en lång lista av ord som kommer i bokstavsordning. Detta medför att BSTn endast kommer att få högerbarn och när man vill söka i listan måste man börja från början varje gång. P.g.a detta kommer BSTn då att tillhöra tidskomplexiteten $O(n)$ vilket blir sämre ju större trädet blir. På text 4 är resultatet bättre jämfört med text 3, om man kollar på innehållet i text 4 så ser man att det är som i text 3 en lång lista med ord i alfabetisk ordning dock finns varje ord flera gånger. Detta innebär att om man söker på ett visst ord som dyker upp flera gånger i trädet så kommer den första instansen av ordet att ”väljas ut”. Det blir dock svårare att söka på ord som kommer efter det upprepade ordet för att man då måste traversera flera noder med samma innehåll, vilket leder till att antalet jämförelser då går upp.

3 AVL

Ett AVL-träd är egentligen bara ett binärt sökträd, dock så har det en stor fördel jämfört med BST, nämligen att det är självbalanserande. Detta innebär att trädet kommer att ”rotera” elementen för att minska sin totala höjd. P.g.a att den är självbalanserande så undviker den det värsta fallet som kan uppstå i ett BST, nämligen då t.ex.de mesta eller alla barn är t.ex. vänsterbarn. Operationer som find(), insert() osv kan då alltid antas tillhöra tidskomplexiteten $O(log(n))$ för att trädet kommer alltid att ha en ungefär perfekt struktur, alltså lika många höger- och vänsterbarn.

3.1 Resultat av AVL

Om man tittar på resultaten från AVL trädet och jämför de med resultaten från BST så ser man att AVL trädet utförde mindre jämförelser på varje textfil. Man kan också se att det blir en stor skillnad på text 3 mellan BST och AVL. BST utförde 17.796.870 jämförelser medans AVL-trädet endast utförde 111.735 jämförelser, vilket är en markant skillnad. Detta är just p.g.a att AVL-trädet roterar sina element för att få en bättre struktur. Så man kan anta att istället för att ha i princip en lista som BST hade så hade AVL-trädet en trädliknande (triangel) struktur. När sökningen i trädet sedan utfördes eliminerades ca. hälften av elementen i början för de var antingen större eller mindre än det sökta elementet.
Tittar man på tiderna för AVL-trädet så ser man att de är lika med de tider som man fick med BST. Detta kan bero på att både AVL-trädet och BST fortfarande är träd och därför tar det ungefär lika lång tid att utföra alla beräkningar trots att BST utför mer jämförelser. Dock så känns detta orealistiskt då flera operationer, som t.ex. jämförelser, borde medföra en längre körtid. Det är möjligt att det är rotationerna som AVL utför vid varje hämtning som gör att tiden blir samma som för BST.

4 SLC

Den sorterade länkade samlingen (SLC) är inget träd utan som namnet antyder, en lista. Utöver detta är den också enkelriktad. En fördel med detta är att man får "direktaccess" till alla element i listan, alltså man kan säga "ge mig det 4:de elementet" så får man det. Dock är detta inte en bra datastruktur när man vill söka i listan för man då måste börja från det första elementet och söka bakåt varje gång. Det underlättar lite att listan är sorterad med det minsta elementet längst fram och större element efter det. Dock blir det inte mycket bättre då man måste fortfarande jämföra det elementet man söker efter med varje element i listan och se om de matchar eller inte, vilket fortfarande medför att man måste gå igenom, i värsta fall alla element i listan. Man kan därför anta att en find-operation tillhör tidskomplexiteten $O(n)$, där n är antal element i listan.

4.1 Resultat av SLC

Resultaten an SLC strukturen är ganska dåliga om man jämför med de tidigare strukturerna (BST och AVL). Antalet jämförelser är högre än eller lika med de från BST med respektive textfil. Detta är på grund utav att BST inte är självbalanserande som AVL-trädet är och därmed kan det uppstå situationer då ett BST kan liknas med en lista. Detta blir tydligt om man jämför resultatet på de 3 sista textfilernas resultat på BST och SLC. Antalet jämföreslser här är exakt lika och detta beror på att i just detta fallet så får BST en struktur då alla barn är antingen höger- eller vänsterbarn. Vilket då skulle bli en lista av element i någon ordning. Om man går in i textfilerna och kollar på innehållet så ser man att det är en lista av ord som är sorterade i bokstavsordning. Detta leder till att BST endast skulle få högerbarn och det är ekvivalent med en lista som är sorterad med det minsta elementet längst fram. Alltså skulle tidskomplexiteten här tillhöra $O(n)$. Antalet jämförelser mellan text 1 och text 2 varierar ganska kraftigt i denna strukturen, tittar man på text 1 och text 2 så ser man att text 1 är en "vanlig" text medans text 2 är en lista med ord med dubbletter. Text 1 gör att man måste utföra flera jämförelser när man vill sätta in ett element i SLCn eftersom måste man kolla om elementet framför är mindre än eller större än och sedan fortsätta så tills man hittar rätt plats för elementet. I text 2 så finns det flera stycken dubbletter, det gör det lättare att hitta rätt plats då man sätter in elementet direkt när man hittar ett element som är lika med eller större än elementet som man vill sätta in. Detta sker även i text 1 men dubbletterna uppträder inte lika frekvent utan när en dubblett dyker upp så finns det redan massa andra element i kön som måste jämföras först, detta händer inte lika ofta med text 2. Det går snabbare att sätta in nya element om det finns dubbletter i samlingen, då SLCs add metod är implementerad på ett sätt så att den sätter in elementet före sin dubblett.

Körtiderna för SLC är även de inte väldigt imponerande, jämför man dem med de från BST så är de nästan alltid sämre. Detta kan bero på att trots att BST beter sig som en lista i speciella fall så utför den inte samma operationer

som en ”riktig” lista skulle göra (dessutom så är den dubbellänkad medans SLC är enkellänkad). Detta kanske medför att t.ex. find-operationen som utförs i BST är mer effektiv än den som utförs i SLC. Man kan bekräfta att SLC generellt sett är sämre än BST när det kommer till sökning om man kollar på resultatet för text 3 för BST och SLC. Både BST och SLC utförde samma anta jämförelser (17.796.870) dock så tog BST mindre tid på sig 10ms jämfört med 200ms för SLC.

5 SPL

Ett splay träd (SPL) är även detta ett binärt sökträd i grunden, dock så har det den speciella egenskapen att det flyttar (splayar) det element som man vill operera på till toppen av SPL-trädet, alltså till roten. Detta görs under antagandet att om man utför en operation på ett element så vill man förmodligen utföra några andra operationer på samma element snart. Eftersom man har splayat upp elementet till roten av trädet så blir det väldigt lätt att hitta det när man vill utföra de följande operationerna. Alltså tillhör SPL-trädet $O(log(n))$ när man söker efter ett slumpmässigt element, dock tillhör det $O(1)$, konstant tid, om man söker på samma element igen.

5.1 Resultat av SPL

Av resultaten att döma är SPL-trädet bäst av alla de andra datastrukturerna, antalet jämförelser i alla textfiler är lägre än alla de andra resultaten. Detta är för att när man väl en gång har sökt efter ett element så ligger det i toppen av trädet, om man sedan vill komma åt det igen så krävs det inte många jämförelser för att hitta det. Man kan se en stor skillnad på antalet jämförelser på text 3 för SPL jämfört med t.ex. BST och detta är just på grund utav att elementet som man söker redan ligger i eller nära toppen och då krävs det som sagt mindre jämförelser för att hitta det igen.
Tidsåtgången för SPL-trädet ligger lite högre än för AVL-trädet och BST, detta kan bero på att SPL-trädet måste splaya upp varje element som man söker efter till toppen av trädet vilket kan vara tidskrävande. AVL trädet utför även det rotationer dock utförs dessa rotationer inte hela tiden, utan bara när den totala höjden blir för hög.